

Guzsik Dániel

Munkafüzet

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>		
	Guzsik Dániel		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Guzsik, Dániel	2019. december 12.	

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	A Brun tételes feladat kidolgozása.	nbatfai
0.0.5	2019-03-01	Formális módosítások, Feladatok kidolgozása: Helló, Turing!	GuzsikD
0.0.6	2019-03-02	Formális módosítások, Feladatok kidolgozása: Helló, Turing!	GuzsikD
0.0.7	2019-03-03	Formális módosítások, Feladatok kidolgozása: Helló, Chomsky!	GuzsikD

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.8	2019-03-10	Formális módosítások, Feladatok kidolgozása: Helló, Caesar!	GuzsikD
0.0.9	2019-	Formális módosítások, Feladatok kidolgozása:	GuzsikD
0.0.10	2019-03-17	Formális módosítások, Feladatok kidolgozása: Helló, Mandelbrot!	GuzsikD
0.0.11	2019-03-24	Formális módosítások, Feladatok kidolgozása: Helló, Welch!	GuzsikD
0.0.12	2019-04-03	Formális módosítások, Feladatok kidolgozása: Helló, Conway!	GuzsikD
0.0.13	2019-04-10	Formális módosítások, Feladatok kidolgozása: Helló, Swarzenegger!	GuzsikD
0.0.14	2019-04-17	Formális módosítások, Feladatok kidolgozása: Helló, Chaitin!	GuzsikD

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	4
2. Helló, Turing!	6
2.1. Végtelen ciklus	6
2.2. Lefagyott, nem fagyott, akkor most mi van?	8
2.3. Változók értékének felcserélése	9
2.4. Labdapattogás	11
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	13
2.6. Helló, Google!	15
2.7. 100 éves a Brun téTEL	17
2.8. A Monty Hall probléma	18
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. Hivatalos nyelv	22
3.4. Saját lexikális elemző	23
3.5. l33t.l	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	32
4.4. C EXOR törő	34
4.5. Neurális OR, AND és EXOR kapu	36
4.6. Hiba-visszaterjesztéses perceptron	42
5. Helló, Mandelbrot!	43
5.1. A Mandelbrot halmaz	43
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	45
5.3. Biomorfok	48
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	57
6. Helló, Welch!	63
6.1. Első osztályom	63
6.2. LZW	66
6.3. Fabejárás	70
6.4. Tag a gyökér	74
6.5. Mutató a gyökér	78
6.6. Mozgató szemantika	79
7. Helló, Conway!	81
7.1. Hangyszimulációk	81
7.2. Java életjáték	83
7.3. Qt C++ életjáték	91
7.4. BrainB Benchmark	98
8. Helló, Schwarzenegger!	99
8.1. Szoftmax Py MNIST	99
8.2. Mély MNIST	99
8.3. Minecraft-MALMÖ	99

9. Helló, Chaitin!	100
9.1. Iteratív és rekurzív faktoriális Lisp-ben	100
9.2. Gimp Scheme Script-fu: króm effekt	102
9.3. Gimp Scheme Script-fu: név mandala	106
10. Helló, Gutenberg!	109
10.1. Juhász István, Magasszintű programozási nyelvek I.	109
10.2. Programozás bevezetés	117
10.3. Programozás	117
III. Második felvonás	121
11. Helló, Berners-Lee!	123
11.1. C++ és Java nyelvek összehasonlítása	123
11.2. Python	124
12. Helló, Arroway!	125
12.1. Az objektumorientált paradigma alapfoglamai. Osztály, objektum, példányosítás.	125
12.2. OO Szemlélet	125
12.3. Homokozó	127
12.4. „Gagyi”	135
12.5. Yoda	136
12.6. Kódolás from scratch	137
13. Helló, Liskov!	140
13.1. Liskov helyettesítés sértése	140
13.2. Szülő-gyerek	142
13.3. Anti OO	145
13.4. Ciklomatikus komplexitás	146
14. Helló, Mandelbrot!	149
14.1. Reverse engineering UML osztálydiagram	149
14.2. Forward engineering UML osztálydiagram	150
14.3. Egy esettan	151
14.4. BPMN	152

15. Helló, Chomsky!	154
15.1. Encoding	154
15.2. Full screen	155
15.3. Paszigráfia Rapszódia OpenGL full screen vizualizáció	157
15.4. Perceptron osztály	158
16. Helló, Stroustrup!	161
16.1. JDK osztályok	161
16.2. Másoló-mozgató szemantika	162
16.3. Hibásan implementált RSA törése	164
16.4. Változó argumentumszámú ctor	165
17. Helló, Gödel!	167
17.1. Gengszterek	167
17.2. C++11 Custom Allocator	168
17.3. STL map érték szerinti rendezése	170
17.4. Alternatív Tabella rendezése	170
17.5. GIMP Scheme hack	171
18. Helló, !	177
18.1. FUTURE tevékenység editor	177
18.2. OOCWC Boost ASIO hálózatkezelése	178
18.3. SamuCam	179
18.4. BrainB	179
19. Helló, Lauda!	181
19.1. Port scan	181
19.2. AOP	181
19.3. Android Játék	183
19.4. Junit teszt	185
20. Helló, Calvin!	187
20.1. MNIST	187
20.2. CIFAR-10	189
20.3. Android telefonra a TF objektum detektálója	190

IV. Irodalomjegyzék	192
20.4. Általános	193
20.5. C	193
20.6. C++	193
20.7. Lisp	193

Ábrák jegyzéke

2.1. Egy szál terhelése:	6
2.2. Egy szál terhelése altatással:	7
2.3. minden szál terhelése:	8
2.4. PageRank 4 honlapra felrajzolva	15
2.5. Táblázatban kiszámolva	15
2.6. A B_2 konstans közelítése	18
3.1. A turing gépes ábra:	20
4.1. Lokális memória nélküli neuron (perceptron)	37
4.2. OR logikai kapu	38
4.3. OR-AND logikai kapu	39
4.4. EXOR logikai kapu rejtett neuron nélkül	40
4.5. EXOR logikai kapu rejtett neuronokkal	41
5.1. Mandelbrot kétdimenziós koordinátarendszerben való ábrázolása	43
5.2. Mandelbrot 3.1.2 kimenete	48
5.3. Biomorf ábrázolása:	51
5.4. CUDA-s Mandelbrot halmaz váza, 600 x 600-as kép esetén:	52
5.5. CUDA-s kép kiszámítása ennyivel gyorsabb	55
5.6. Mandelbrot C nyelven:	56
5.7. Első nagyítás után:	56
5.8. Mandelbrot nagyítása JAVA környezetben	62
6.1. Binfa felépítése:	67
6.2. Emlékeztetőül az Inorder bejárásra:	71
6.3. Példa Postorder bejárásra:	73
12.1. PolarGen program eredménye	127

14.1. Visual Paradigm által létrehozott UML diagram	150
14.2. Visual Paradigm által Animal UML diagram	150
14.3. Esettan futtatva	152
14.4. BPMN egy üzleti tevékenységről	153
15.1. Encoding proba	154
15.2. Encoding siker	155
15.3. Labirintus	157
15.4. Paszigráfia Rapszódia OpenGL futtatás	158
15.5. Paszigráfia Rapszódia OpenGL	158
15.6. Perceptron futtatás	159
15.7. Perceptron kimenet	160
16.1. Futtatás után az új és régi kép	166
16.2. Mandelbrot.png képek mérete	166
20.1. Látható, hogy felismer egyszerre akár több tárgyat is	191
20.2. Csoda, hogy látok	191

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 alkalmi igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Minden esetre 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 gyereknek, 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 gyereknek, 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 (hogy lássuk más is) példával.

Hogyan nyomjuk?

Rántsd 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 dblatex 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
dblatex 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 'dblatex' 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 találod az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

Ne cifrázzuk: programok írása. Mik akkor a programok? Mit jelent az írásuk?

1.2. Milyen doksikat olvassak el?

- Kezd ezzel: <http://esr.fsf.hu/hacker-howto.html>!
- Olvasgasd aztán 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
- C kapcsán a [**KERNIGHANRITCHIE**] könyv adott részei.
- C++ kapcsán a [**BMECPP**] könyv adott részei.
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.
- Amiből viszont a legeslegjobban lehet tanulni, az a [The GNU C Reference Manual](#), mert gcc specifikus és programozókra van hangolva: szinte csak 1-2 lényegi mondat és apró, lényegi kódcsipetek! Aki pdf-ben jobban szereti olvasni: <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf>
- Az R kódok olvasása kis általános tapasztalat után automatikusan, erőfeszítés nélkül menni fog. A Python nincs ennyire a spektrum magától értetődő végén, ezért ahhoz olvasd el a [**BMECPP**] könyv - 20 oldalas gyorstalpaló részét.

1.3. Milyen filmeket nézzek meg?

- 21 - Las Vegas ostroma, <https://www.imdb.com/title/tt0478087/>, benne a **Monty Hall probléma** bemutatása.
- Kódjátszma, <https://www.imdb.com/title/tt2084970/>, benne a **kódtörő feladat** élménye.

- , , benne a 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.

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!

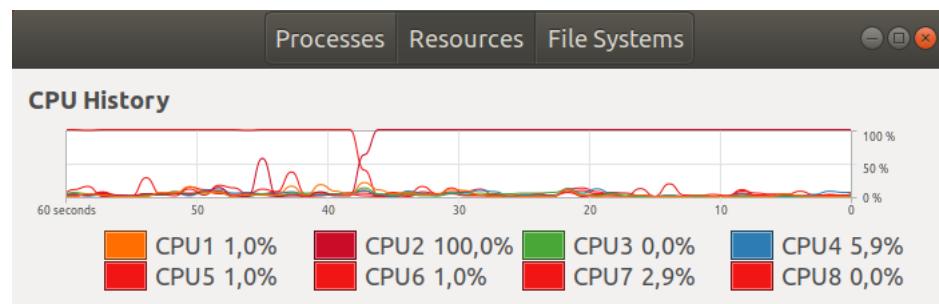
A feladatok közül a legegyszerűbb, amikor 100 százalékban dolgoztatunk egy magot. A kivitelezéshez nem kell más, mint egy végtelen ciklus. Ezt kétféle képpen is megcsinálom most, mégpedig for és while ciklusokkal is. Miközben a ciklusok futnak, közben egy mag próbálja befejezni a futásukat, így az adott mag pörög ahogyan csak bír.

For ciklussal:

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

While ciklussal:

```
int main()
{
    while(1);
}
```



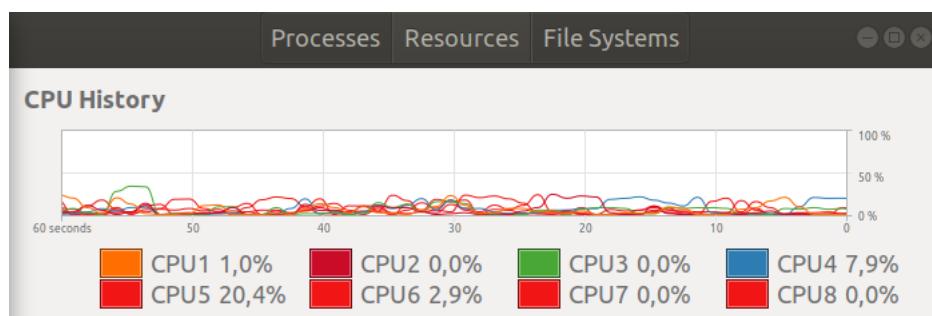
2.1. ábra. Egy szál terhelése:

Következő, hogy 0 százalékban dolgoztassuk a magot. Egy egyszerű sleep function-nal lehet megoldani ezt a feladatot.

```
#include <unistd.h>

int main()
{
    for (;;)
        sleep(1);

    return 0;
}
```



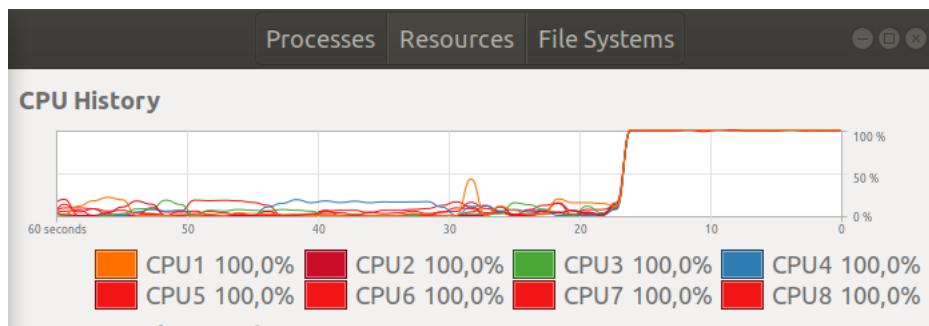
2.2. ábra. Egy szál terhelése altatással:

Mikor minden magot akarunk egyidőben 100 százalékosan futtatni, akkor a következő megoldás az egyik lehetséges megoldás. Egy végtelen ciklust ráeresztünk minden magra. A #pragma omp parallel segítségével eresztyük rá az összes szálra. Ha csak adott mennyiségű szálra akarjuk futtatni egyszerre, akkor #pragma omp parallel num_threads(adott_mennyiség) beírásával tudjuk kivitelezni. A program fordításakor egy -fopenmp kapcsolót is bele kell írni.

```
#include <unistd.h>
#include <omp.h>

int main()
{
    #pragma omp parallel
    for(;;);

    return 0;
}
```



2.3. ábra. minden szál terhelése:

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 pszeudódjára:

```
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 épőlő Lefagy2 már nem tartalmaz feltételezett, csak 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 násználata nélkül!

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

Változók felcserélése segédváltozó felhasználása és logikai utasítás nélkül kétféleképp lehetséges C nyelven.

Egyik megoldás, hogy a "kizáró vagy", EXOR vagy XOR művelet segítségével cseréljük meg az adott változókat. Fontos megjegyezni, hogy ez csak számoknál fog működni, karaktereknél és stringeknél nem.

```
#include <stdio.h>

int main()
{
    int a = 2;
    int b = 9;
    printf("a = %d\nb = %d\n", a, b);

    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
    printf("Csere:\na = %d\nb = %d\n", a, b);

    return 0;
}
```

A megadott értékeket a \wedge operátor bitenként fogja összeadni. Azaz a számoknak a kettes számrendszerbeli értékükeikkel fogja a műveletet végrehajtani. Különböző bitértékek esetén 1-et, azonos értékek esetén 0-t ad vissza.

```
-----|a = 0 0 1 0| /*eredeti a*/
-----|b = 1 0 0 1| /*eredeti b*/

a = a ^ b ---->|a = 1 0 1 1| /*össze exoroztuk az a-t és b-t*/
b = a ^ b ---->|b = 0 0 1 0| /*megcserélt b*/
a = a ^ b ---->|a = 1 0 0 1| /*megcserélt a*/
```

A második módszer a C++ beépített swap függvényének segítségül hívása

```
#include <iostream>

using namespace std;

int main()
{
    int a = 2;
    int b = 9;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    swap( a, b );

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    return 0;
}
```

}

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 videókon.)

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

Megoldás if-et használva:

```
#include <stdio.h>
#include <curses.h>
#include <unistd.h>

int
main ( void )
{
    WINDOW *ablak; /*mutató az aktuális konzolablakra*/
    ablak = initscr (); /*amíg a program fut, ezt az ablakot használja*/

    int x = 0;
    int y = 0; /*labda kezdőpozíciók*/

    int xnov = 1;
    int ynov = 1; /*a léptetés mértéke*/

    int mx;
    int my; /*az ablak mérete*/

    for (;;)
    {
        getmaxyx ( ablak, my , mx ); /*az ablak aktuális méretét hívja be*/
        mvprintw ( y, x, "O" ); /*a labda kirajzoltatása*/

        refresh (); /*valós időben frissítjük az ablakot*/
        usleep ( 100000 ); /*mikrosec-ben mérve, altatjuk a programot*/
                           /*a labda sebességét is ez határozza meg*/
        clear (); /*csak az aktuális labda jelenjen meg*/

        x = x + xnov;
        y = y + ynov; /*mozgatjuk a labdát a koordináták növelésével*/

        if ( x>=mx-1 ) /*ha elértük a konzol jobb oldalát, visszafordulunk ←*/
        /*
        {
            xnov = xnov * -1;
```

```
    }
    if ( x<=0 ) /*ha elértek a konzol bal oldalát, visszafordulunk*/
    {
        xnov = xnov * -1;
    }
    if ( y<=0 ) /*ha elértek a konzol tetejét, visszafordulunk*/
    {
        ynov = ynov * -1;
    }
    if ( y>=my-1 ) /*ha elértek a konzol alját, visszafordulunk*/
    {
        ynov = ynov * -1;
    }
}
return 0;
}
```

Megoldás if nélkül:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

static void gotoxy(int x, int y)                      /*kurzor pozicionálása*/
{
    int i;
    for(i=0; i<y; i++) printf("\n");                  /*lefelé tolás*/
    for(i=0; i<x; i++) printf(" ");
    printf("o\n");
}

void usleep(int);
int main(void)
{

    int egyx=1;
    int egyy=-1;
    int i;
    int x=10;                                         /*a labda kezdeti pozíciója*/
    int y=20;
    int ty[23]; //magasság                         /*a pálya mérete*/
    int tx[80]; //szélesség

    /*pálya széleinek meghatározása*/

    for(i=0; i<23; i++)
        ty[i]=1;

    ty[1]=-1;
```

```
ty[22]=-1;

for(i=0; i<79; i++)
    tx[i]=1;

tx[1]=-1;
tx[79]=-1;

for(;;)
{
    (void)gotoxy(x,y);
/*printf("o\n"); Áthelyezve a gotoxy függvényve*/

x+=egyx;
y+=egyy;

egyx*=tx[x];
egyy*=ty[y];

usleep (100000);
(void)system("clear");
}

}
```

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. Használд ugyanazt a while ciklus fejet, amit Linus Torvalds a BogoMIPS rutinjában!

Tisztázzuk először, hogy mi is az a BogoMIPS?

A MIPS jelentés "Millions of Instructions Per Second", azaz feladatok milliói, amik egy másodperc alatt vannak végrehajtva. A "bogo" szó pedig arra utal, hogy ez nem egy valódi doleg, sokkal inkább egy kitaláció. Magyarul sajnos elég rosszul hangzik. Az ötlet Linus Torvalds fejéből pattant ki, mert kellett neki egy időzített loop, ami a processzor sebességét nézi meg bootolásnál. Először az 1.0-ás Linux kernelben jelent meg.

```
static void calibrate_delay(void)
{
    int ticks;

    printk("Calibrating delay loop.. ");
    while (loops_per_sec <= 1) {
        ticks = jiffies;
        __delay(loops_per_sec);
        ticks = jiffies - ticks;
```

```

if (ticks >= HZ) {
    __asm__("mull %1 ; divl %2"
        : "=a" (loops_per_sec)
        : "d" (HZ),
        "r" (ticks),
        "0" (loops_per_sec)
        : "dx");
    printk("ok - %lu.%02lu BogoMips\n",
        loops_per_sec/500000,
        (loops_per_sec/5000) % 100);
    return;
}
printf("failed\n");
}

```

Most pedig nézzük meg, hogy milyen hosszú lehet egy gépi szó.

```

#include <stdio.h>

int main(void)
{
    int s=0;
    int c=1;
    while(c != 0)
    {
        c <= 1;
        ++s;
    }
    printf("Egy gépi szó maximális hossza %d karakter.\n", s );
}

```

Az s változóban fogjuk számolni a lépéseket 0-tól indulva. Tudjuk, hogy a számítógép kettes számrendszerben gondolkodik. Így képzeljük el, hogy c változóból értékünk kettes számrendszerben van ábrázolva. A kérdés az az, hogy egy int változó mekkora felületű, azaz hány bit értékű? Egy számot úgy tárol a gép, hogy a szám kettes számrendszerbeli értékét veszi és azt teszi a hely végére. Az elejét pedig feltölti megfelelő mennyiségű nullával. A program akkor fog jól működni, ha olyan számot adunk meg aminek a kettes számrendszer szerinti legkisebb helyiértékű száma 1-es. Tehát páratlan számokkal fog működni helyesen.

A while ciklusban addig megyünk, amíg el nem érjük a c változónk kettes számrendszerbeli utolsó 1-es karakterét. Ekkor minden bit 0 értéket vesz fel, tehát a változónk értéke is 0 lesz. Amíg ezt el nem érjük, a léptetés közben egyesével növeljük az s változónkat. Íg a végén megkapjuk a maximális szóhosszt.

Kimenetünk a következő: Egy gépi szó maximális hossza 32 karakter. Tehát 32 hosszúságú egy int változó. Így legnagyobb számunk a $(2^{32}) - 1$ lehet.

Az előbb int változó hosszát néztük meg, most nézzük meg a long int változó hosszát

```

#include <stdio.h>

int main(void)

```

```
{
    int s=0;
    long int c=1;
    while(c != 0)
    {
        c <= 1;
        ++s;
    }
    printf("Egy gépi szó maximális hossza %d karakter.\n", s );
}
```

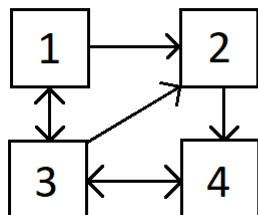
Kimenetünk a következő: Egy gépi szó maximális hossza 64 karakter. Tehát 64 hosszúságú egy long int változó, azaz kétszer hosszabb mint egy int. Így legnagyobb számunk a $(2^{64}) - 1$ lehet.

2.6. Helló, Google!

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

A Larry Page által kifejlesztett PageRank algoritmus nem csinál másit, mint megmondja nekünk, hogy egy honlap milyen minőségű, a rá mutató honlapok számának és minőségének függvényében.

Megoldás Móricka-rajz:



$$\text{PR}(h_2) = \sum_{B(h_2)} \frac{\text{PR}(4)}{N(4)}$$

2.4. ábra. PageRank 4 honlapra felrajzolva

A felvázolt állapot táblázatos formában az algoritmussal kiszámolva:

	A	i(1)	i(2)	PR	
1	1/4	1/12	1.5/12	1	
2	1/4	2.5/12	2/12	2	A: alaphelyzet
3	1/4	4.5/12	4.5/12	3	i: iterációk
4	1/4	4/12	4/12	4	PR: Page Rank

2.5. ábra. Táblázatban kiszámolva

Az alábbi C programkódot futtatva számolhatjuk ki az értékeket:

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

void
kiir (double tomb[], int db)
{
    int i;
    for (i=0; i<db; i++)
        printf("PageRank [%d]: %lf\n", i, tomb[i]);
}

double tavolsag(double pagerank[],double pagerank_temp[],int db)
{
    double tav = 0.0;
    int i;
    for(i=0;i<db;i++)
        tav +=abs(pagerank[i] - pagerank_temp[i]);
    return tav;
}

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

    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};

    long int i,j,h;
    i=0; j=0; h=5;

    for (;;)
    {
        for(i=0;i<4;i++)
            PR[i] = PRv[i];
        for (i=0;i<4;i++)
        {
            double temp=0;
            for (j=0;j<4;j++)
                temp+=L[i][j]*PR[j];
            PRv[i]=temp;
        }
        if ( tavolsag(PR,PRv, 4) < 0.00001)
```

```
        break;
    }
    kiir (PR, 4);
    return 0;
}
```

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/nbatfai/bhax/blob/master/attention_raising/Primek_R

Mi is a Brun téTEL?

A Brun téTEL szerint ezeknek az ikerprímeknek a reciprokösszegük egy konstanshoz konvergálnak(közeliTENEK).

Prímszám az a termézeszetes szám, amely csak önmagával és eggyel osztható. Eukleidész görög matematikus már Krisztus előtt tudta, hogy végtelen sok prímszám van, de ma sem tudja senki, hogy végtelen sok ikerprím van-e. Két prím ikerprím, ha különbségük 2.

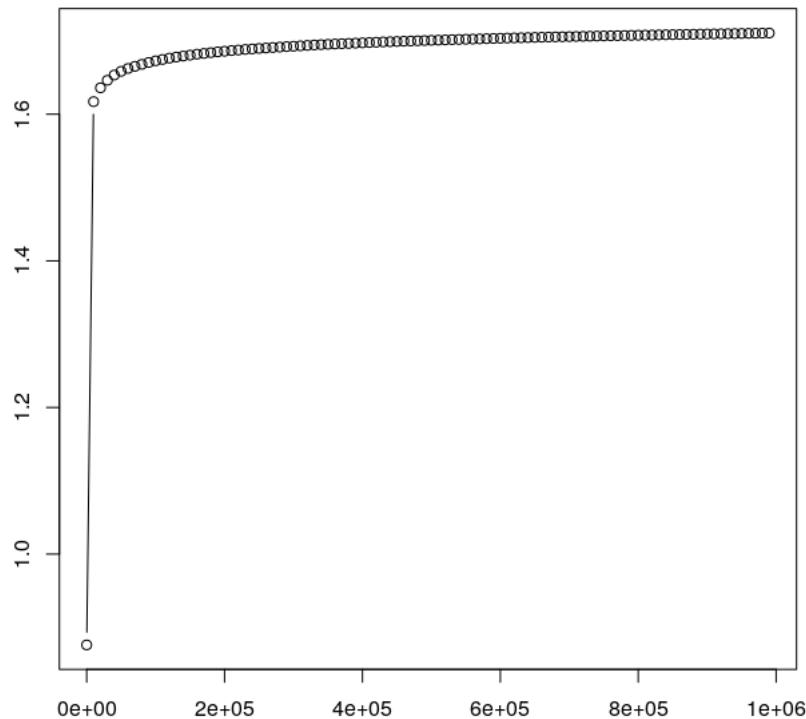
```
library (matlab)

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 következő ábra azt mutatja, hogy a szumma értéke, hogyan nő, egy határértékhez tart, a B_2 Brun konstanshoz. Ezt ezzel a csipettel rajzoltuk ki, ahol először a fenti számítást 13-ig végezzük, majd 10013, majd 20013-ig, egészen 990013-ig, azaz közel 1 millióig. Vegyük észre, hogy az ábra első köre, a 13 értékhez tartozó 1.044023.



2.6. ábra. A B_2 konstans közelítése

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

Let's Make a Deal című showban a műsorvezető Monty Hall döntések elő állította a show részvevőit. 3 ajtó közül kellett választaniuk egyet. Két ajtó mögött egy-egy kecske, míg az egyik mögött viszont a főnyeremény volt. A játékosnak egy ajtót kellett kiválasztania.

Miután a játékos választott egy ajtót a műsorvezető kinyitott egy ajtót, ami mögött kecske lapult és nem a játékos által választott ajtót nyitotta ki természetesen. Ekkor a játékos előtt két zárt ajtó maradt és ezek közül újra választhatott. Választhatta az elsőként választott ajtót, de választhatta másik ajtót ami ezen kívül még zárva volt.

A Monty Hall paradoxon szerint ha ekkor (a második körben, amikor már csak két ajtó közül kellett választani) a játékos megváltoztatja a döntését és az eredetileg választott ajtó helyett a másikat nyitja ki, akkor az esélyei megduplázódnak.

A műsorvezetőről elnevezett paradoxon is ezen kérdések egyikén alapszik.

A paradoxon bizonyításához szükséges R kód a következő:

```
kiserletek_szama=10000000
```

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

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)]

}

nemvaltoztatesnyer= which(kiserlet==jatekos)
valtoztat=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama) {

  holvalt = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i]))
  valtoztat[i] = holvalt[sample(1:length(holvart),1)]

}

valtoztatesnyer = which(kiserlet==valtoztat)

sprintf("Kiserletek szama: %i", kiserletek_szama)
length(nemvaltoztatesnyer)
length(valtoztatesnyer)
length(nemvaltoztatesnyer)/length(valtoztatesnyer)
length(nemvaltoztatesnyer)+length(valtoztatesnyer)
```

3. fejezet

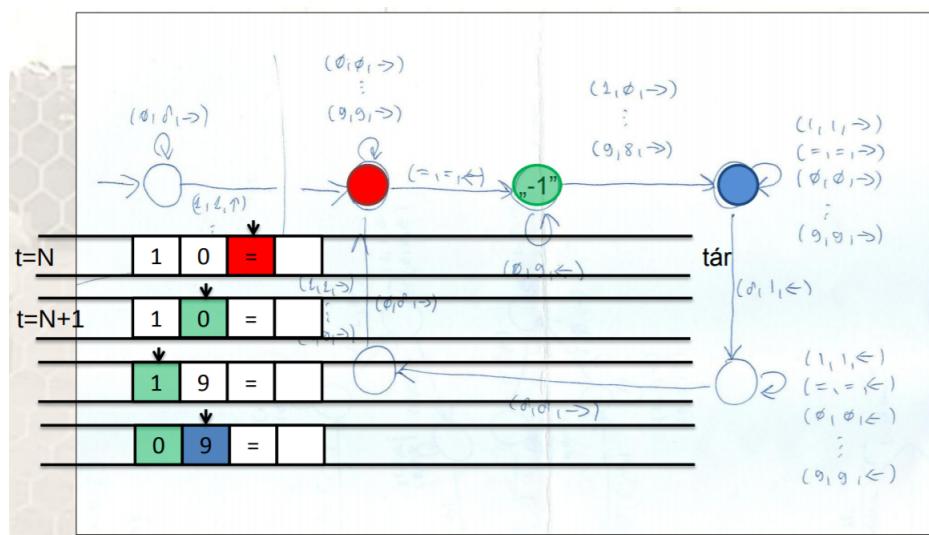
Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Az unáris számrendszer nagyon egyszerűen működik. Kiválasztunk egy szimbólumot, amivel későbbiekben ábrázolni fogjuk a számokat. A mi szimbólumunk "|" ez lesz most. Amennyit ér az átváltandó bináris szám, annyiszor kell kiírni egymás után a "|" szimbólumot. Például a 2 szám unárisan || alakú lesz. A 7-es szám pedig ||||| alakú. Ezt a számrendszeret elég nehézkes kiolvasni főleg nagy számok esetén.

Lehet egyszerűsíteni azért valamennyire mégpedig a következőképpen: minden egyes helyiértékű számhoz más szimbólumot vezetünk be. Tegyük fel, hogy az 1-es helyiértékű számok szimbóluma "|", a 10-es helyiértékűek "*", a 100-as helyiértékűek pedig "+". Ezekkel dolgozva a 231 szám a következő képpen fog kinázni unárisban: ++**|



3.1. ábra. A turing gépes ábra:

A fenti turing decimális számok unárisba való átváltását hajtja végre. A működése nagyon egyszerű. A

bemenetre adott számból kivon 1-et egészen addig, míg az értéke 0 nem lesz. minden egyes kivonásnál egy karakterrel növeli a unáris számrendszerbeli kimenetünket.

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

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

A formális nyelvtan: A matematikában, informatikában és a nyelvészettel egy formális nyelv olyan szavakból, áll, melyeket egy abc-ből vett, jól definiált, bizonyos szabályoknak megfelelő betűk alkotnak. Egy formális nyelv abc-je szimbólumokból, betűkből és tokenekből állhat, melyek strinengekké állnak össze.

A formális nyelvtan 2 nagy csoportja, az analitikus és generatív nyelvtan.

Analitikus nyelvtan esetében, a nyelvtani szabályokat alkalmazva "csupán" azt tudjuk eldönteni, hogy a kiértékelni kívánt karaktersorozat eleme-e a nyelvnek, vagy sem. (Igaz vagy Hamis értéket kapunk vissza)

A generatív nyelvtan, olyan nyelvészeti szabályok összessége, amely a nyelvtant olyan szabályrendszernek tekinti, amely pontosan azon a szavak kombinációit generálja, amelyek az adott nyelvben grammatikus mondatokat alkotnak. A kifejezést Noam Chomsky használta először az 50-es évek végén.

A Chomsky-féle nyelvi hierarchia a következőképpen alakul:

- 3. típusú nyelvek: reguláris
- 2. típusú nyelvek: környezetfüggetlen
- 1. típusú nyelvek: környezetfüggő
- 0. típusú nyelvek: rekurzívan felsorolható

1. Példa $L = \{ a^n b^n c^n \mid N > 0 \}$:

Szabályok:

```
S -> abc vagy aSQ  
bQc -> bbcc  
cQ -> cc  
cc -> Qc
```

1. lépés. Állítsuk elő a megfelelő számú a-t:

```
S --> aSQ --> aaSQQ --> aaabcQQ -->
```

2. lépés. Alkalmazzuk az átalakítások szabályait:

```
--> aaabccQ --> aaabQcQ  
aaabQcc --> aaabbccc --> aaabbQcc --> aaabbccc
```

2. Példa $L = \{ a^n b^n c^n \mid N > 0 \}$:

- minden "A"-t átalakítunk "a"-ra.

- minden "B"-t átalakítunk "b"-re.
- minden "C"-t átalakítunk "c"-re.

Szabályok:

$S \rightarrow aSBC$
 $S \rightarrow \beta$ (a béta karakter üres stringet fog visszaadni)

változók cseréje, ha szükségünk lesz rá:

$CB \rightarrow HB$
 $HB \rightarrow HC$
 $HC \rightarrow BC$

$aB \rightarrow ab$
 $bB \rightarrow bb$
 $bC \rightarrow bc$
 $cC \rightarrow cc$

1. lépés. Állítsuk elő a megfelelő számú a-t, B-t és C-t. (még egyelőre nem lesznek jó sorrendben)

$S \rightarrow aSBC \rightarrow aaSBCBC \rightarrow aaaSBCBCBC \rightarrow aaa\beta BCBCBC \rightarrow$

2. lépés. Elérjük, hogy az összes B a C-k elő kerüljön.

$aaaBCBCBC \rightarrow aaaBBBCCC \rightarrow$

3. lépés. Végül átalakítjuk az összes változót konstanssá.

$aaaBBBCCC \rightarrow aaabbbcc \rightarrow$

Megoldás forrása: https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_1.pdf

3.3. Hivatkozási nyelv

A [KERNIGHANRITCHIE] könyv C referencia-kézikönyv/Utasítások melléklete alapján definiál 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:

Rövid kódok, amik C89-es szabvánnyal nem fordulnak, de C99-el már igen.

C99-nél már lehetséges dupla '/' jel után kommentet írni, még C89-nél ez hibát jelent.

```
#include <stdio.h>

int main(void) //na itt a bibi
{
    printf("Hello World\n");
    return 0;
}
```

A másik ilyen hiba lehet, ha for ciklus utasításán belül próbálunk meg változót deklarálni c89 szabvánnyal:

```
#include <stdio.h>

int main()
{
    for (int i = 0; i < 5; ++i)
    {
        printf("%d\n", i );
    }
    return 0;
}
```

Viszont ez nem jelenthet hibát, ha c99 szabvánnyal fordítunk

Fordítás a következőképpen működik: gcc programnev.c -o programnev ehhez pedig hozzá fűzni még egy -std=c89 vagy -std=c99 kapcsolót.

3.4. Saját lexikális elemző

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

Lexikális analízisnek hívjuk azokat a folyamatokat, amikor az inputon lévő karaktersorozatokat tokenek sorozatává alakítjuk át. Lexernek hívjuk azokat a programokat, amik az előbb leírt lexikális analízist hajtják végre.

Az alább leírt L kódunkból az általunk használt lexer egy C kódot fog készíteni, amit lefordítunk és futtatunk.

A kód első részét a Lexer beleteszi az általa készített C kódba. Itt számoltatjuk meg a szövegben talált valós számokat. Ezen felül itt adjuk meg, hogy mit is keresünk pontosan. Esetünkben a 0 és 9 közötti számokat keressük.

```
%{
#include <stdio.h>
int valos_szamok = 0;
%}
szam [0-9]
```

A második rész a fordítási szabályokat tartalmazza. Az általunk keresett karakter lánc a szabályok szerint a következőképpen néz ki:

A karakterlánc elejét tetszőleges számú (lehet 0 darab is '*) egyjegyű szám képzi. Majd ezt követően a lánc második felén, egy minimum egy hosszúságú szám sorozat jöhet. A második fele vagy létezik vagy nem, ezt a ? operátorral fogalmaztuk meg és mindenféle képpen ':-tal kell kezdődnie, ha létezik.

```
%%
{szam}*(\.{szam}+)? {++valos_szamok;
    printf("[valoszz=%s %f]", yytext, atof(yytext));}
%%
```

A program utolsó részével hívjuk meg a lexikális elemzést. Miután ez lefut, kiíratjuk az inputon beérkezett valós számok darabszámát.

```
int main()
{
    yylex ();
    printf("Valós számok darabszáma: %d\n", valos_szamok);
    return 0;
}
```

Telepítsd a lexert a következőképpen: sudo apt install flex

Majd a következő parancsokkal fordítsd le, végül futtasd a kapott C fájlt:

```
$ lex -o realnumber.c realnumber.l
$ gcc real.c -o lex -lfl
$ ./lex
```

Megoldás videó: https://www.youtube.com/watch?v=9KnMqrkj_kU

Megoldás forrása: https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_1.pdf

3.5. l33t.l

Lexelj össze egy l33t cipher-t!

A cipher algoritmusokat kódolásra és dekódolásra használják. Kettéoszthatjuk őket azzal, hogy a dekódolási-kódolási folyamat során ugyanazt a titkos kulcsot használja-e az algoritmus, vagy két különbözőt a két folyamathoz.

Az l33t röviden és tömören egy szleng. A neve a működéséből adódóan jöhetett létre. Lényege, hogy a szöveget alkotó karaktereket lecserélik, formailag hasonló, de kissé szokatlan karakterekre, ezzel minimálisan torzítve, esetleg rövidítve az eddig megszokott abc-t és nyelvtant.

Az előző feladathoz hasonlóan, most is 3 nagyobb részből épül fel a programunk.

Létrehozunk egy konstanst L337SIZE néven. Ez a szám az l337dlc7 tömb méretét osztva a cipher struktúra bájtból megadott méretével lesz egyenlő. Ezek után létrehozunk egy struktúra típust. Ebben fogjuk beolvasni a karaktereket. Továbbá itt fogjuk definiálni a beolvasott karakterekhez rendelhető karakterkészleteket is. Egy karaktert egy stringgel fogunk behelyettesíteni.

```
% {
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>

#define L337SIZE (sizeof l337dlc7 / sizeof (struct cipher))

struct cipher {
    char c;
    char *leet[4];
```

```
    } l337d1c7 [] = {
    /*itt lesz 36 sornyai definíció*/
    };
%
```

A 36 sornyai definíció alapjául a következő Wikipedia oldal szolgál: [Wikipedia leet karakterek helyettesítése](#)
A második részben, az L337SIZE tömb sorait járjuk végig, hogy megtaláljuk a behelyettesíteni kívánt karakter sorát. Ha megtaláltuk a kívánt karaktert, akkor egy véletlenszerű szám segítségével program eldönti, hogy a 4 lehetséges string helyett, melyiket helyettesítse be.

```
%%
. {

    int found = 0;
    for(int i=0; i<L337SIZE; ++i)
    {

        if(l337d1c7[i].c == tolower(*yytext))
        {

            int r = 1+(int) (100.0*rand() / (RAND_MAX+1.0));

            if(r<70)
                printf("%s", l337d1c7[i].leet[0]);
            else if(r<80)
                printf("%s", l337d1c7[i].leet[1]);
            else if(r<90)
                printf("%s", l337d1c7[i].leet[2]);
            else
                printf("%s", l337d1c7[i].leet[3]);

            found = 1;
            break;
        }

    }

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

}
%%
```

Az utolsó rész sorai a C forráskódot fogják alkotni. Itt inicializáljuk a véletlen-szám generátort, indítjuk a lexikális elemzést. A program a standard inputot fogja olvasni.

```
int
main()
{
    srand(time(NULL)+getpid());
```

```
yylex();  
return 0;  
}
```

A programot ugyan úgy fogjuk futtatni, mint az előző feladatban, azaz:

```
$ lex -o leet.c leet.l  
$ gcc leet.c -o leet -lfl  
$ ./leet
```

Megoldás videó: https://www.youtube.com/watch?v=06C_PqDpD_k

3.6. A források olvasása

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

```
if(signal(SIGINT, jelkezelo)==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 jelkezelo 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 megy ránézésre, elkapja valamelyiket esetleg a *splint* vagy a *frama*?

i.

```
if(signal(SIGINT, SIG_IGN) !=SIG_IGN)  
    signal(SIGINT, jelkezelo);
```

Ha a SIGINT jel kezelése nem lett figyelmen kívül hagyva, akkor a jelkezelő függvénynek kell kezelnie azt.

ii.

```
for(i=0; i<5; ++i)
```

Ez egy for ciklus, amit 0-ról indítunk és 5-ször fut le, eggyel növelve az i értékét. Az *++i* egy preinkrementáló, mert i értékét még akkor növeli eggyel, mielőtt belépne a ciklusmagba.

iii.

```
for(i=0; i<5; i++)
```

Ebben a for ciklusban egy post inkrementációt használunk, ami annyit jelent, hogy az i változónk értéke a ciklusmag végén történik meg.

iv.

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

Ebben a for ciklusban egy tomb elemeinek sorszámát növelnénk 1-el. Legalábbis talán ez lenne a célja a költőnek. Ha a ciklusmagban kiíratjuk minden az i értékét, akkor láthatjuk hogy az első érték minden valamelyen memóriaszemét lesz. Kerülendő implementáció.

v.

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

Ebben a for ciklusban az i változónkat n-1-ig növeljük, de csak akkor, ha teljesül ($*d++ = *s++$) a feltétel is. A splinter kiabál, hogy a logikai operátor után nem boolean típusú érték jön.

vi.

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

Két decimális számot az f függvény segísgével kiakarunk íratni. A bökkenő, hogy a függvény argumentumainak sorrendje nincs előre meghatározva.

vii.

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

Két decimális számot íratunk ki. Az első az f függvény argumentuma lesz, az így kapott értéket kapjuk. A második pedig egyszerűen csak az a értékét kapjuk.

viii.

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

A kiíratásnál az első értékünk alakulása a következő: az f függvény megkapja az a változó címét, majd a függvény az erre a címre mutató pointer alatti értéket fogja vissza adni. Míg a második érték pedig az a lesz önmaga.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

```
$ (\forall x \exists y ((x < y) \wedge (y \text{ prim}) \wedge (S y \text{ prim})) \leftrightarrow ) $
```

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

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

I. Végtelen sok prímszám van.

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

II. Végtelen sok iker-prímszám van.

```
$ (\forall x \exists y ((x < y) \wedge (y \text{ prim}) \wedge (S y \text{ prim})) \leftrightarrow ) $
```

III. Véges sok prímszám van.

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

IV. Véges sok prímszám van.

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

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/MatLog_LaTeX

Megoldás videó: <https://youtu.be/ZexiPy3ZxsA>, https://youtu.be/AJSXOQFF_wk

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciajára
- egészek tömbje
- egészek tömbjének referenciajára (nem az első elemé)
- egészre mutató mutatók tömbje
- egészre mutató mutatót visszaadó függvény
- egészre mutató mutatót visszaadó függvényre mutató mutató
- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
- 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;
```

Egy int típusú a változót hozunk létre

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

Egy b mutató, ami a-nak a memóriacímét tartalmazza

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

Egy referencia változó, ami a-nak az értékét kapja meg

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

Egy 5 elemű, inteket tartalmazó tömb

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

Itt a tr megkapja a c tömb elemeit

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

EgészkeEgy függvény, ami egészekre mutató mutatót ad visszat tároló, tömbre mutató tömb

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

Egy függvény, ami egészekre mutató mutatót ad vissza

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

Egy függvény mutató, ami egészekre mutató mutatót ad vissza

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

Egy egéssel visszatérő 2 egészet váró függvényre mutató mutatóval visszatér " o 2 egészet váró függvény

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

Függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre mutató mutató.

4. fejezet

Helló, Caesar!

4.1. int *** háromszög mátrix

Részletenként haladva járjuk be a programot

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

int
main ()
{
    int sorok_szama = 5;
    double **tm;
```

Elsőként megadjuk, hogy hánnyal lesz a mátrixunk. Másodszor pedig deklarálunk egy 8 bájt méretű double típusú pointert, amin keresztül hivatkozni fogunk a mátrix soraiban található elemekre.

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

A mátrix minden sorához lefoglalunk egy-egy mutatót a memóriában, a malloc függvényteljesítéssel. A függvény által visszaadott mutatót double típusúra kényszerítjük. Ha ez nem jön össze, akkor kilépünk a programból a visszakapott null pointer miatt.

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

}
```

Majd a háromszög-mátrix struktúrájának alapján, minden sorban soronként egyel több helyet foglalunk le a memóriában. Itt már nem mutatók, hanem a mátrixunk értékei lesznek tárolva. Ha nem jön össze, itt is kilépünk a programból.

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

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

Egyesével feltöljtük a mátrixunkat lebegőpontos számokkal. Soronként, azon belül oszloponként haladva.

```
for (int i = 0; i < sorok_szama; ++i)
    free (tm[i]);

free (tm);

return 0;
}
```

Legvégül pedig ismét végiglépkedünk a korábban lefoglalt memóriaterületeken és felszabadítunk minden. Ezzel megakadályozva, hogy elfolyjon a memóriánk.

Megoldás videó 1: <https://www.youtube.com/watch?v=1MRTuKwRsB0>

Megoldás videó 2: <https://www.youtube.com/watch?v=RKbX5-EWpzA>

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Lépésről-lépéstre, kódcsipertről-kódcsipetre.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define MAX_KULCS 100
#define BUFFER_MERET 256
```

Beállítjuk a kódoláshoz megadott kulcs és a buffer maximális méretét.

```
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 main-be létrehozzunk két char típusú tömböt, amibe a kulcs és a buffer értékeit tároljuk. Majd inicializálunk egy kulcs_index változót, amibe majd a kulcs aktuális elemét tároljuk, karakterrel-karakterre léptetve a kódolás közben, míg az olvasott_bajtok változóba, a beolvasott bajtok számát fogjuk tárolni. Az strlen() függvénytel a kulcs méretét kapjuk meg. Azután az strcpy()-val bemásoljuk az előbb megadott parancssori argumentumban tárolt sztringet a kulcs tömbünkbe. A függvény 3. paramétere megadjanekünk, hogy max.mennyi karaktert másolhatunk át.

```
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;
    }
}
```

Addig olvasunk a bemenetről, amíg van mit és tároljuk a beolvasott bajtokat a bufferben. Ha már nincs mit olvasni, a read függvény 0-val tér vissza. minden egyes beolvasott bajtot össze EXOR-ozunk a kulcs soron következő karakterével. Végül pedig kiíratjuk az eredményt.

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

Megoldás forrása:https://progpater.blog.hu/2011/02/15/felvetelt_hirdet_a_cia

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

A titkosításhoz használt algoritmust egy külön osztályban hoztuk létre.

```
public class exor {

    public exor(String keyString,
                java.io.InputStream inputStream,
```

```
java.io.OutputStream outputStream)
throws java.io.IOException {

    byte [] kulcs = keyString.getBytes();
    byte [] buffer = new byte[256];
    int keyIndex = 0;
    int readBytes = 0;

    while((readBytes =
        inputStream.read(buffer)) != -1) {

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

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

        }

        outputStream.write(buffer, 0, readBytes);

    }

}

public static void main(String[] args) {

    try {

        new exor(args[0], System.in, System.out);

    } catch(java.io.IOException e) {

        e.printStackTrace();

    }

}

}
```

A paraméterként átadott kulcsszöveget tároljuk egy stringben, majd létrehozunk egy bejövő-kimenő csatornát. Beolvassuk a kulcsot egy byte típusú tömbbe. Inicializáljuk a kulcsindex és olvasott bájtok változókat. Majd egy while ciklus a bemenetet olvassa tömbönként buffer méret szerint. Ezután egy for ciklus segítségével bejárjuk az olvasott_bajtok változót és exorozzuk a buffer tartalmát a kulccsal. Majd kulcs_index-t növeljük a kulcs méret eléréséig. A main()-be használjuk a trycatch-et, ami a hibák elkapására használatos. A try tartalmazza az utasítást, ha valami nem stimmel hibát dob, a catch "elkapja" és kapunk egy hibaüzenetet.

Megoldás forrása:<https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html>

4.4. C EXOR törő

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

```
#define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 8
#define _GNU_SOURCE

#include <stdio.h>
#include <unistd.h>
#include <string.h>

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;
}
```

Kiszámítjuk az átlagos szóhosszt, mégpedig a következőképpen: a szöveg karakterszámát elosztjuk a szövegen található szóközök számával

```
int
tiszta_lehet (const char *titkos, int titkos_meret)
{

    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");

}
```

Az alap koncepció, hogy az eredeti szöveg biztosan tartalmazza a leggyakoribb magyar szavakat (hogy, nem, az, ha). Az átlagos szóhossz azért kell, hogy a törések számát csökkenthessük.

```
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);  
}
```

Bájtonként végrehajtjuk az EXOR-t. A % segítségével a kulcs akkor is aktuális marad, ha a szöveg hosszabb mint a keresett kulcs.

```
int  
main (void)  
{  
  
    char kulcs[KULCS_MERET];  
    char titkos[MAX_TITKOS];  
    char *p = titkos;  
    int olvasott_bajtok;  
  
    while ((olvasott_bajtok =  
            read (0, (void *) p,  
                  (p - titkos + OLVASAS_BUFFER <  
                   MAX_TITKOS) ? OLVASAS_BUFFER : titkos + MAX_TITKOS - p)))  
        p += olvasott_bajtok;  
  
    // maradek hely nullazasa a titkos bufferben  
    for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)  
        titkos[p - titkos + i] = '\0';  
  
    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]\nTiszta szoveg: [%s]\n",
ii, ji, ki, li, mi, ni, oi, pi, titkos);

exor (kulcs, KULCS_MERET, titkos, p - titkos);
}

return 0;
}
```

A while ciklus addig fog tartani, amíg van mit olvasni a bemenetről. Egymásba ágyazott for ciklusokkal előállítjuk az összes lehetséges kulcsot és kipróbáljuk minden. A végén újra exor-ozunk, így nem lesz szükségünk egy második bufferre.

4.5. Neurális OR, AND és EXOR kapu

R

Mi is az a neurális hálózat?

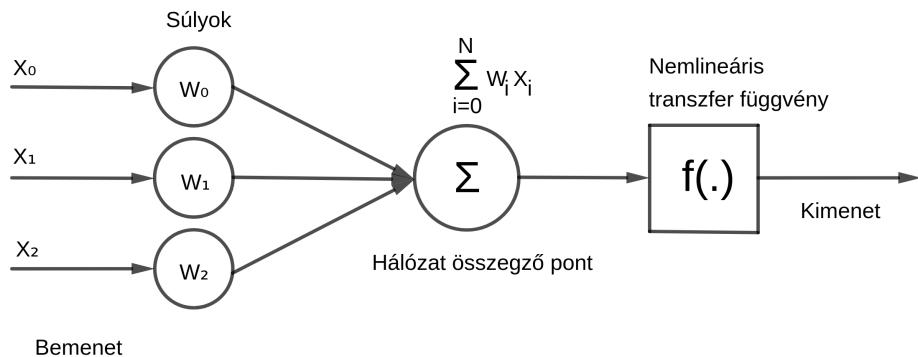
Egy olyan eszköz (legyen hardveres, vagy szoftveres), amely képes párhuzamosan feldolgozni a beérkező információt (esetünkben inputot) , illetve:

- neuronok (feldolgozást végző elemek) összekapcsolt rendszeréből áll
- minta alaján való tanulásra képes, tanulási algoritmussal rendelkezik
- előhívási algoritmus segítségével képes a korábban megtanult információ felhasználására/előhívására

Neurális hálózatok működésének két fő fázisa:

- A tanulási folyamat kezdete, a hálózat kialakítása. Egy lassabb folyamat, mely potenciálisan magában hordozhatja a sikertelen tanulási szakaszokat is.
- Az előhívási fázis, az előzővel ellentétben már jóval gyorsabb folyamat. Az előhívási algoritmus meghívása.

Legfontosabb építőeleme a **neuron** (feldolgozó elem), mely több bemenettel és egy kimenettel rendelkező eszköz, ami a bemenet és kimenet között egy nemlineáris leképezést hoz létre a megfelelő (transzfer) függvény segítségével.



4.1. ábra. Lokális memória nélküli neuron (perceptron)

Neuronok 3 fő típusa:

- **Bemeneti neuron:** Egyetlen bemenetük (a hálózat bemenete) és egyetlen kimenetük (a hálózak kimenete) van. Nincs jelfeldolgozó szerepük.
- **Kimeneti neuron:** A hálózatból a környezetbe továbbítja az információt.
- **Rejtett neuron:** bemenetük és kimenetük is a többi neuronhoz kapcsolódik.

Egy neuron akkor fog aktiválódni, ha a bemenetek súlyozott összege meghalad egy bizonyos értéket. A megfelelő bemenettel és eltolássúllyal rendelkező neuronok képesek lesznek logikai kapuként funkcionálni. Ezekkel a kapukkal képesek vagyunk az alapvető logikai függvényeket megjeleníteni.

OR logikai kapu (R implementáció):

```
library(neuralnet)
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
```

```

OR      <- c(0,1,1,1)

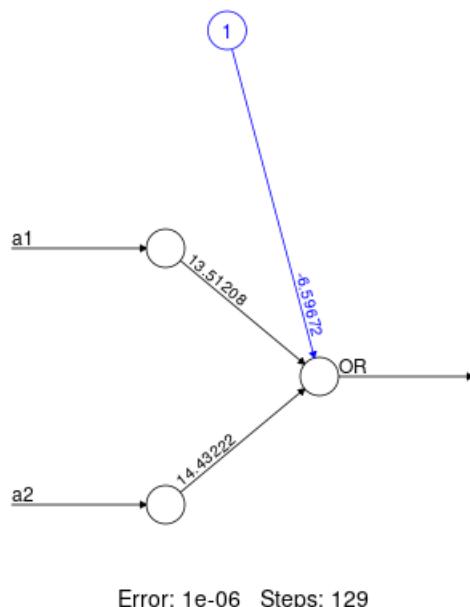
or.data <- data.frame(a1, a2, OR)

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

plot(nn.or)

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

```



4.2. ábra. OR logikai kapu

Eredmény:

```

$net.result
 [,1]
[1,] 0.001362976
[2,] 0.999008555
[3,] 0.999604714
[4,] 0.999999999

```

OR-AND logikai kapu (R implementáció):

```

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

```

```

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

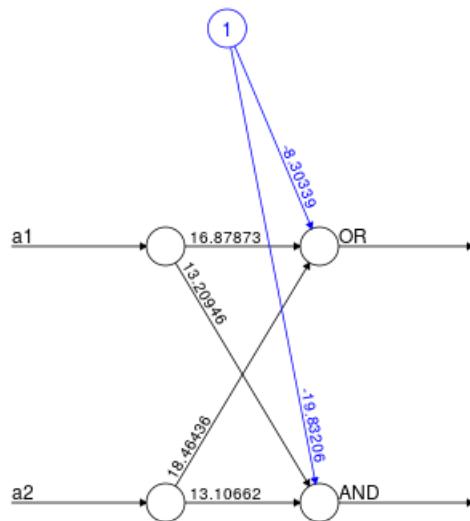
orand.data <- data.frame(a1, a2, OR, AND)

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])

```



Error: 3e-06 Steps: 191

4.3. ábra. OR-AND logikai kapu

Eredmény:

```

$net.result
      [,1]      [,2]
[1,] 0.0002476137 2.438077e-09
[2,] 0.9998113326 1.328202e-03
[3,] 0.9999613515 1.198563e-03
[4,] 1.0000000000 9.984747e-01

```

EXOR logikai kapu rejtett neuron nélkül (R implementáció):

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

```

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

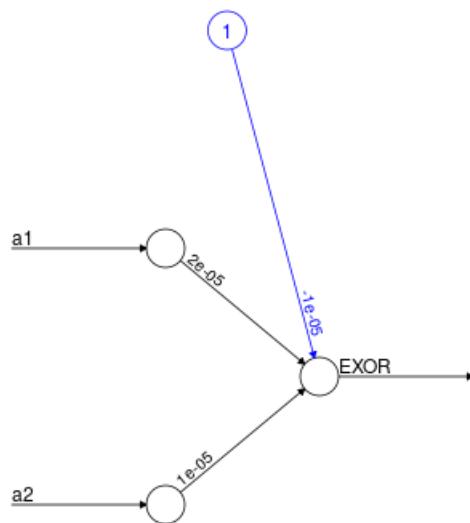
exor.data <- data.frame(a1, a2, EXOR)

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

plot(nn.exor)

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

```



Error: 0.5 Steps: 82

4.4. ábra. EXOR logikai kapu rejtett neuron nélkül

Eredmény:

```

$net.result
      [,1]
[1,] 0.4999972
[2,] 0.5000010
[3,] 0.4999987
[4,] 0.5000025

```

EXOR logikai kapu rejtett neuronokkal (R implementáció):

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

```

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

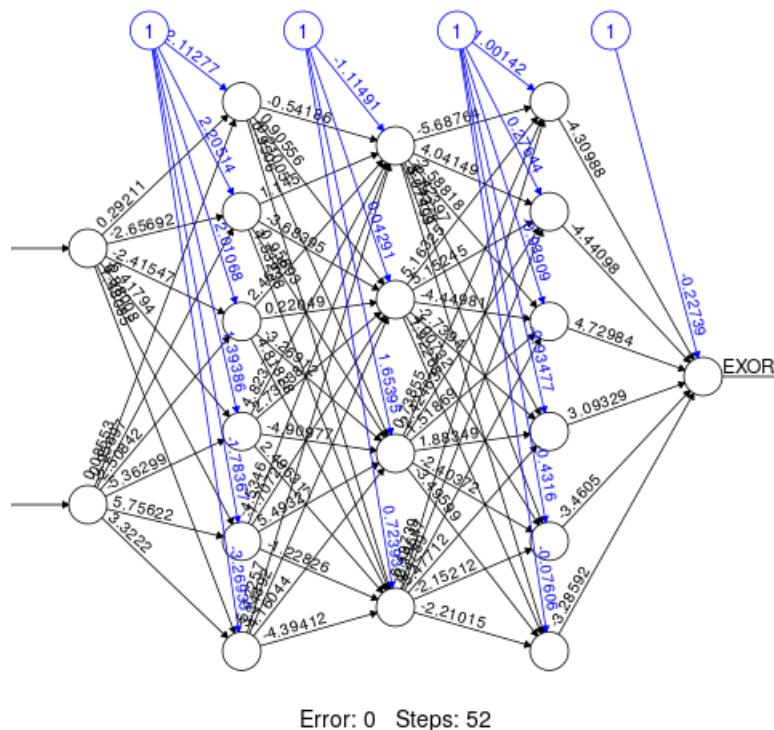
exor.data <- data.frame(a1, a2, EXOR)

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])

```



4.5. ábra. EXOR logikai kapu rejtett neuronokkal

Eredmény:

```

$net.result
      [,1]
[1,] 0.0001283067
[2,] 0.9993446185
[3,] 0.9993605209
[4,] 0.0001806955

```

Forrás 1: [bhax/attention_raising/NN_R/nn.r](https://github.com/bhax/attention_raising>NN_R/nn.r)

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

Megoldás forrása 1: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/NN_R

Megoldás forrása 2: https://www.tankonyvtar.hu/hu/tartalom/tamop425/0026_neuralis_4_4/ch01s04.html#id4925

4.6. Hiba-visszaterjesztéses perceptron

C++

Egy többrétegű perception tanítása:

- Definiáljuk a kezdeti súlyokat.
- Az input végighalad a hálózaton, a súlyok változatlanok maradnak.
- Az ezáltal kapott kimeneti jelet összehasonlítjuk a tényleges kimeneti jellel.
- A hibát visszaküldjük a hálózat neuronjain keresztül és változtatunk a súlyokon, oly módon, hogy a hibák száma minimálisra csökkenjen

A kérdés viszont az, hogy milyen módon változtassuk meg a súlyokat a lehető legkevesebb hiba elérésének érdekében?

Ez a hiba-visszaterjesztés (back-propagation) algoritmussal történik a kimeneti réteg(ek)ből a rejtett rétegekbe.

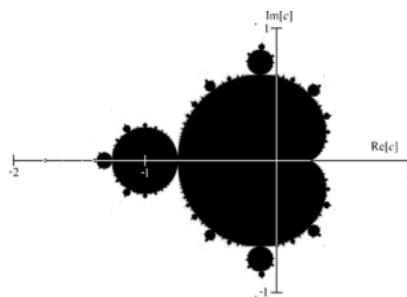
- A kimeneti rétegre meghatározzuk a hibák számát.
- Az így kapott hibaértékeket visszaterjesztjük a kimeneti réteg előtti, rejtett rétegre.
- A kapott hibákat egyre korábbi rétegekre terjesztjük oly módon, hogy a hiba skálázódik, az aktuális és az őt megelőző súlyok értékeinek függvényében.
- Az algoritmust addig folytatjuk, amíg el nem érjük a bemeneti réteget.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Egy "örökké létező önmagát újra és újra reprodukáló, kaotikus, inflálódó világgegyetemet ábrázol." A mintha megfigyelése csak a legújabb technológiai vívmányok segítségével vált lehetséges. Szabad szemmel ugyanis csak villázó fényeket láthatnánk. Pedig ezek valójában végtelenszer ismétlődő Mandelbrot halmazok.



5.1. ábra. Mandelbrot kétdimenziós koordinátarendszerben való ábrázolása

A Mandelbrot halmaz, egy, a komplex számsíkon vizuálisan ábrázolható fraktál-minta. Ahhoz, hogy értelmezhessük ezt a mondatot, haladjuk szépen sorban. Mi is az a komplex szám? Valószínűleg mindenki meg tudja mondani, hogy mennyi 4-nek a négyzetgyöke(2), vagy 16-nak (4). De tudjuk-e, hogy mennyi -1-nek? A válasz pedig i , egy kitalált szám, melyet négyzetre emelve -1-et kapunk. Ha összekapcsolunk egy valós számot, pl.: 3-at a z -vel, akkor az így kapott $3i$ egy komplex szám lesz. A 3 a valós rész, az i pedig a képzetes (imaginárius) rész.

Ahhoz, hogy a komplex számokat vizualizálni tudjuk, szükség van egy Descartes-féle derékszögű koordináta rendszerhez hasonló 2 dimenziós térré. Ez lesz a komplex számsík. A vízszintes tengely lesz a valós tengely, a függőleges tengely pedig a képzetes tengely. Ezen a számsíkon képesek vagyunk ábrázolni minden komplex számot. Az így ábrázolt elemek talán legfontosabb eleme az origótól való távolsága. Ez által sokkal egyszerűbben vizualizálhatjuk nem csak magukat a számokat, de a velük elvégezhető műveleteket is (összeadás, kivonás). De hogy jutunk el innen, a Mandelbrot halmazig?

```
#include <png++/png.hpp>
```

```
#define N 800
#define M 800
#define MAXX 0.7
#define MINX -2.0
#define MAXY 1.35
#define MINY -1.35
```

A `png++` csomag segítségével fogjuk kirajzolatni egy fájlba az eredményt. Definiáljuk az elkészíteni kívánt kép méreteit, illetve a koordinátarendszeren a tartományt, amin belül szeretnénk, hogy a program dolgozzon.

```
void GeneratePNG( int tomb[N][M] )
{
    png::image< png::rgb_pixel > image(N, M);
    for (int x = 0; x < N; x++)
    {
        for (int y = 0; y < M; y++)
        {
            image[x][y] = png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y] ←
                ]);
        }
    }
    image.write("mandelbrot1.png");
}
```

Az ez után létrehozott függvény fogja nekünk kiírni a halmaz képét soronként és oszloponként haladva egy `png` fájlba.

```
struct Komplex
{
    double re, im;
};
```

A komplex számok ábrázolásához létrehozott struktúra valós és imaginárius értékekkel.

```
int main()
{
    int tomb[N][M];

    int i, j, k;

    double dx = (MAXX - MINX) / N;
    double dy = (MAXY - MINY) / M;

    struct Komplex C, Z, Zuj;

    int iteracio;

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

```

for (j = 0; j < N; j++)
{
    C.re = MINX + j * dx;
    C.im = MAXY - i * dy;

    Z.re = 0;
    Z.im = 0;
    iteracio = 0;

    while(Z.re * Z.re + Z.im * Z.im < 4 && iteracio++ < 255)
    {
        Zuj.re = Z.re * Z.re - Z.im * Z.im + C.re;
        Zuj.im = 2 * Z.re * Z.im + C.im;
        Z.re = Zuj.re;
        Z.im = Zuj.im;
    }

    tomb[i][j] = 256 - iteracio;
}
}

GeneratePNG(tomb);

return 0;
}

```

Számolni kezdjük a $f_c(z) = z^2 + c$ iterációit. Végiglépkedünk a rácspontról. C.re (valós rész) és C.im (imaginárius rész) a háló rácspontrajának megfelelő komplex számot alkotják. Z.re és Z.im iterációs komplex szám részei. Amennyiben az iterációk során a c távolsága a Z_0 ponttól (azaz az origótól) nagyobb lesz mint 2, vagy elérjük az iterációs maximumot, akkor az érték nem lesz eleme a mandelbrot halmaznak.

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#mandelbrot_halmaz

Így fordítsd és futtasd:

```

$ g++ -c mandelbrot.cpp `libpng-config --cflags`
$ g++ -o mandelbrot mandelbrot.o `libpng-config --ldflags`
$ ./mandelbrot           <-- létrehozzuk a képet
$ rm -rf *.o              <-- eltakarítjuk az ideiglenes fájlokat
$ rm -rf mandelbrot       <-- eltakarítjuk az ideiglenes fájlokat

```

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

Az `complex` osztály beemelésével lehetőségünk lesz olyan komplex típusokat deklarálni, amelyek egy változónak a valós és imaginárius részét is el tudják tárolni. Így az algoritmus komplex számait egy egységeként lehet kezelni.

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

```
#include <complex>

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

    int szelesseg = 1366;
    int magassag = 1024;
    int iteraciosHatar = 255;
    double a = -2;
    double b = 0.7;
    double c = -1.35;
    double d = 1.35;
```

Definiáljuk az elkészíteni kívánt kép méreteit, illetve a koordinátarendszeren a tartományt, amin belül szereznénk, hogy a program dolgozzon.

```
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;
}
```

A program futtatásánál megadható argumentumokat vizsgáljuk. minden egyes argumentumot beolvasás-kor egy egésszé vagy egy lebegőpontos számmá alakítunk át, majd elhelyezzük őket egy tömbben. Így állíthatjuk a képméretet és a valós-számrendszerbeli tartományt.

```
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";

// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    // k megy az oszlopokon
```

```
for ( int k = 0; k < szelesseg; ++k )
{
    // c = (reC, imC) a halo racspontjainak
    // megfelelo komplex szam

    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 ) );
}

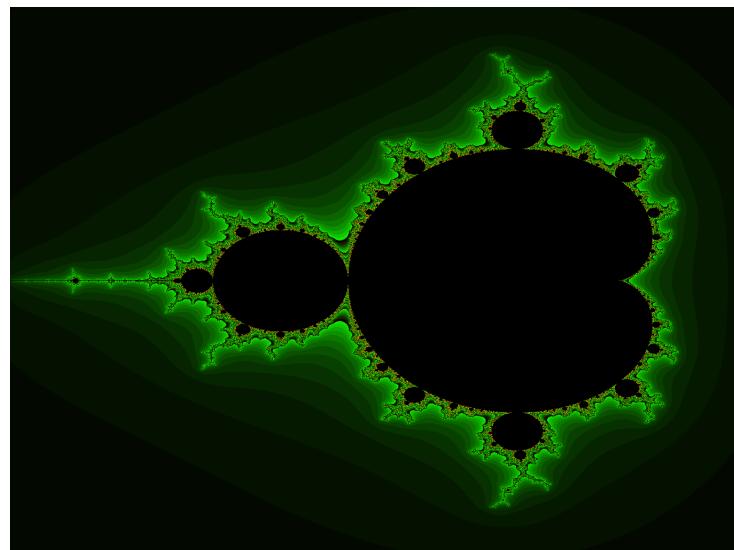
int szazalek = ( double ) j / ( double ) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}

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

A kép, mint 2 dimenziós objektum pixeleit egyesével kezdjük el vizsgálni, hogy része lesz-e a Mandelbrot halmazunknak. A while ciklus belseje az előző Mandelbrot halmaz pontban vázolt algoritmushoz képest nagyon leegyszerűsödik, hála a std::complex osztálynak.

Így fordítsd és futtasd:

```
$ g++ 3.1.2.cpp -lpng -O3 -o 3.1.2
$ ./3.1.2 mandelbrot2.png 1366 1024 2040 -2 0.7 -1.35 1.35
Szamitas
mandelbrot2.png mentve.
```



5.2. ábra. Mandelbrot 3.1.2 kimenete

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

5.3. Biomorfok

Mik is a biomorfok? A biomorfok olyan alakzatok, melyek nagyon hasonlítanak egy természetes organizmus mikroszkópikus képére, viszont matematikai és nem biológiai eredetűek. Mi esetünkben egy kétdimenziós számsíkon ábrázolt fraktólokról van szó. Legismertebb fraktálok: "Julia-halmazok", "Mandelbrot-halmaz" és a véletlenül felfedezett "Biomorfok".

A különbség a Mandelbrot halmaz és a Julia halmazok között az, hogy a Mandelbrot-halmaz komplex iterációban a C változó:

```
// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    for ( int k = 0; k < szelesseg; ++k )

        // c = (reC, imC) a halo racspontjainak
        // megfelelo komplex szam

        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;  
}
```

Míg a Julia-halmaz ábrázolásánál a c konstans lesz. minden vizsgált z rácpontra ugyanaz:

```
// j megy a sorokon  
for ( int j = 0; j < magassag; ++j )  
{  
    // k megy az oszlopokon  
    for ( int k = 0; k < szelesseg; ++k )  
    {  
        double rez = a + k * dx;  
        double imZ = d - j * 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;  
            }  
        }  
    }  
}
```

A korábbi Mandelbrot halmazt kiszámoló forráskódot módosítva:

```
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: ./3.1.2 fajlnev szelessseg magassag n a b c ←
        d reC imC R" << std::endl;
    return -1;
}

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

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

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

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

// j megy a sorokon
for ( int y = 0; y < magassag; ++y )
{
    // k megy az oszlopokon

    for ( int x = 0; x < szelessseg; ++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;
            //z_n = std::pow(z_n, 2) + std::sin(z_n) + cc;
            if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
            {
                iteracio = i;
                break;
            }
        }
}
```

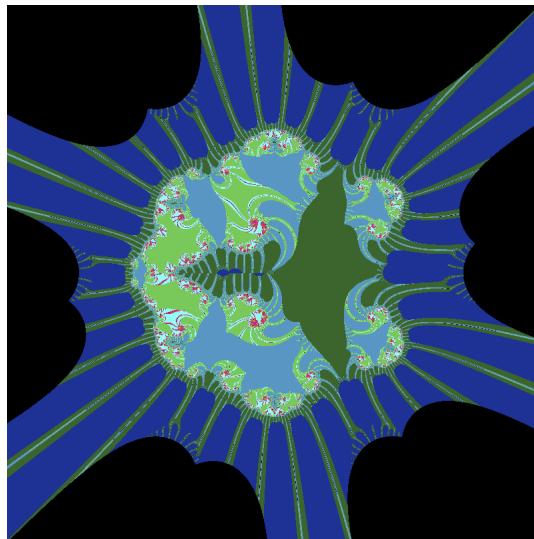
```
    kep.set_pixel ( x, y,
                    png::rgb_pixel ( (iteracio*20)%255, (iteracio ←
                        *40)%255, (iteracio*60)%255 ) );
}

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

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

Így fordítsd és futtasd:

```
$ g++ 3.1.3.cpp -lpng -O3 -o 3.1.3
$ ./3.1.3 bmorf.png 800 800 10 -2 2 -2 2 .285 0 10
Szamitas
bmorf.png mentve.
```



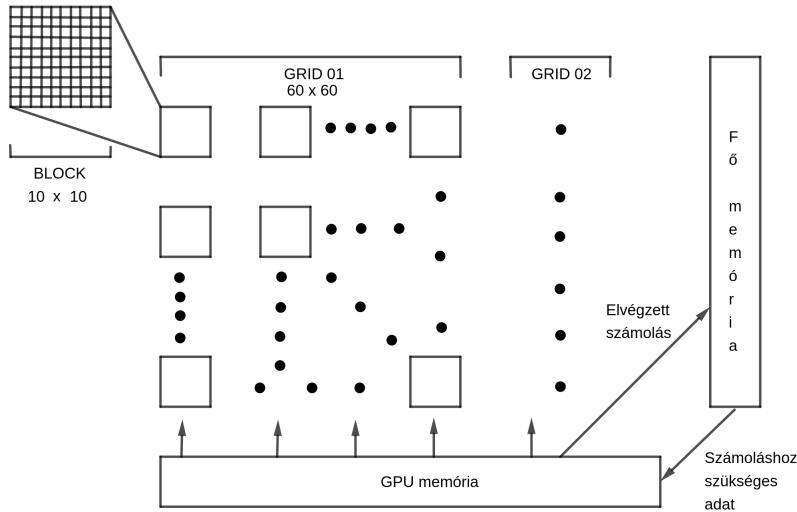
5.3. ábra. Biomorf ábrázolása:

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

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

5.4. A Mandelbrot halmaz CUDA megvalósítása

A CUDA az NVidia kártyák programozói interfésze, amit párhuzamos számításokhoz - amiben a videokártya jó - lehet felhasználni. Ezzel sok terhet levesz a CPU válláról.



5.4. ábra. CUDA-s Mandelbrot halmaz váza, 600 x 600-as kép esetén:

```
#include <png++/image.hpp>
#include <png++/rgb_pixel.hpp>

#include <sys/times.h>
#include <iostream>

#define MERET 600
#define ITER_HAT 32000

__device__ int
mandel (int k, int j)
{
    // Végigzongorázza a CUDA a szélesség x magasság rácsot:
    // most eppen a j. sor k. oszlopban vagyunk

    // 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;
// z_{n+1} = z_n * z_n + c iterációk
// számítása, amíg |z_n| < 2 vagy még
// nem értük el a 255 iterációt, ha
// viszont elértek, akkor úgy vesszük,
// hogy a kiinduláci c komplex számra
// az iteráció konvergens, azaz a c a
// Mandelbrot halmaz eleme
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;
}

/*
__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);

}

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

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

    // dim3 grid (MERET, MERET);
    // mandelkernel <<< grid, 1 >>> (device_kepadat);

    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);

}

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;

}
```



The screenshot shows a terminal window with the following session:

```
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$ g++ mandelpngt.c++ -lpng16 -O3 -o mandelT
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$ ./mandelT mandelt.png
956
9.56184 sec
mandelt.png mentve
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$ nvcc mandelpngc_60x60_100.cu -lpng16 -O3 -o mandelC
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$ ./mandelC mandelc.png
mandelc.png mentve
3
0.03929 sec
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$
```

5.5. ábra. CUDA-s kép kiszámítása ennyivel gyorsabb

Megoldás videó: <https://www.youtube.com/watch?v=gvaqijHIRUs>

Megoldás forrása:https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux-adatok.html

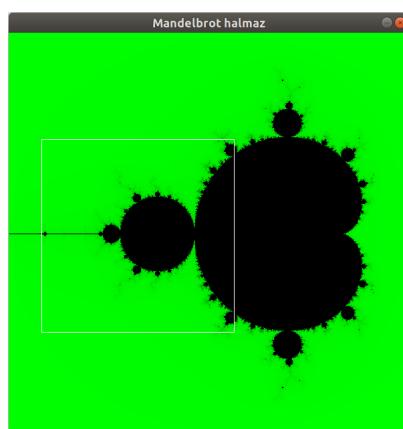
5.5. Mandelbrot nagyító és utazó C++ nyelven

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!

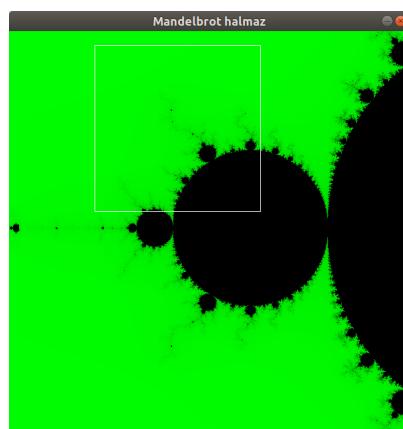
A források adottak, már csak fordítani, futtatni kell őket. Első lépésként telepítsünk fel egy könyvtárat a sudo apt-get install libqt4-dev paranccsal. Rakjunk minden forrást egy mappába, majd a qmake -project parancsot használva létrehozunk egy valami.pro fájlt. Ezt a fájlt nyissuk meg terminálban, majd gépeljük be ezt a karakter sorozatot: QT += widgets

Ezután a qmake valami.pro parancssal létrehozzunk egy makefile-t, majd a make parancsot kiadva létrehozunk további fájlokat.

Futtatás után a következőt kapjuk:



5.6. ábra. Mandelbrot C nyelven:



5.7. ábra. Első nagyítás után:

A nagyítást folytathatjuk kedvünk szerint. "n" gomb lenyomásával újra számolhatunk pontosabb értékek kiszámítása érdekében

Szükséges források:

[Programozó Páternoszter](#)

[UDPROG közösségi](#)

5.6. Mandelbrot nagyító és utazó Java nyelven

A projektet a [Javát tanítók](#) oldalán található útmutató alapján készítettem el.

```
public class MandelbrotHalmaz extends java.awt.Frame implements Runnable {  
  
    protected double a, b, c, d;  
  
    protected int szélesség, magasság;  
  
    protected java.awt.image.BufferedImage kép;  
  
    protected int iterációsHatár = 255;  
  
    protected boolean számításFut = false;  
  
    protected int sor = 0;  
  
    protected static int pillanatfelvételSzámláló = 0;  
  
    public MandelbrotHalmaz(double a, double b, double c, double d,  
                           int szélesség, int iterációsHatár) {  
        this.a = a;  
        this.b = b;  
        this.c = c;  
        this.d = d;  
        this.szélesség = szélesség;  
        this.iterációsHatár = iterációsHatár;  
  
        this.magasság = (int)(szélesség * ((d-c) / (b-a)));  
  
        kép = new java.awt.image.BufferedImage(szélesség, magasság,  
                                              java.awt.image.BufferedImage.TYPE_INT_RGB);  
  
        addWindowListener(new java.awt.event.WindowAdapter() {  
            public void windowClosing(java.awt.event.WindowEvent e) {  
                setVisible(false);  
                System.exit(0);  
            }  
        });  
    }  
}
```

Deklaráljuk a komplex sík tartományát, illetve a síkra vetített hálót és adatait.

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

```
public void keyPressed(java.awt.event.KeyEvent e) {
    if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
        pillanatfelvétel();

    else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {
        if(számításFut == false) {
            MandelbrotHalmaz.this.iterációsHatár += 256;

            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;

            számításFut = true;
            new Thread(MandelbrotHalmaz.this).start();
        }
    }
}
});
```

Figyeljük a billentyűzetről érkező inputokat.

- S lenyomására pillanatkép készül.
- N lenyomására újraszámoljuk és kicsivel pontosabb számolást végzünk.
- M lenyomására újraszámoljuk és sokkal pontosabb számolást végzünk.

```
setTitle("A Mandelbrot halmaz");
setResizable(false);
setSize(szélesség, magasság);
setVisible(true);

számításFut = true;
new Thread(this).start();
}

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

    g.drawImage(kép, 0, 0, this);

    if(számításFut) {
        g.setColor(java.awt.Color.RED);
        g.drawLine(0, sor, getWidth(), sor);
    }
}
```

A megjeleníteni kívánt ablak beállításai. A számítást futás közben egy vörös szakasszal jelöljük.

```
public void pillanatfelvétel() {  
  
    java.awt.image.BufferedImage mentKép =  
        new java.awt.image.BufferedImage(szélesség, magasság,  
            java.awt.image.BufferedImage.TYPE_INT_RGB);  
    java.awt.Graphics g = mentKép.getGraphics();  
    g.drawImage(kép, 0, 0, this);  
    g.setColor(java.awt.Color.BLUE);  
    g.drawString("a=" + a, 10, 15);  
    g.drawString("b=" + b, 10, 30);  
    g.drawString("c=" + c, 10, 45);  
    g.drawString("d=" + d, 10, 60);  
    g.drawString("n=" + iterációsHatár, 10, 75);  
    g.dispose();  
  
    StringBuffer sb = new StringBuffer();  
    sb = sb.delete(0, sb.length());  
    sb.append("MandelbrotHalmaz_");  
    sb.append(++pillanatfelvételszámláló);  
    sb.append("_");  
  
    sb.append(a);  
    sb.append("_");  
    sb.append(b);  
    sb.append("_");  
    sb.append(c);  
    sb.append("_");  
    sb.append(d);  
    sb.append(".png");  
  
    try {  
        javax.imageio.ImageIO.write(mentKép, "png",  
            new java.io.File(sb.toString()));  
    } catch(java.io.IOException e) {  
        e.printStackTrace();  
    }  
}
```

A pillanatkép elmentése, a kép adatainak beállítása, illetve az aktuális adatok képre mentése.

```
public void run() {  
  
    double dx = (b-a)/szélesség;  
    double dy = (d-c)/magasság;  
    double reC, imC, reZ, imZ, ujreZ, ujimZ;  
    int rgb;  
  
    int iteráció = 0;
```

```
for(int j=0; j<magasság; ++j) {
    sor = j;
    for(int k=0; k<szélesség; ++k) {

        reC = a+k*dx;
        imC = d-j*dy;

        reZ = 0;
        imZ = 0;
        iteráció = 0;

        while(reZ*reZ + imZ*imZ < 4 && iteráció < iterációsHatár) {

            ujreZ = reZ*reZ - imZ*imZ + reC;
            ujimZ = 2*reZ*imZ + imC;
            reZ = ujreZ;
            imZ = ujimZ;

            ++iteráció;

        }

        iteráció %= 256;

        rgb = (255-iteráció) |
            ((255-iteráció) << 8) |
            ((255-iteráció) << 16);

        kép.setRGB(k, j, rgb);
    }
    repaint();
}
számításFut = false;
}

public static void main(String[] args) {
    new MandelbrotHalmaz(-2.0, .7, -1.35, 1.35, 600, 255);
}
```

A program lelke, ami a halmaz számítását végzi. A Dupla for ciklussal végigmegyünk a szélesség x magasság dimenzióin. A $c = (reC, imC)$ a rácspontoknak megfelelő komplex szám. $z_{n+1} = z_n * z_n + c$ -t iteráljuk amíg el nem érjük a 2-t, vagy az iterációs határt. Ha elérjük a határt, akkor a vizsgált pont a halmaz része, tehát az iteráció konvergens.

A nagyító programot ennek a programnak a továbbbbfejlesztésével hozhatjuk létre. Ehhez létrehozunk egy újabb osztályt.

```
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {
```

```
private int x, y;  
  
private int mx, my;
```

Itt megjegyezzük a nagyítani kívánt terület bal felső sarkát, valamint a terület szélességét és magasságát.

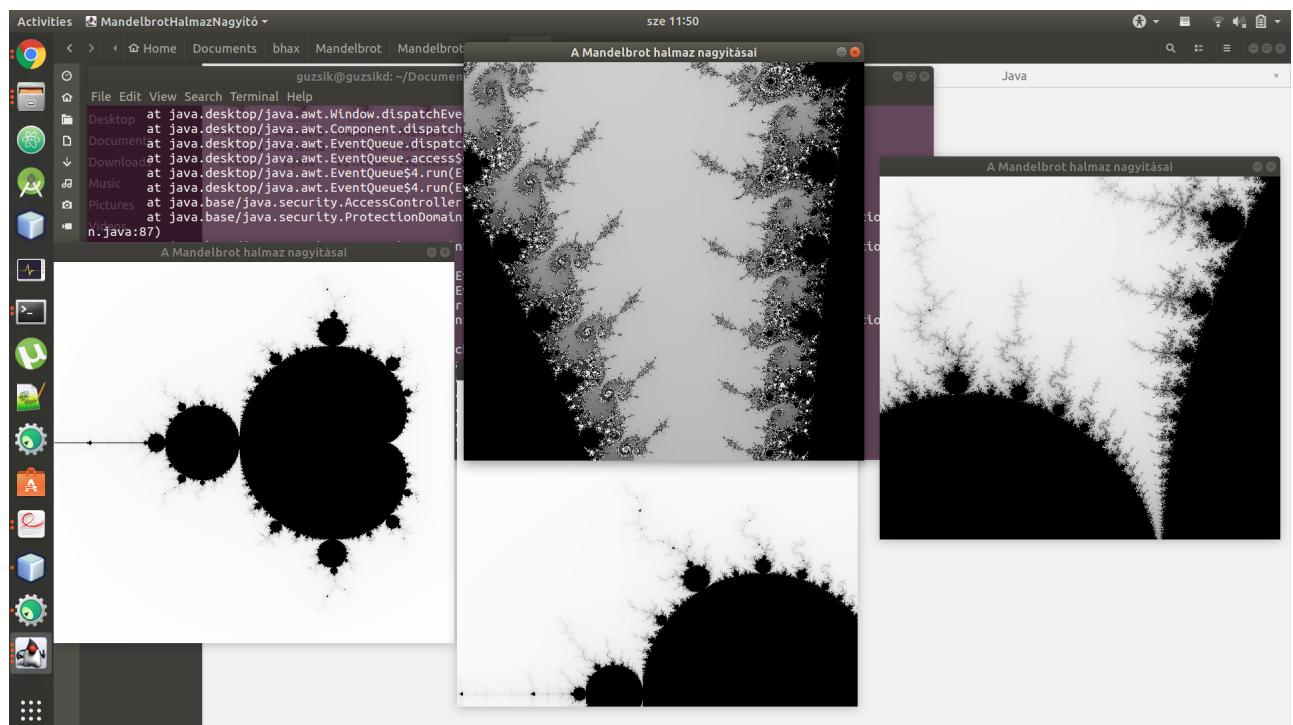
```
public MandelbrotHalmazNagyító(double a, double b, double c, double d,  
        int szélesség, int iterációsHatár) {  
  
    super(a, b, c, d, szélesség, iterációsHatár);  
    setTitle("A Mandelbrot halmaz nagyításai");  
  
    addMouseListener(new java.awt.event.MouseAdapter() {  
  
        public void mousePressed(java.awt.event.MouseEvent m) {  
  
            x = m.getX();  
            y = m.getY();  
            mx = 0;  
            my = 0;  
            repaint();  
        }  
  
        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);  
        }  
    });  
  
    addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {  
  
        public void mouseDragged(java.awt.event.MouseEvent m) {  
  
            mx = m.getX() - x;  
            my = m.getY() - y;  
            repaint();  
        }  
    });  
}
```

```
}
```

Egérkattintással és nyomvatartással kijelöljük a nagyítani kívánt területet. Ha felengedjük az egeret, akkor az újonnak kijelölt terület ismét kiszámításra kerül.

```
public void paint(java.awt.Graphics g) {  
  
    g.drawImage(kép, 0, 0, this);  
  
    if(számításFut) {  
        g.setColor(java.awt.Color.RED);  
        g.drawLine(0, sor, getWidth(), sor);  
    }  
  
    g.setColor(java.awt.Color.GREEN);  
    g.drawRect(x, y, mx, my);  
}  
  
public static void main(String[] args) {  
  
    new MandelbrotHalmazNagyító(-2.0, .7, -1.35, 1.35, 600, 255);  
}  
}
```

A nagyítani kívánt területet újrarajzoltatjuk. ha éppen számol a program, akkor egy vörös szakasszal jelöljük, hogy hol tart éppen.



5.8. ábra. Mandelbrot nagyítása JAVA környezetben

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 eltéve kiszámolt szám.

Az algoritmus készítéséhez segítségül szolgált a [Javát tanítok](#) honlapján található útmutatás és az [UDPROG közössége](#) repójá.

Polártranszformációs algoritmus C++:

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>

using namespace std;

class Random
{

public:

    Random();
    ~Random() {}

    double get();

private:

    bool exist;
    double value;

};
```

Létrehozzuk a Random osztályt, a publikus részben az osztály konstruktora, destruktora és a random számokat lekérő függvény fog helyet kapni, míg a privát részben azt fogjuk tárolni, hogy létezik-e korábban kiszámolt másik random, illetve ennek mi az értéke.

```
Random::Random()
{
    exist = false;
    srand (time(NULL));
}
```

A konstruktorban inicializáljuk a randomszám generáló srand függvényt.

```
double Random::get()
{
    if (!exist)
    {
        double u1, u2, v1, v2, w;

        do
        {
            u1 = rand () / (RAND_MAX + 1.0);
            u2 = rand () / (RAND_MAX + 1.0);
            v1 = 2 * u1 - 1;
            v2 = 2 * u2 - 1;
            w = v1 * v1 + v2 * v2;
        }
        while (w > 1);

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

        value = r * v2;
        exist = !exist;

        return r * v1;
    }

    else
    {
        exist = !exist;
        return value;
    }
}
```

A lekérő függvény. Amennyiben nincs eltárolva random számunk, akkor létrehozunk két normálist. Az elsőt ki fogjuk íratni, a másikat eltároljuk. Ha van eltárolva szám, akkor csak annyit csinálunk, hogy azt kiíratjuk.

```
int main()
{
    Random rnd;
```

```
for (int i = 0; i < 2; ++i) cout << rnd.get() << endl;  
}
```

A main() függvényben már nem sok tennivalónk van, csak meghívjuk a random osztály lehívó függvényét.

```
$ g++ random_class.cpp -o random  
$ ./random  
2.46919  
0.688235
```

Polártranszformációs algoritmus JAVA:

```
public class PolárTranszF {  
  
    boolean létezik_tárolt = false;  
    double tárolt;  
  
    public PolárTranszF() {  
  
        létezik_tárolt = false;  
  
    }  
}
```

A publikus PolárTranszF osztályunkban deklaráljuk a logikai tagot, ami azt figyeli, hogy van-e korábban eltárolt random, illetve ennek az értékét.

```
public double matek_rész() {  
  
    if(!létezik_tárolt) {  
  
        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;  
        létezik_tárolt = !létezik_tárolt;  
  
        return r*v1;  
    }  
}
```

```
    } else {
        létezik_tárolt = !létezik_tárolt;
        return tárolt;
    }
}
```

Ha nincs korábban eltárolt random értékünk, akkor meghívjuk a misztikus random-generáló matematikai eljárást. Amennyiben volt eltárolva, akkor cask az értékét adjuk vissza és a logikai tagot átállítjuk.

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

    PolárTranszF g = new PolárTranszF();

    for(int i=0; i<2; ++i)
        System.out.println(g.matek_rész());

}
```

Majd legvégül kétszer meghívjuk a misztikus matematikai eljárást. Első hívásnál ki fogja nekünk írni, a frissen kiszámolt számok egyikét, második hívásnál pedig az első alkalommal kiszámolt, majd elmentett értéket adja vissza.

```
$ javac PolárTranszF.java
$ java PolárTranszF
0.678582973383662
1.6936497301003002
```

Ha "fellapozzuk" a Java Development Kit `src.zip`-ben, azon belül a `/java/util/` elérési útvonalon található `Random.java` fájlt, akkor tapasztalhatjuk, hogy nem állunk túl messze a valóságtól. Ugyanis az 583. sortól megtalálhatjuk a JAVA saját randomszám generátorát a `nextGaussian` fantázianévre hallgató függvényt, ami szinte azonos a fentebb bemutatott C++-os és JAVA-s példákkal.

Megoldás forrása:<https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html/>

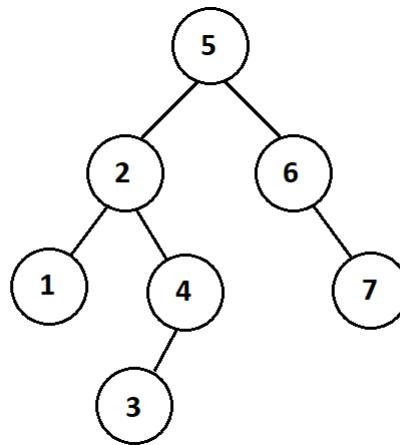
6.2. LZW

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

Az Lempel–Ziv–Welch (LZW) algoritmust az 1980-as évek közepén, Abraham Lempel és Jacob Ziv már létező algoritmusát továbbfejlesztve, Terry Welch publikálta. Az algoritmus a UNIX alapú rendszerek fájl-tömörítő segédprogramja által terjed el leginkább, továbbá GIF kiterjesztésű képek és PDF fájlok veszteségmentes tömörítéséhez is használják. Az USA-ban 2003-ban, a világ többi részén 2004-ben az algoritmus szabadalma lejárt, ezért azóta alkalmazása a háttérbe szorult.

A bináris fát úgy építjük föl, hogy a bemenetre érkező nullákat és egyeseket olvassuk. Ha olyan egységet olvasunk be, ami már korábban volt, akkor olvassuk tovább. Az így kapott új egységet fogjuk a fa gyökerétől

kezdve "lelépkedni". Ha az adott egység ábrázolva van, visszaugrunk a gyökérbe és olvassuk tovább az inputot.



6.1. ábra. Binfa felépítése:

A program készítéséhez [Programozó Páternoszter](#)-en található útmutatót hívtam segítségül.

A kód bemutatása lépésről lépésre:

```

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

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

Itt definiáljuk a binfa struktúrát.

```

BINFA_PTR
uj_elelem ()
{
    BINFA_PTR p;

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

Az új elem létrehozásakor memóriát szabadítunk majd fel, hiba esetén viszont kiléünk a programból.

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

Deklaráljuk, de még nem definiáljuk a fa kiírásához és a lefoglalt memória felszabadításához használt függvényeket. (Egyelőre nem foglalunk le nekik helyet a memóriában.)

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

    BINFA_PTR gyoker = uj_elem ();
    gyoker->ertek = '/';
    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);
    extern int max_melyseg;
    printf ("melyseg=%d\n", max_melyseg);
    szabadit (gyoker);
}
```

Először létrehozzuk a gyökeret és ráállítjuk a fa pointert. Olvassuk az inputon érkező 0-kat és 1-eseket.

- Ha 0-t olvasunk és az aktuális nodenak (amire a fa pointerünk jelenleg mutat) nincs nullás gyermeke, akkor az `uj_elem` függvénytel létrehozunk neki egyet, majd megkapja értékül a 0-t. Ezután ennek az újonnan létrehozott gyermeknek beállítjuk a bal és jobb gyermeket NULL pointerekre és visszaugrunk a pointerrel a gyökérre. Ha a beolvasáskor már volt nullás gyermek az aktuális nodenak, akkor a mutatót ráállítjuk.
- Ha 1-t olvasunk és az aktuális nodenak (amire a fa pointerünk jelenleg mutat) nincs egyes gyermek, akkor az `uj_elem` függvénytel létrehozunk neki egyet, majd megkapja értékül az 1-t. Ezután ennek az újonnan létrehozott gyermeknek beállítjuk a bal és jobb gyermeket NULL pointerekre és visszaugrunk a pointerrel a gyökérre. Ha a beolvasáskor már volt egyes gyermek az aktuális nodenak, akkor a mutatót ráállítjuk.

```
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);
        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);
        --melyseg;
    }
}
```

A `kiir` függvénytel jelenítjük meg magát a fát a parancssorban. Mivel 90 fokkal el van forgatva az eredmény balra és fentről lefelé írjuk ki az ágakat ezért inorder bejárás esetén először a jobb oldali ágat, majd a gyökeret, majd végül a bal oldali ágat rajzoltatjuk ki. Közben számoljuk a fa mélységét is.

```
void
szabadit (BINFA_PTR elem)
{
```

```
if (elem != NULL)
{
    szabadit (elem->jobb_egy);
    szabadit (elem->bal nulla);
    free (elem);
}
```

Rekurzívan hívjuk a függvényt, amivel felszabadítjuk a korábban lefoglalt elemeket.

A programot fordítva és futtatva tetszőleges inputtal:



```
guzsik@guzsikd: ~/Documents/bhax/Welch/Binfa
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/bhax/Welch/Binfa$ ./lzw <fa.txt
01111001001001000111

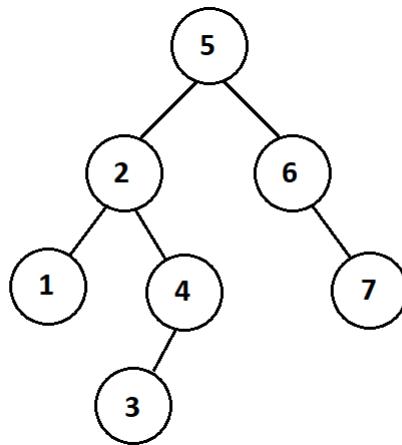
-----1(4)
----1(3)
---1(2)
--0(3)
---0(4)
----0(5)
-/1)
----1(3)
---0(2)
----0(3)
melyseg=5
guzsik@guzsikd:~/Documents/bhax/Welch/Binfa$
```

Megoldás forrása:https://progpater.blog.hu/2011/02/19/gyonyor_a_tomor/

6.3. Fabejárás

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

Az előző LZW bináris fa feladatban a fát Inorder módon jártuk be. Azaz a bal oldali részfát dolgozuk fel, majd a gyökeret és végül a jobb oldali részfát.



6.2. ábra. Emlékeztetőül az Inorder bejárásra:

A forráskódon belül, ezt a `kiír` függvénynél láthattuk.

```
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);
        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);
        --melyseg;
    }
}
```

A program a következő fát rajzolta ki nekünk:



```
guzsik@guzsikd: ~/Documents/bhax/Welch/Binfa
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/bhax/Welch/Binfa$ ./lzw <fa.txt
01111001001001000111

-----1(4)
----1(3)
---1(2)
--0(3)
---0(4)
----0(5)
-/1(1)
----1(3)
---0(2)
----0(3)
melyseg=5
guzsik@guzsikd:~/Documents/bhax/Welch/Binfa$
```

A preorder bejárás a következőképpen fog alakulni. Először minden a gyökeret dolgozzuk fel, majd a bal oldali részfát (ott is először a gyökeret) és végül a jobb oldali részfát (ott is a gyökeret).

A forráskódon a következő apró módosítást hajtjuk végre:

```
void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;

        kiir (elem->jobb_egy);
        kiir (elem->bal_nulla);
        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ↔
               , melyseg);

        --melyseg;
    }
}
```

A program a következő fát rajzolta ki nekünk:

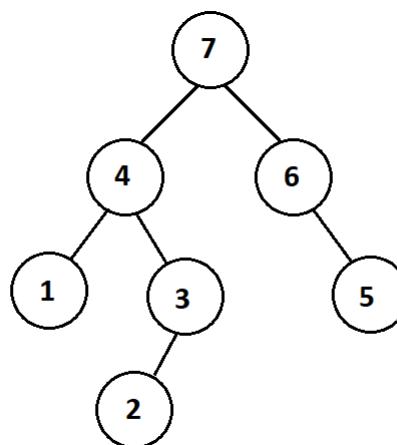
```

guzsik@guzsikd: ~/Documents/bhax/Welch/Binfapre
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/bhax/Welch/Binfapre$ ./lzwpre <fa.txt
01111001001001000111

-----1(4)
----1(3)
---0(5)
---0(4)
---0(3)
----1(2)
----1(3)
----0(3)
----0(2)
---/(1)
melyseg=5
guzsik@guzsikd:~/Documents/bhax/Welch/Binfapre$ 

```

A postorder bejárásnál először minden a bal oldali részfát (ott is először a bal részfát) majd a jobb oldali részfát (ott is a jobb részfát) dolgozzuk fel. A legvégre marad a gyökér.



6.3. ábra. Példa Postorder bejárásra:

A forráskódon a következő apró módosítást hajtjuk végre:

```

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->jobb_egy);
        kiir (elem->bal nulla);
        --melyseg;
    }
}

```

A program a következő fát rajzolta ki nekünk:

```

guzsik@guzsikd: ~/Documents/bhax/Welch/Binfapost
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/bhax/Welch/Binfapost$ ./lzwpost <fa.txt
01111001001001000111

---/(1)
-----1(2)
-----1(3)
-----1(4)
-----0(3)
-----0(4)
-----0(5)
-----0(2)
-----1(3)
-----0(3)
melyseg=5
guzsik@guzsikd:~/Documents/bhax/Welch/Binfapost$

```

6.4. Tag a gyökér

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

A bináris fa építését az újonnan létrehozott `LZWTREE` osztályban szeretnénk megvalósítani. Ide kellene még beágyaznunk a fa csomópontjának (`Node`) a jellemzését is. Azért fogjuk beágyazni, mert egyelőre semmilyen különleges szerepet nem kap. Szimplán csak a fa részeként tekintünk rá.

```

#include <iostream>

class LZWTREE
{
public:
    LZWTREE () : fa(&gyoker) {}

    ~LZWTREE ()
    {
        szabadit (gyoker.egyesGyermekek ());
        szabadit (gyoker.nullasGyermekek ());
    }

    void operator<<(char b)
    {
        if (b == '0')
        {

```

```

    if (!fa->nullasGyermek ())
    {
        Node *uj = new Node ('0');
        fa->ujNullasGyermek (uj);
        fa = &gyoker;
    }
    else
    {
        fa = fa->nullasGyermek ();
    }
}
else
{
    if (!fa->egyesGyermek ())
    {
        Node *uj = new Node ('1');
        fa->ujEgyesGyermek (uj);
        fa = &gyoker;
    }
    else
    {
        fa = fa->egyesGyermek ();
    }
}
void kiir (void)
{
    melyseg = 0;
    kiir (&gyoker);
}
void szabadit (void)
{
    szabadit (gyoker.jobbEgy);
    szabadit (gyoker.balNulla);
}

```

A fa konstruktora és destruktora után az LZWTree osztályon belül definiáljuk a balra eltoló bitshift operátor tűlterhelését. Ez segíteni fog az inputról érkező karaktereket "beletolni" a LZWTree objektumba. Így fog felépülni a fa.

```

private:

class Node
{
public:
    Node (char b = '/') : betu (b), balNulla (0), jobbEgy (0) {};
    ~Node () {};
    Node *nullasGyermek () {
        return balNulla;
    }
}
```

```
    }
    Node *egyesGyermek ()
    {
        return jobbEgy;
    }
    void ujNullasGyermek (Node * gy)
    {
        balNulla = gy;
    }
    void ujEgyesGyermek (Node * gy)
    {
        jobbEgy = gy;
    }

private:
    friend class LZWTree;
    char betu;
    Node *balNulla;
    Node *jobbEgy;
    Node (const Node &);
    Node & operator=(const Node &);
};
```

Amennyiben paraméter nélküli a `Node` konstruktur, akkor az alapértelmezett '/'-jellel fogja azt létrehozni. Egyébként a meghívó karakter kerül a betű helyére. A bal és jobb gyermekre mutató mutatókat nulára állítjuk. Az aktuális `Node` minden meg tudja mondani, hogy mi a bal illetve jobb gyermek. Meg is mondhatjuk neki, hogy melyik csomópont legyen ezentúl az új gyermek. Barátjaként deklaráljuk az `LZWTree` osztályt, hogy az dolgozhasson a csomópontokkal.

```
Node gyoker;
Node *fa;
int melyseg;

LZWTree (const LZWTree &);
LZWTree & operator=(const LZWTree &);

void kiir (Node* elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        kiir (elem->jobbEgy);

        for (int i = 0; i < melyseg; ++i)
            std::cout << "___";
        std::cout << elem->betu << "(" << melyseg - 1 << ")" << std::endl;
        kiir (elem->balNulla);
        --melyseg;
    }
}
```

```
    }
    void szabadit (Node * elem)
    {
        if (elem != NULL)
        {
            szabadit (elem->jobbEgy);
            szabadit (elem->balNulla);
            delete elem;
        }
    }
};
```

Mindig az aktuális csomópontra mutatunk. A `kiír` függvénytel jelenítjük meg magát a fát. Felszabadítjuk a két gyermeket, nehogy elfolyjon a memóriánk.

```
int
main ()
{
    char b;
    LZWTree binFa;

    while (std::cin >> b)
    {
        binFa << b;
    }

    binFa.kiir ();
    binFa.szabadit ();

    return 0;
}
```

Bitenként olvassuk a bemenetet, de a fát már karakterenként építjük fel.

Fordítás és futtatás után:

```
$ g++ tree.cpp -o tree
$ ./tree < teszt.txt
-----1(3)
-----1(2)
-----1(1)
-----0(2)
-----0(3)
-----0(4)
---/(0)
-----1(2)
-----0(1)
-----0(2)
```

Megoldás forrása:https://progpter.blog.hu/2011/04/01/imadni_fogjak_a_c_t_egy_emberkent_tiszta_szivbol_2
Továbbá:https://progpter.blog.hu/2011/04/14/egyutt_tamadjuk_meg

6.5. Mutató a gyökér

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

Eddig a LZWTree-ben a fa gyökere mindig egy objektum volt. A fa mutatót kezdetben ráállítottuk és a gyökér referencia értékét adta vissza. Mostantól a gyökér is és a fa is egy-egy mutató lesz. Tehát nem a referenciát adjuk át a fa mutatónak, hanem magát a gyökér mutatót.

```
class LZWTree
{
public:
    LZWTree ()
    {
        gyoker = new Node();
        fa = gyoker;
    }

    ~LZWTree ()
    {
        szabadit (gyoker->egyesGyermek ());
        szabadit (gyoker->>nullasGyermek ());
        delete gyoker;
    }
}
```

A konstruktorban a gyökeret új csomópontra mutató mutatóként hozzuk létre, a destrukturban pedig ugyan-így szabadítjuk fel. Az eddig a gyökér előtt álló referenciajeleket mindenholnan kitörölgetjük.

```
Node *gyoker;
Node *fa;
int melyseg;
```

A csomópontban a védett tagok között is objektumként szerepelt eddig, ott is át kell írni pointerré.

Láásuk hogy fordul-e és fut-e a program a módosítások után?

```
$ g++ pointer.cpp -o pointer
$ ./pointer < teszt.txt
-----1(3)
-----1(2)
----1(1)
----0(2)
-----0(3)
-----0(4)
---/(0)
-----1(2)
----0(1)
-----0(2)
```

6.6. Mozgató szemantika

Írj az előző programhoz mozgató konstruktort és értékkadást, a mozgató konstruktor legyen a mozgató értékkadásra alapozva!

A feladatot az [UDPROG közössége](#) repója segítségével készítettem el.

Mozgató konstruktor helyett itt egy másoló konstruktor:

```
LZWTree& operator= (LZWTree& copy) //másoló értékkadás
{
    szabadit(gyoker->egyesGyermek ());
    szabadit(gyoker->nullasGyermek ());

    bejar(gyoker, copy.gyoker); //rekurzívan bejárom a fákat és ←
        átmásolom az értékeket

    fa = gyoker; //mindkét fában visszaugrok a gyökérhez
    copy.fa = copy.gyoker;
}

void bejar (Node * masolat, Node * eredeti) //rekurzív famásolás, ←
    másoló értékkadáshoz
{
    if (eredeti != NULL) //ha létezik a másolandó fa
    {
        if ( !eredeti->nullasGyermek() ) //ha nem létezik az ←
            eredeti nullasgyermeke
        {
            masolat->ujNullasGyermek(NULL);
        }
        else //ha létezik az eredeti nullásgyermeke
        {
            //létrehozni a masolat nullasgyermeket és meghívni újra a ←
            bejárat
            Node* uj = new Node ('0');
            masolat->ujNullasGyermek (uj);
            bejar(masolat->nullasGyermek(), eredeti->nullasGyermek());
        }

        if ( !eredeti->egyesGyermek() ) //ha nem létezik az eredeti ←
            egyesgyermeke
        {
            masolat->ujEgyesGyermek(NULL);
        }
        else //ha létezik az eredeti egyesgyermeke
        {
            //létrehozni a masolat egyesgyermeket és meghívni újra a ←
            bejárat
        }
    }
}
```

```
Node *uj = new Node ('1');
    masolat->ujEgyesGyermek (uj);
bejar(masolat->egyesGyermek(), eredeti->egyesGyermek());
}
else //ha nem létezik a másolandó fa
{
    masolat = NULL;
}
}
```

7. fejezet

Helló, Conway!

7.1. Hangyszimulációk

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

A program a hangyák kommunikációját hivatott szimulálni. Az alábbi kódcsipetet az ant.h fájlba találhatjuk, amivel csupán a hangyák pozícióját adjuk meg az x és y koordináták segítségével, valamint az irányát, amit a dir változóban tárolunk. A typedef -fel egy alias nevet készítünk , amivel tudunk hivatkozni majd az Ant típusú elemekből álló vektorokra.

```
#ifndef ANT_H
#define ANT_H

class Ant
{
public:
    int x;
    int y;
    int dir;
Ant(int x, int y): x(x), y(y) {
    dir = qrand() % 8;
}
};

typedef std::vector<Ant> Ants;
#endif
```

Nézzük a következő osztályt, az AntWin-t. Ezt megtaláljuk az antwin.h és antwin.cpp fájlokban. A headerben vannak a különböző elemek deklarációi, míg a cpp-ben azoknak a definíciói. Ezáltal egy olvashatóbb, letisztultabb forrást kapunk. Mivel Qt-t használunk inklúdolnunk kell jó pár header fájt. Például QMainWindow-val tudjuk elkészíteni a programunk ablakját, amiben majd a hangyák szaladgálnak. A szaladgáláshoz inklúdolnunk kell a QPainter header-t. Unicode karakterkódolású sztringeket a QString segítségével tudunk tárolni, míg QCloseEvent -nek köszönhetően a program bezárását tudjuk szabályozni paraméterekkel.

Ez felelős a programablak létrehozásáért, beállítja az adott paramétereket. Itt van implementálva a program futása és bezárása. Az egyes billentyűk funkciói is itt lettek meghatározva.

```
#ifndef ANTWIN_H
#define ANTWIN_H
#include <QMainWindow>
#include <QPainter>
#include <QString>
#include <QCcloseEvent>
#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 ( QCcloseEvent *event ) {
        antThread->running=false;
        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:
    FT
    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
```

A Qthread headert azért kell inklúdolnunk, hogy kezelní tudjuk a program egyes szálait. Az AntThread osztály konstruktora megkapja azokat az értékeket, amiket már "korábban" az AntWin osztályban megkapott a program, ezután kezdi el mozgatni a hangyákat. Kiszámolja az irányukat a feromonszintek és a cellatelítettség alapján. Beállítja az egyes cellák feromonértékeit, és megadja az AntWin osztálynak, hogy mely cellákat kell átszínezni.

Futtatáshoz szükséges előkészületek a következők. Fontos, hogy mindegyik fájl egy mappában legyen. Ahhoz, hogy futtatni tudjuk, először le kell töltenünk a qtbase5-dev könyvtárat a szokásos módon.

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

Megoldás forrása:

7.2. Java életjáték

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

Életfolyamatok szimulálása

A Conway-féle sejtautomata szabályai zseniálisan egyszerűek. Az élettér egy négyzetháló. minden cellában egy sejt élhet. minden sejtet nyolc szomszédos cella vesz körül. minden sejt a szomszédjainak létszámtól függően életben marad, vagy meghal. Bizonyos esetekben egy üres cellában új sejt születik. Ha egy generációban egy sejtnak kettő vagy három élő szomszédja van, akkor a sejt élni fog a következő generációban is, minden más esetben a sejt kihal (túlnépesedés, vagy elszigeteltség miatt). Ha egy üres cellának pontosan három élő sejt van a szomszédjában, akkor ott új sejt születik. Az alakzatok viselkedése nagyon hasonlít élő szervezetek változásaihoz (szaporodás, kihalás, fejlődés, visszafejlődés, stb), ezért megilleti a "szimulációs játék" elismerő jelző - egy játék, amely utánozza a valós életfolyamatokat.

Röviden:

- Egy élő sejt meghal, szomszédok száma kisebb, mint 2 vagy nagyobb, mint 3
- Egy sejt életben marad, ha szomszédok száma 2 vagy 3
- Létre jön egy sejt ott, ahol a szomszédok száma pontosan 3

Sikló-kilövőről röviden:

Conway 50 dollárt ígért egy bizonyítottan végtelenül növekedő formáció kiindulási állapotáért. Ekkor az egyetemeken, bankokban, hivatalokban beindult a formációgyártás. Matematikus és laikus, fizikus és méla filzof próbálkozott, hogy felállítsa azt a formációt, amelyik végtelen terjeszkedik, nekifeszülve a virtuális tér bitzónáinak.

A díjat egyébként a Massachusetts Egyetem Mesterséges Intelligencia Csoportjának két tagja nyerte. Egy valóban meglepő felfedezést tettek: találtak egy "sikló kilövőt"! Az induló alakzat "ágyúvá" alakul, amely

először a 40-ik lépésben lő ki egy "siklót", majd ütemesen ismétlődve, minden további 30-ik periódusban egy-egy továbbít. Mivel minden "sikló" születésekor öt újabb sejt kerül a táblára, a népesség nyilvánvalóan korlátlanul növekszik.

A kód a [Javát tanítok](#) oldalán található Sejtautomata szimuláció alapján készült.

```
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 = 10;  
  
    private java.awt.Robot robot;
```

Boolean típusokban deklaráljuk, hogy egy sejt lehet élő vagy halott. A rácsindex mutatja majd az aktuális rácsot. Cellák szélessége és magassága pixelben megadva. A sebességet levettem egészen 10-re, hogy gyorsabban érjük el majd a káoszt.

```
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);  
}  
});
```

A kiinduló rácsmező minden sejtje halott lesz kezdetben. Ezekre fogunk alakzatokat helyezni. Ha az ablakot bezárjuk, a program kilép.

```
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_G)  
            várakozás /= 2;  
        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L)  
            várakozás *= 2;  
        repaint();  
    }  
});
```

A billentyűzetről érkező inputot figyeljük.

- "K" lenyomása esetén a kirajzolt cellák méretét felezzük.
- "N" lenyomása esetén a kirajzolt cellák méretét duplázzuk.
- "G" lenyomása esetén gyorsítunk.
- "L" lenyomása esetén lassítunk.

```
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();
    }
});
```

Az egérről érkező inputot figyeljük. Kattintással és egérhúzással jelölünk ki új sejteket.

```
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);
```

A program indulásakor megjelenő ablak adatai. Az iteráció elindítása.

```
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);
        }
    }
}
```

Kirajzoltatjuk az aktuális sejteket ábrázoló teret. ha élő a sejt akkor feketére színezzük, ha halott, akkor fehérre. A rácsokhoz szürke színt használunk.

```
public int szomszédokSzáma(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;
}
```

Az aktuális sejt nyolc szomszédját számoljuk végig, magát a sejtet kihagyjuk.

```
public void időFejlődés()
{
    boolean [][] rácsElőtte = rácsok[rácsIndex];
    boolean [][] rácsUtána = rácsok[(rácsIndex+1)%2];

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

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

            if(rácsElőtte[i][j] == ÉLŐ)
            {

                if(élők==2 || élők==3)
                    rácsUtána[i][j] = ÉLŐ;
                else
                    rácsUtána[i][j] = HALOTT;
            }
            else
            {

                if(élők==3)
                    rácsUtána[i][j] = ÉLŐ;
                else
                    rácsUtána[i][j] = HALOTT;
            }
        }
        rácsIndex = (rácsIndex+1)%2;
    }
}
```

A sejtek életére vonatkozó (fentebb már említett) 4 szabályt vizsgáljuk itt végig. Ezzel számoljuk ki, hogy a következő körben az adott sejt élő vagy halott lesz.

```
public void run()
{
    while(true)
    {
        try
```

```
{  
    Thread.sleep(várakozás);  
}  
  
catch (InterruptedException e) {}  
  
időFejlődés();  
repaint();  
}  
}
```

A sejtér fejlődése az idő függvényében.

```
public void sikló(boolean [][] rács, int x, int y)  
{  
  
    rács[y+ 0][x+ 2] = ÉLŐ;  
    rács[y+ 1][x+ 1] = ÉLŐ;  
    rács[y+ 2][x+ 1] = ÉLŐ;  
    rács[y+ 2][x+ 2] = ÉLŐ;  
    rács[y+ 2][x+ 3] = ÉLŐ;  
  
}  
  
public void siklóKilövő(boolean [][] rács, int x, int y)  
{  
  
    rács[y+ 6][x+ 0] = ÉLŐ;  
    rács[y+ 6][x+ 1] = ÉLŐ;  
    rács[y+ 7][x+ 0] = ÉLŐ;  
    rács[y+ 7][x+ 1] = ÉLŐ;  
  
    rács[y+ 3][x+ 13] = ÉLŐ;  
  
    rács[y+ 4][x+ 12] = ÉLŐ;  
    rács[y+ 4][x+ 14] = ÉLŐ;  
  
    rács[y+ 5][x+ 11] = ÉLŐ;  
    rács[y+ 5][x+ 15] = ÉLŐ;  
    rács[y+ 5][x+ 16] = ÉLŐ;  
    rács[y+ 5][x+ 25] = ÉLŐ;  
  
    rács[y+ 6][x+ 11] = ÉLŐ;  
    rács[y+ 6][x+ 15] = ÉLŐ;  
    rács[y+ 6][x+ 16] = ÉLŐ;  
    rács[y+ 6][x+ 22] = ÉLŐ;  
    rács[y+ 6][x+ 23] = ÉLŐ;  
    rács[y+ 6][x+ 24] = ÉLŐ;  
    rács[y+ 6][x+ 25] = ÉLŐ;  
  
    rács[y+ 7][x+ 11] = ÉLŐ;
```

```

rács[y+ 7][x+ 15] = ÉLŐ;
rács[y+ 7][x+ 16] = ÉLŐ;
rács[y+ 7][x+ 21] = ÉLŐ;
rács[y+ 7][x+ 22] = ÉLŐ;
rács[y+ 7][x+ 23] = ÉLŐ;
rács[y+ 7][x+ 24] = ÉLŐ;

rács[y+ 8][x+ 12] = ÉLŐ;
rács[y+ 8][x+ 14] = ÉLŐ;
rács[y+ 8][x+ 21] = ÉLŐ;
rács[y+ 8][x+ 24] = ÉLŐ;
rács[y+ 8][x+ 34] = ÉLŐ;
rács[y+ 8][x+ 35] = ÉLŐ;

rács[y+ 9][x+ 13] = ÉLŐ;
rács[y+ 9][x+ 21] = ÉLŐ;
rács[y+ 9][x+ 22] = ÉLŐ;
rács[y+ 9][x+ 23] = ÉLŐ;
rács[y+ 9][x+ 24] = ÉLŐ;
rács[y+ 9][x+ 34] = ÉLŐ;
rács[y+ 9][x+ 35] = ÉLŐ;

rács[y+ 10][x+ 22] = ÉLŐ;
rács[y+ 10][x+ 23] = ÉLŐ;
rács[y+ 10][x+ 24] = ÉLŐ;
rács[y+ 10][x+ 25] = ÉLŐ;

rács[y+ 11][x+ 25] = ÉLŐ;
}

```

A glider és a glider-gun elhelyezése a sejttérben.

```

public void update(java.awt.Graphics g)
{
    paint(g);
}

public static void main(String[] args)
{
    new Sejtautomata(100, 75);
}
}

```

Végül teszünk róla, hogy ne villogjon az ablak és egy 100 x 75-ös mérettel példányosítjuk a létrehozott objektumot.

Fordítjuk és futtatjuk:

```
$ javac Sejtautomata.java
```

```
$ java Sejtautomata
```

Futás indulásakor a glider-gunok tökéletesen működnek. Szépen sorra lövik ki a siklókat, míg a végén az egész élettér betelik és kihal a pálya.

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html#id564903>

Javát tanítok: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html#id564903>

7.3. Qt C++ életjáték

Most Qt C++-ban!

A main.cpp felépítése:

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

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

    return a.exec();
}
```

A beincludált sejtablak.h:

```
#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
```

Itt található az élő vagy halott sejtek boolean változója. A sejttér mérete. A rács többdimenziós tömbjeire mutató mutatók.

sejtablak.cpp:

```
#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;

    racs[y+ 3][x+ 13] = ELO;

    racs[y+ 4][x+ 12] = ELO;
    racs[y+ 4][x+ 14] = ELO;

    racs[y+ 5][x+ 11] = ELO;
    racs[y+ 5][x+ 15] = ELO;
    racs[y+ 5][x+ 16] = ELO;
    racs[y+ 5][x+ 25] = ELO;

    racs[y+ 6][x+ 11] = ELO;
    racs[y+ 6][x+ 15] = ELO;
    racs[y+ 6][x+ 16] = ELO;
    racs[y+ 6][x+ 22] = ELO;
    racs[y+ 6][x+ 23] = ELO;
    racs[y+ 6][x+ 24] = ELO;
```

```
racs[y+ 6][x+ 25] = ELO;  
  
racs[y+ 7][x+ 11] = ELO;  
racs[y+ 7][x+ 15] = ELO;  
racs[y+ 7][x+ 16] = ELO;  
racs[y+ 7][x+ 21] = ELO;  
racs[y+ 7][x+ 22] = ELO;  
racs[y+ 7][x+ 23] = ELO;  
racs[y+ 7][x+ 24] = ELO;  
  
racs[y+ 8][x+ 12] = ELO;  
racs[y+ 8][x+ 14] = ELO;  
racs[y+ 8][x+ 21] = ELO;  
racs[y+ 8][x+ 24] = ELO;  
racs[y+ 8][x+ 34] = ELO;  
racs[y+ 8][x+ 35] = ELO;  
  
racs[y+ 9][x+ 13] = ELO;  
racs[y+ 9][x+ 21] = ELO;  
racs[y+ 9][x+ 22] = ELO;  
racs[y+ 9][x+ 23] = ELO;  
racs[y+ 9][x+ 24] = ELO;  
racs[y+ 9][x+ 34] = ELO;  
racs[y+ 9][x+ 35] = ELO;  
  
racs[y+ 10][x+ 22] = ELO;  
racs[y+ 10][x+ 23] = ELO;  
racs[y+ 10][x+ 24] = ELO;  
racs[y+ 10][x+ 25] = ELO;  
  
racs[y+ 11][x+ 25] = ELO;  
  
}
```

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.cpp:

```
#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)
    {
        for(int j=0; j<szelessseg; ++j)
        {

            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()
{
}
```

7.4. BrainB Benchmark

Ez egy játék, aminek a lényege, hogy samu entropy-n kell tartani az egeret. Miközben az egér samun van, addig újabb és újabb entropy-k születnek ezzel nehezítve ajátékot. Hiszen minél több entropy van, annál nehezebb samu mozgását követni. A játék igazából az agy kognitív képességeit méri fel az adott 10 perces játékidő alatt. A játék végén készül egy screenshot, na meg egy txt amiben a játék során keletkezett adatokat tudjuk vissza nézni.

Megoldás forrása:

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

Megoldás videó: <https://youtu.be/j7f9SkJR3oc>

Megoldás forrása: <https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0> (/tensorflow-0.9.0/tensorflow/exa...
https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol

Tanulságok, tapasztalatok, magyarázat...

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Minecraft-MALMÖ

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:

Röviden a Lisp-ről.

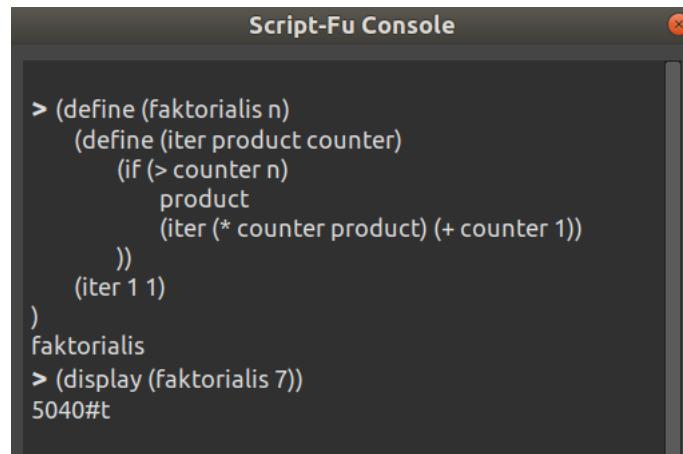
A Lisp egy 1950-es években kialakulú nyelv, ami a mai napig fontos szerepet játszik a programozás területén. A mesterséges intelligencia területén is használják. A Lisp egy listafeldolgozó elven működő nyelv (list processing). A köznapi infix feldolgozástól eltérően a Lisp dialektusai prefix alakkal dolgozik. Magyarán szólva az operátorok függvényeknek felelnek meg, melyeknek paraméterei az operanduszok.

A Scheme nyelv a LISP-ből származik. A szintaxis javarészét megegyezik a LISP-ével. Mindkét nyelv fő alkalmazási területe a mesterséges intelligencia. A Scheme gyengén típusos nyelv dinamikus típusellenőrzéssel és lexikális hatásköri szabályokkal.

A Scheme dialektus ismertetésére implementáljuk iteratív, majd rekurzív módon a faktoriálist kiszámító algoritmust:

```
(define (faktorialis n)
(define (iter product counter)
  (if (> counter n)
      product
      (iter (* counter product) (+ counter 1)))
  )
(iter 1 1))<para>
</para>

(display (faktorialis 3))
```



```
Script-Fu Console > (define (faktorialis n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product) (+ counter 1)))
  )
  (iter 1 1)
)
faktorialis > (display (faktorialis 7))
5040#t
```

Faktoriális rekurzívan

A kifejezések egytől egyig lista alakúak. A "define" kulcsszó a lista legelső eleme és ez hozza létre az eljárásunkat. A (faktorialis n) pedig a listánkon belül egy allista, aminek első eleme az eljárásunk neve, utána pedig a paraméterek következnek.

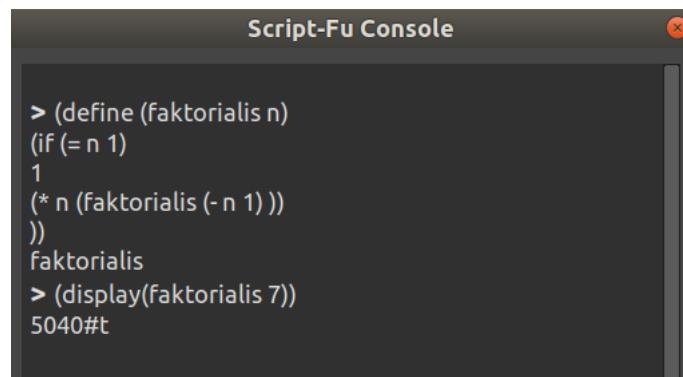
Az "iter" néven definiált eljárást használjuk az iteráció leképzéséhez. Ez az eljárás két argumentumot vár, azaz egy-egy változót. Első argumentumként a ciklusváltozót kapja meg, majd a számlálót. Az iter-en belül teszteljük "n" segítségével a szabad változó értékét. Ha meghaladja az értéke a "counter" értékét, akkor vissza tér a kiszámolt faktoriális értékkel. Ha nem haladja meg, akkor számol tovább.

Rekurzív megoldással sokkal egyszerűbb a kivitelezés.

```
(define (faktorialis n)
  (if (= n 1)
      1
      (* n (faktorialis (- n 1) )))
  )
)

(display (faktorialis 3))
```

Egyszerűbb és olvashatóbb. Mondhatni erre találták ki a Scheme-t.



```
Script-Fu Console > (define (faktorialis n)
  (if (= n 1)
      1
      (* n (faktorialis (- n 1) )))
  )
  faktorialis > (display(faktorialis 7))
5040#t
```

Faktoriális rekurzívan

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

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

A kódot innen érheted el: [Chrome](#)

Feladatunk végeredménye A GIMP képszerkesztő egy kiterjesztése lesz. Ahhoz, hogy futtassuk a .scm fájlunkat be kell helyezni a GIMP scripts könyvtárába.

Előző feladatban Scheme-ben írtunk szkriptet, most pedig Script-fu szkriptet írunk. A Script-fu egy Scheme dialektuson alapuló nyelv, ami a GNU Image Manipulation program nyelve. A Script-fu segítségével lényegében automatizálni tudunk egy chrome hatású effektet, következő feladatban pedig ennek segítségével fogunk mandalát is készíteni.

Minden egyes GIMP funkció (forgítás, nagyítás, tükrözés, stb.) benne van egy adatbázisban. A Script-fu nyelvben tudjuk használni lekérdezéssel és meghívással ezeket a GIMP függvényeket. Az adatbázis GUI-n keresztül elérhetjük az összes függvényt, illetve böngészhetünk benne. Ezáltal könnyebb a meló.

A forráskód röviden:

```
(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)
)
```

A színek manipulálásához szükséges adatokat itt tároljuk. Ez az eljárás lényegében egy tömböt ad vissza. A "let*" segítségével hozunk létre lokális változókat.

```
; (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)
)
)
```

Egy rekurziós függvényt láthatunk ami kiveszi a megadott indexű elemet a listából. A "car" a lista első elemét adja vissza, itt akkor ha az index egyenlő 1-gyel, ha ez nem teljesül akkor levonunk az "x" értékből 1-t és meghívódik a "cdr" ami a listát adja vissza az első elem nélkül.

Következő rekurziós függvényünk az krómosítani kívánt szöveg szélességét határozza meg, ez kell a rajzoláshoz. Láthatjuk, hogy immár GIMP-s függvényekhez folyamodunk. Meghatározzuk a szöveg szélességét felhasználva a `gimp-text-get-extents-fontname` első visszatérési értékét, ami a szélesség. Ezt értékül adjuk a `text-width` változónknak a `set!` parancsal.

Az előbb megírt keresési eljárást felhasználva pedig megszerezzük a második visszatérési értékét is, ami a magasság.

```
; (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)))
```

Megkezdjük a tényleges krómósítási algoritmus implementálását az előbb említett cikk alapján.

A helyi változók deklarálása és inicializálása után megkezdjük a lépések megvalósítását.

Először létre kell hoznunk egy réteget, nem üres réteget hozunk létre, mert egy általunk definiált Úrlapon megkérjük a felhasználót, hogy adjon meg valamilyen karakterláncot amit szeretne krómósítani néhány paraméter mellett.

Folytatva az első lépését, felhasználva a bekért paramétereket, létrehozunk egy fekete hátterű képet, amin az input szöveg színé fehér lesz. Töröljük a kijelölést, hogy elkerüljük az előreláthatatlan következményeit ha meghagyjuk.

```
;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)
```

Az angolul *Gaussian* szóval megilletett elmosást alkalmazzuk a jelenlegi képre, hogy kisimítsuk a szöveget, tehát ne legyen olyan élesek a betűk.

A 3. lépéshoz beállítjuk, hogy a szövegnek lekanyarítottak legyenek élei a megfelelő színekre vonatkozó konfigurációs paraméterek testreszabásával, majd a 4. lépéshoz az enyhített elmosást alkalmazunk.

```
;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)
)
```

Invertáljuk a kijelölést, hogy létrehozzunk egy átlátszó réteget, tehát eltűnik a fekete háttér. A szövegre alkalmazzuk a színátmenetet, amit megadott a user, majd alkalmazzuk a BumpMap függvényt rá megfelelő paraméterekkel. A 9. lépéshoz történik meg a csoda: létrehozunk egy polinomot, egy görbü(spline), aminek tulajdonságait állítgatva, különböző módon kerül alkalmazva a képre a króm effektus. Végén átméretezzük a képünket, majd megjelenítjük az eredményt a usernek.

```
)
; (script-fu-bhax-chrome-border "Bátf41 Haxor Stream" "Sans" 160 1920 1080 ←
400 '(255 0 0) "Crown molding" 7)
; (script-fu-bhax-chrome-border "Programozás" "Sans" 110 768 576 300 '(255 0 ←
0) "Crown molding" 6)
```

```
(script-fu-register "script-fu-bhax-chrome-border"
  "Chrome3-Border2"
  "Creates a chrome effect on a given text."
  "Norbert Bátfai"
  "Copyright 2019, Norbert Bátfai"
  "January 19, 2019"
  ""
  SF-STRING      "Text"        "Bátf41 Haxor"
  SF-FONT        "Font"        "Sans"
  SF-ADJUSTMENT  "Font size"   '(160 1 1000 1 10 0 1)
  SF-VALUE       "Width"       "1920"
  SF-VALUE       "Height"      "1080"
  SF-VALUE       "New width"   "400"
  SF-COLOR       "Color"       '(255 0 0)
  SF-GRADIENT    "Gradient"    "Crown molding"
  SF-VALUE       "Border size" "7"
)
(script-fu-menu-register "script-fu-bhax-chrome-border"
  "<Image>/File/Create/BHAX"
)
```

Itt látható kód azért kell, hogy a GIMP menüjébe beépíthessük a szkript hívását. Egy űrlapot jelenít meg, ami segítségével bekéri a szükséges paramétereket a felhasználótól. Mi a kódban egy alapértelmezett beállítást is megadunk. Végül megadjuk a szkriptünk elérési útvonalát a gui-n belül az utolsó függvényhívással.



Chrome effekt így néz ki

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_hackelete_a_scheme_programozasi_nyelv

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

A kódot innen érheted el: [Mandala](#)

A mandala egy indiai vallási motívum. Lényegében egy olyan ábra, mely középpontosan forgatva többször is kiadja önmagát. Azaz miközben 360 fokos forgatást végezünk a középpontja körül néhány fokonként újra az eredeti ábrát láthatjuk.

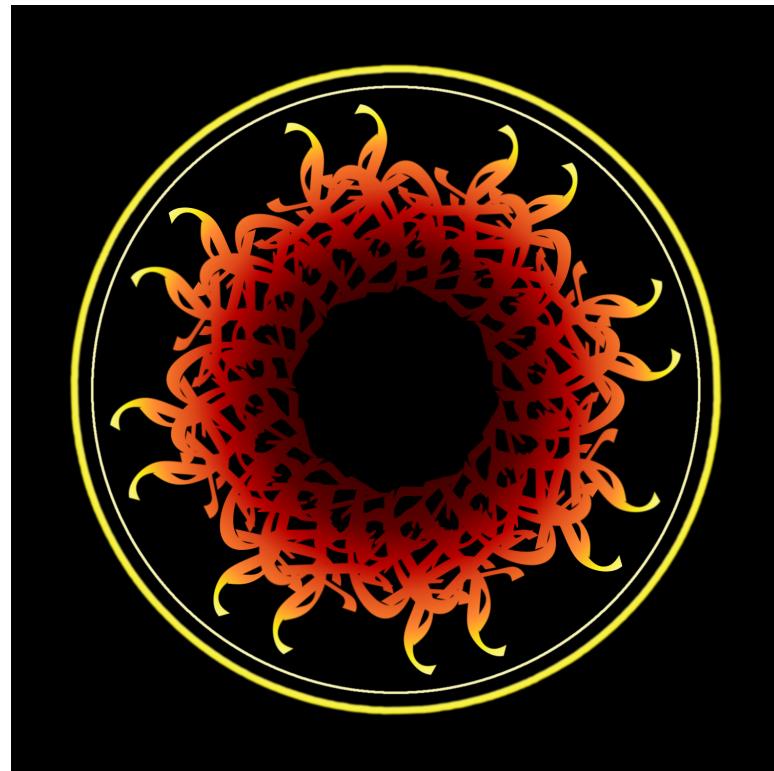
Jelen esetben saját névből kell létrehozni egy mandalát. Ezt úgy kell kivitelezni, hogy:

- Kiválasztunk egy számunkra szép betűtípusat, érdemes valamilyen kacifántosat.
- Leírjuk a GIMP-pel a képünk közepére a szövegünket, érdemes rövidet felfirkálni.
- Készítünk róla egy másolatot, majd vízszintesen tükrözük az eredeti tetejére illesztve.
- Összevonjuk a két réteget, majd egy újabb másolatot hozunk létre amit elforgatunk a kép közepénél 90 fokkal.
- Összevonjuk megint a két réteget, majd egy újabb másolatot hozunk létre amit elforgatunk a kép közepénél 45 fokkal.
- Összevonjuk megint a két réteget, majd egy újabb másolatot hozunk létre amit elforgatunk a kép közepénél 30 fokkal.
- Rajzolunk két különböző sugarú és keretvastagságú kört a szöveg köré.
- És végül összevonás után kiválasztunk és alkalmazunk a képen egy tetszőleges színátmenetet.

Mint előző feladatban is, most is a .scm fájlt be kell helyezni a GIMP scripts könyvtárába, hogy futtathassuk. Futtatás után a következő képeket kaphatjuk:



Bátfai tanárúr eredeti kódja alapján



Kicsit átírva saját vezetéknévvel

10. fejezet

Helló, Gutenberg!

10.1. Juhász István, Magasszintű programozási nyelvek I.

1. Alapfogalmak

A napjainkra kifejlődött programozási nyelveket három szintre sorolják. Első szinten a hardver közeléi gépi nyelvek vannak. Második szinten az assembly szintű nyelvek helyezkednek el, míg végül harmadik szinten a magas szintű programozási nyelvek vannak. A harmadik szinten lévő nyelvek általának legközelebb a programozókhöz, illetve felhasználókhöz. A magas szintű programozási nyelven készített programokat forráskódoknak nevezzük.

A számítógépek processzorai saját gépi nyelvezetű programokkal képes dolgozni. Viszont ezen nehézkes egy ember számára kódolni. Megoldás, hogy valamilyen úton - módon a magas szintű nyelven megírt programokat gépi nyelvre fordítsuk. Erre két lehetőségünk is van. Fordítóprogramok segítségével vagy interpreteres megoldással kell fordítanunk.

Mi is az a fordítóprogram? A fordítóprogram segít nekünk gépi kóddá (tárgyprogrammá) alakítani a magas szintű nyelven megírt programunkat. A fordítóprogram futtatás vagy fordítás közben több szempontból vizsgálja meg a forráskódunk. Először lexikális elemzést hajt végre, majd megnézi hogy szintaktikailak helyes-e a kódunk, ezek után szemantikailag is megvizsgálja. Ha az eddigiek minden megfeleltek a fordító program elvárásainak, akkor generálja a gépi kódot. Hozzá kell tenni, hogy ezen lépések alatt a forráskód egésze egy egységeként van kezelve.

2. Programnyelvek osztályozása: Imperatív és Dekleratív nyelvek

Az imperatív nyelvek algoritmikus nyelvek, amik szorosan kapcsolódnak a Neumann-architektúrához. Ezen nyelveken írt forráskódok sorozatából álló programok, melyeknek legfontosabb eszköze a változó. A forráskód algoritmusai a változók értékének manipulálásával változtatja a memória tartalmát. Két alcsoportja van, az objektumorientált és eljárásorientált nyelvek.

Az imperatív nyelvekkel ellentétben a deklaratív nyelvben a programozónak csak azt kell megmondania, hogy mit akar, és az algoritmust az értelmező- vagy fordítóprogram állítja elő. Egy deklaratív nyelv nem kötődik szorosan a Neumann-architektúrához, és nem algoritmikus. A deklaratív programozás két válfaját szokás megkülönböztetni: a logikai és a funkcionális programozást.

Van egy harmadik csoport is, a "más elvű nyelvek". Ebbe a csoportba azon nyelvek tartoznak, amik nem imperatív és nem deklaratív nyelvek.

3. Alapelemek

A karakterkészlet a forráskód megírásának alapköve. A karakterkészlet elemeiből képezhetjük le a programkódunkat, ezen elemekből hozhatjuk létre a bonyolultabb nyelvi elemeket. Különböző nyelvek eltérő karakterkészlettel dolgozhatnak.

Lexikális egységek a forráskód azon részei, melyek fordítás során egyetlen különálló egységgé válnak értelmezve. Fajtái a következők: többkarakteres szimbólum, szimbolikus név, címke, jegyzés, és literál.

Többkarakteres szimbólum egyfajta jelentéssel bír az adott nyelv számára. Leginkább operátorok vagy elhatárolók formájában jönnek elő a többkarakteres szimbólumok. (pl.: C nyelvben ++ operátor, vagy a komment jel //)

Szimbolikus nevek a programozó segítségére vannak. Három fajtája van: azonosító, kulcsszó, standard azonosító.

Az azonosító arra lett kitalálva, hogy a programozó az eszközeinek nevet adhasson és a későbbiekben ezen a néven hivatkozhasson azokra. Szintaktikailag megvannak határozva mégpedig a következő képpen: betűvel kell kezdődnie és betűvel vagy számmal kell folytatódnia.

A legtöbb nyelvnek megvannak a saját kulcsszavai. Ezeknek a kulcsszavaknak (más néven foglalt szó, vagy alapszó) az adott nyelv jelentést tulajdonít, amit persze a programozó nem tud megváltoztatni. A kulcsszavak az utasítások megadását szolgálják.

Standard azonosítónak hívjuk azokat a kulcsszavakat, amiket a programozó képes megváltoztatni. Leginkább függvények azonosítására használjuk.

A címke utasítások megjelölésére szolgál az eljárásorientált nyelvekben.

A megjegyzés a programozó dolgát segíti. A megjegyzés (vagy komment) a fordító által figyelmen kívül hagyott karaktersorozatot jelent. Hasznos a forráskód elemeinek és azon tulajdonságainak, esetleg fontosságának a felírása komment formában.

Literál egy fix értéket meghatározó programozási eszköz. Egy literál adattípus és érték komponensekből felépülő adat.

4. Adattípusok

Minden nyelv rendelkezik saját adattípusokkal, de mi is adhatunk meg újabb adattípust. Az adattípusok fontos tulajdonságai a következők: a tartományuk, a valük végezhető műveletek és reprezentációjuk. Két alapvető részre oszthatjuk az adattípusokat, mégpedig skalár (egyszerű) és struktúrált (összetett). Az egyszerű adattípusok elemi értékeket tartalmaznak, míg a struktúrált adattípusok több akár különböző adattípusú elemekből állnak össze.

Egyszerű típusok a következők:

Az egész típusú adatok számértékeket tartalmazhat, amik nem lehetnek lebegőpontosak. Ezek típusok eltérő memória tartománnyal rendelkezhetnek (short, long).

A valós típus a lebegőpontos számok ábrázolásához szükséges típus.

A karakteres típus egy karakterből, míg a string típus egy karakterlánc azaz több karaktert tartalmaz. Ábrázolásukhoz szükséges az alkalmazott kódtáblától függően egy vagy két bajt karakterenként.

Logikai típus két értéket vehet föl, ezek az igaz vagy hamis értékek. Általában igaz az 1, míg a hamis a 0 értékkel egyenlő.

Felsorolásos típus egy a programozó által definiálható típus.

Összetett típusok a következők:

Tömb típus egy nagyon egyszerű összetett típus. Egy tömb statikus és homogén, azaz rögzített elemszámú és minden elem azonos típusú.

Mutató típus tartományába a tárhely címek kerültek. Tárhelyek indirekt címzésére használják. Speciális eleme egy nem valódi tárcím, ami a nyelvben null néven szerepel.

5. Nevesített konstans

Olyan literál, amelyhez egy azonosító nevet társítottak. Így név, típus és érték komponensekből áll össze. Kötélező deklarálni használat előtt. Használata akkor válik igazn hasznossá, ha a forráskódban sokszor utalunk a literálra annak nevével és a literál értékét megakarjuk változtatni. Ekkor elég csak a deklarációban változtatni.

6. A változó

Egy változó név, attribútum, érték és cím komponensekből áll össze. A nevével tudunk hivatkozni a deklarálás után a változóra. Az attribútum a változó viselkedését határozza meg. Explicit és implicit módon is lehet deklarálni egy változót. A változóknak van élettartamuk, és a címkékomponens meglétéből tudjuk ezt meghatározni. Egy cím lehet statikus, dinamikus, esetleg megadhatjuk tárkiosztással is a címet.

7. Alapelemek

Két r

8. Kifejezések

9. Utasítások

Az eljárásorientált nyelvekben az eljárások részeit adjuk meg az utasítások segítségével. Ezen felül a fordítóprogram az utasítások segítségével képzi le a tárgyprogramot. Két nagy fajtája van, mégpedig a deklarációs és a végrehajtó utasítás.

A deklarációs utasítás a fordítóprogramnak szól, nincs mögöttük tárgykód. A tárgykódot a végrehajtó utasításokból képzi le a fordító program.

A végrehajtó utasítások a következők:

Az értékkadó utasítással lehet megadni a változók értékkomponenseit vagy módosítani azt a program futásának bármely pillanatában.

Üres utasítás csak pontosvesszőt tartalmaz. Formailag utasítást igénylő helyre írjuk, ha ott egyébként nem kell semmit végrehajtani.

Ugró utasítás segítségével egy megjelölt végrehajtható utasításhoz ugrunk.

Kétirányú elágaztatás utasítás használatos, ha a program futása közben két utasítás közül szeretnénk választani.

Többirányú elágaztatás utasítás segítségül szolgál ha feltételhez kötötten választani szeretnénk a program folytatását illetően több, egymást kizáró utasítás közül. A feltétel egy kifejezés és értékei által történik meg a választás.

Ciklusszervező utasítás segít nekünk, hogy egy utasítást tetszés szerint akárhányszor végrehajthassunk.

Feltételes ciklus utasítás abban különbözik egy ciklus utasítástól, hogy a ciklus végrehajtásának száma egy feltétel igazság értékétől függ. Kétféle feltételes ciklus utasításról beszélhetünk (kezdőfeltételes és végfeltételes) mégpedig a feltétel ciklusban lévő helyzete alapján. Ha a ciklus fejében helyezkedik el a feltétel, akkor kezdő feltételes ciklus utasításról beszélünk. Egy kezdőfeltételes ciklus először kiértékelődik, aztán igazságértekétől függően folytatódik (C-ben while ciklus). Egy végfeltételes (C-ben do) ciklus először végrehajtódik, aztán a feltétel igazságértekétől függően újra végrehajtódik vagy sem (itt a feltétel a ciklus végén helyezkedik el).

Előírt lépésszámú ciklusnál a fejben adjuk meg a ciklusparamétereket. A ciklusparaméterek között szerepel egy ciklusváltozó, aminek a felvett értékeivel fut le a ciklusmag. A ciklusparaméter tartalmaz egy kezdő és egy vég értéket is, ami meghatározza a ciklusváltozó értékkészletének tartományát és egy lépésközöt, ami az értékkészleten belül felvethető értékek távolságát adja meg. Ez a ciklus elöltesztelő vagy hältutesztelő is lehet.

Felsorolásos ciklus egy általánosított előírt lépésszámú ciklus. Explicit módon meghatározott értékeket vesz fel a ciklusváltozó és az összes felvett érték mellett lefut a ciklusmag. Nem lehet üres vagy végtelen utasítás.

Végtelen ciklus sem a fejben sem a ciklusvégen nem tartalmaz információt a program futásának ismétlődéséről. Egy végtelen ciklus feltétel nélkül újra és újra lefut ideális körülmények között, mivelhogy nincs kilépési feltétel.

Összetett ciklus tetszőlegesen sok ciklusfeltétel adható meg a ciklusfejben.

A c nyelvben az eddig tárgyaltakon kívül még három vezérlő utasítás létezik, mégpedig: continue, break, return. A continue utasítás nem hajtja végre a ciklusmagon belül lévő későbbi utasításokat, hanem a ciklus ismétlődés-feltétel vizsgálat után vagy újabb cikluslépésbe kezd vagy befejezi a ciklust. A break utasítást követően befejezi a ciklust és kilép az elágaztató utasításból. A return utasítás szabályosan befejezi a függvényt és a vezérlést vissza adja a hívónak.

10. A programok szerkezete

Kétféle opció létezik. Vagy egyben kell fordítani a forráskódot, vagy feltördelhető külön-külön fordítható részekre.

Vannak olyan nyelvek, amikben a forráskód több egymástól elkülönülő részeiből áll, amiket külön kell fordítani. Ezek mélységében nem struktúrált felépítésű forráskódok.

Egyes nyelvekben a forráskód mélységében struktúrált felépítésű és egy egységeként kell fordítani a programkódot.

Léteznek olyan nyelvek, amikben az előbbi két szerkezet kombinálva létezik. Ekkor a forráskód fizikailag elkülönülő mélységében struktúralható részeiből áll.

Az eljárásorientált nyelvek programegységei: alprogram, blokk, csomag, taszk

Alprogramok

Az alprogramok használata megkönnyíti az összetettebb programok elkészítését. Segítségükkel jól áttekinthető, könnyen javítható vagy módosítható strukturált programszerkezetet alakíthatunk ki. Egy alprogram akkor hasznos, ha egy programrész többször is megjelenik a forráskódunkban. Ezt az ismétlődő programrészt kiemeljük az előfordulási helyekről és elkülönítjük úgy, hogy hivatkozni tudunk rá. Az előfordulási helyeken csak az alprogramra való hivatkozás fog szerepelni.

Hívási lánc, rekurzió

Egy programrész meghívhat egy másik programrészét, ami szintén meghívhat egy programrészét és ez mehet még így sokáig. Ezen hívások sokaságát nevezzük hívási láncnak. A lánc első programegységét főprogramnak nevezzük. A lánc minden tagja aktív, de csak az legutoljára meghívott tag működik. Rekurzióról beszélünk, ha egy aktív tagot újból meghívunk. Közvetlen rekurzió az, amikor egy alprogramban szerepel egy hivatkozás saját magára. Közvetett rekurzió pedig az, amikor egy alprogram a hívási láncban egy már korábban szereplő aktív tagot hív meg.

7. Alapelemek

Az aritmetikai típusok egyszerűek elemeivel műveletek végezhetőek. Integrális típusok: egész, karakter, felsorolásos. Valós típusok: float, double, long double.

Származtatott típusok: tömb, függvény, mutató, struktúra, union, void típus .

Explicit deklarációs utasítás: [CONST] típusleírás eszközazonosítás [= kifejezés]

- azonosító : típusleírás típusú változó
- (azonosító) : típusleírás függvényt címző mutató típusú változó
- *azonosító : típusleírás típusú eszközt címző mutató típusú változó
- azonosító() : típusleírás visszatérési típusú függvény
- azonosító[] : típusleírás típusú elemeket tartalmazó tömb típusú változó
- Saját típus definiálása: TYPEDEF típusleírás név [, típusleírás név] ...;
- Struktúra deklarálása: STRUCT [struktúratípus_név] {mező_deklarációk} [változólista];
- Union deklarálása: UNION [uniontípus_név] {mező_deklarációk} [változólista];

8. Kifejezések

Két komponensű (érték és típus komponens) szintaktikai eszközök. operandusokból (értékeket képvisel), operátorokból (műveleti jeleket képvisel) és kerek zárójelekből (műveleti sorrendet határoz meg) áll. Az operátorok műveletvégzéséhez szükséges operandusok számától függően beszélhetünk egyoperandusú (unáris), kétoperandusú (bináris), vagy háromoperandusú (ternáris) operátorokról.

Kétoperandusú operátoroknál az operandusok és az operátor sorrendje lehet:

- prefix (az operátor az operandusok előtt áll)
- infix (az operátor az operandusok között áll)
- postfix (az operátor az operandusok mögött áll)

A fordítóprogramok az infix kifejezésekkel postfix kifejezéseket állítanak elő, majd a műveletvégzés aszerint történik.

Vannak típusügyenértékűséget és vannak a típuskényszerítést használó nyelvek. A C a numerikus típusoknál megengedi a típuskényszerítést. Itt beszélhetünk bővítésről és szűkítésről. Bővítésnél az átalakítás értékvesztés nélkül végrehajtható. A szűkítés ennek az ellenkezője, itt az átalakításnál értékcsökktítés, esetleg kerekítés történhet. Konstans kifejezés az a kifejezés, aminek értéke fordítási időben eldől.

A C kifejezésorientált nyelv. A típuskényszerítés elvét vallja.

Precedenciatáblázat a C nyelvben (fentről-lefelé erősebbtől-gyengébb felé haladva):

Függvényoperátor és műveleti sorrendet változtató zárójel	()
Tömboperátor	[]
Minősítő operátor	.
Mutatóval minősítő operátor	->
(balról jobbra kötnek)	
<hr/>	
csillag/mutató típusú operátor	*
Tárcímet megadó operátor	&
Plusz előjel	+
Mínusz előjel	-
Ha az operandus értéke nem nulla, akkor az eredmény nulla, egyébként 1	!
Komplemens operátor	~
Operandus értékét egyel növelő	++
Operandus értékét egyel csökkentő	--
A típus ábrázolási hosszát adja bájtban (jobbról balra kötnek)	SIZEOF (típus)
<hr/>	
Szorzás	*
Osztás	/
Maradékképzés	%
(balról jobbra kötnek)	
<hr/>	
Összeadás	+
Kivonás	-
(balról jobbra kötnek)	
<hr/>	
Jobbra léptető bitshift	>>
Balra léptető bitshift	<<
(balról jobbra kötnek)	
<hr/>	
Kisebb	<
Nagyobb	>
Kisebb vagy egyenlő	<=
Nagyobb vagy egyenlő	>=
(balról jobbra kötnek)	
<hr/>	
Egyenlő	==
Nem egyenlő	!=
(balról jobbra kötnek)	

Bitenkénti és (balról jobbra köt)	&
Bitenkénti kizáró vagy (balról jobbra köt)	^
Bitenkénti vagy (balról jobbra köt)	
Logikai és (balról jobbra köt)	&&
Bitenkénti vagy (balról jobbra köt)	
Háromoperandusú operátor (balról jobbra köt)	?:
Értékadó operátorok (jobbról balra kötnek)	= += -= *= /= %= >>= <<= &= ^= =

10. A programok szerkezete

Egyben kell-e lefordítanunk a program teljes szövegét, vagy fel kell osztanunk önállóan fordítható részekre?

- Van olyan programnyelv melyben a program önálló részeiből áll, ezek akár külön is fordíthatóak. Nem strukturáltak.
- Vannak nyelvek, ahol a program csak egy nagy egységként fordítható. A programegységek nem függetlenek.
- Létezhet az első kettő kombinációja is. A nyelvben van független programegység, de strukturával rendelkezik.

Az eljárásorientált nyelvek egységei:alprogram, blokk, csomag, task.

10.1. Alprogramok

Az eljárásorientált nyelvek paradigmáit meghatározza. Újrafelhasználható, ha a program különböző részein a programrész megismétlődik. Egyszer írjuk meg, majd később hivatkozunk rá, újra meghívjuk. Az alprogram felépül fejből (specifikáció), törzsből (implementáció), végből. Négy komponense van: név, formális paraméter lista, törzs, környezet. A név a fejben szereplő azonosító. A formális paraméterlistában a paraméterek nevei és a futás közben szabályozó információik szerepelnek. Ha a paraméter lista üres, akkor paraméter nélküli az alprogram. A törzsben kapnak helyet a deklarációs és végrehajtó utasítások. Az alprogramon belül deklarált eszközök és nevek az alprogram lokális eszközei és nevei. Ezek az alprogramon kívül láthatatlanok. Ellenkezői a globális nevek, amiket az alprogramon kívül deklarálunk. A globális változók együttese a környezet. A függvény egyetlen értéket meghatározó alprogram. A függvény neve minden megadja a visszatérési értékét is. Az eljárás akkor fejeződik be szabályosan, ha elérjük

a végét, vagy külön utasítást adunk ki a befejezésre. minden nyelvben minden programban lennie kell egy főprogramnak. Ez felügyeli az összes többi alprogram működését. A főprogram szabályos lefutása után a vezérlés visszakerül az operációs rendszerhez.

10.2. Hívási lánc , Rekurzió

A hívási lánc, ha egy programegység meghív egy másik programegységet, majd az is meghív egy újabb programegységet, stb. A hívási lánc első eleme mindig a főprogram, minden tag aktív, de csak a legutólag meghívott egység fog működni. Az aktív program újból meghívását rekurzióinak hívjuk. Közvetlen rekurzió esetén az alprogram önmagát hívja meg. Közvetlen rekurzió esetén már korábban meghívott alprogram kerül meghívásra.

10.3. A blokk

Csak egy programegység belsejéban állhat. Van kezdete, törzse, vége. A törzsben vannak a végrehajtó utasítások. A blokk kezdete előtt álló címke a blokk neve. Ahhoz hogy a blokk aktív legyen meg kell várni, hogy rá kerüljön a vezérlés, vagy GOTO utasítással a kezdetére ugorhatunk. Ha a blokk eléri a végét, vagy GOTO-val kiugrunk, akkor a blokk működése befejeződik.

11. Paraméterkiértékelés

A paraméterkiértékelés nem más, mint egy olyan folyamat, ahol a formális és aktuális paraméterek egymáshoz rendelődnek, egy alprogram hívásánál. A paraméterátadás kommunikációját meghatározó információk jönnek létre. Elsődleges és egyedi lesz a formális paraméterlista. Az aktuális paraméterlisták száma az alprogram hívásainak számával egyenlő. Mindig a formálishoz rendeljük az aktuálisat. Beszélhetünk sorrendi vagy név szerinti kötésről.

Sorrendi kötés: Az aktuális paraméterek a felsorolás sorrendjében lesznek hozzárendelve a formális paraméterekhez.

Név szerinti kötés: itt a formális paraméter nevét adjuk meg és mellette az aktuális paramétert, valamelyen szintaktikával. A formális paraméterek sorrendje itt nem számít.

Ha a formális paraméterek száma rögzített, akkor az aktuális paraméterek számának ezzel megegyezőnek kell lennie, VAGY az aktuális paraméterek száma lehet kevesebb is (csak érték szerinti paraméterátadásnál), ekkor amihez nem tartozik aktuális paraméter, majd alapértelmezett módon lesz érték rendelve. Ha a formális paraméterek száma tetszőleges, akkor az aktuális paraméterek száma is tetszőleges lesz. Az aktuális paraméter típusa kovertálható a formális paraméter típusára.

12. Paraméterátadás

Ez az alprogramok és programegységek közötti kommunikáció. Hívó és hívott részből áll. Létrejöhet érték, cím, eredmény, érték-eredmény, név vagy szöveg szerint.

Érték szerinti paraméterátadásnál a formális paraméterek címkomponenssel (hívott oldal), az aktuális paraméterek értékkomponenssel rendelkeznek (hívó oldal). Az alprogram a formális paraméter kezdőértékével dolgozik. Az információ a hívótól a hívott felé áramlik. Az aktuális paraméter kifejezés.

Cím szerinti paraméterátadásnál formális paraméter nem, az aktuális paraméter viszont rendelkezik címkomponenssel. Az alprogram a hívó területen fog dolgozni. Kétirányú információáramlás, az alprogram átvehet és írhat értéket a hívó területre. Az aktuális paraméter változó.

Eredmény szerinti paraméterátadás az aktuális apraméter rendelkezik címkomponenssel, a formális paraméter rendelkezik címkomponenssel a hívott területen. Az alprogram a saját területén dolgozik és nem használja az aktuális paraméter címét. A működés végén átmásolja a formális paraméter értékét a címkomponensére. Kommunikáció iranya a hívottól a hívó felé. Az aktuális paraméter változó.

Érték-eredmény szerinti paraméterátadás esetén a formális paraméternek van címkomponense, az aktuális paraméternek van érték és címkomponense is. Az aktuális paraméter értéke és címe a hívóhoz kerül. Az alprogram saját területén dolgozik, majd az eredmény az aktuális paraméter címére másolódik. Kétirányú kommunikáció. Az aktuális paraméter változó.

Név szerinti paraméterátadásnál az aktuális paraméter egy szimbólumsorozat. A szimbólumsorozat felülírja az alprogram szövegében előforduló formális paraméter minden előfordulását, majd ezután kezdi meg a futást az alprogram.

Szöveg szerinti paraméterátadás annyiban különbözik a név szerintitől, hogy a formális paraméter felülírása a formális paraméter első előfordulása után következik be, futás közben.

Formális paraméterek három csoportja: input paraméter (az alprogram információt kap a hívótól), output paraméter (a hívott információt ad át a hívónak) és input-output paraméter (kétirányú információátadás).

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

Vezérlési szerkezetek

A vezérlési szerkezetek a számítások végrehajtásának sorrendjét határozzák meg. A kifejezések utasítássá válnak, ha pontosvessző követi őket. A kapcsos zárójelek felhasználásával a deklarációkat és utasításokat egyetlen blokkba foghatjuk össze.

Az if-else utasítással döntést, választást írunk le de az else rész nem kötelező. A gép a kifejezés kiértékelése után, ha annak értéke igaz akkor az 1. utasítást, ha értéke hamis, akkor a 2. utasítást hajtja végre. A gép sorban kiértékeli a kifejezéseket. Ha egy kifejezés igaz, akkor a hozzá tartozó utasítást a gép végrehajtja, ezzel a vizsgálat lezárul. Else-if esetén, ha a vezérlés ide kerül, egyetlen korábbi feltétel sem teljesült. Néha ilyenkor semmit sem kell csinálni, így a záró else utasítás elhagyható. A switch utasítás a többirányú programelágaztatásnál használható. Kifejezések értékét hasonlíta össze az állandó értékekkel és ennek megfelelő ugrást hajt végre. A switch kiértékeli a zárójelek közötti kifejezést és összehasonlíta az összes case értékével. A default case-re akkor kerül a vezérlés, ha a többi case egyike sem teljesül. A break utasítás hatására a vezérlés azonnal kilép a switchből. Ugyancsak break utasítással lehet kilépni a while, for és do ciklusokból is. A for (kifejezés1; kifejezés2; kifejezés3) {utasítás;} átférhető while ciklusra. kifejezés1 while (kifejezés2){utasítás kifejezés3;} Végtelen ciklusból return vagy break parancsal lehet kugrani. A harmadik C-beli ciklusfajta, a do-while, a vizsgálatot a ciklus végén, a ciklustörzs végrehajtása után végzi el és a törzs legalább egyszer mindenkorban végrehajtódik. A break utasítással a vizsgálat előtt is ki lehet ugrani a for, while és do ciklusokból, csakúgy, mint a switch-ből. A break utasítás hatására a vezérlés azonnal kilép a legelső zárt ciklusból. A continue utasítás a continue-t tartalmazó ciklus (for, while, do) következő iterációját kezdi meg.

Megoldás videó: <https://youtu.be/zmfT9miB-jY>

10.3. Programozás

[BMECPP]

1. A C nyelv nem objektumorientált újdonságai (1-16. oldal)

A C nyelv továbbfejlesztésének eredménye a C++ nyelv. Vannak olyan változtatások, amik a C nyelv úgymond veszélyes elemeit helyettesítik biztonságosabb módon és vannak olyanok, amik szimplán csak kényelmi finkciós újítások.

Függvényparaméterek és visszatérési értékek

Tetszőleges számú paraméterrel hívható egy C nyelvben, üres paraméterlistával megadott függvény. Viszont C++ nyelven az üres paraméterlista ugyan azt a szerepet tölti be, mintha void paraméterrel adtuk volna meg azt.

2. Az objektumorientáltság alapelvei (17-59. oldal)

Ahogy az informatika fejlődik, úgy bonyolódnak a feladatok. Ezáltal a programkódok bonyolultsága is egyre bonyolultabb az idő elteltével. A 90-es években fejlődött ki az objektum orientált porogramozás, ami nagyban megkönnyítette a programozók életét. Az objektumorientált programozás fő alapelve az egységbe zárás és objektumokból építi fel a programot. Az objektum a valós világ elemeinek programozási modellje, adatokat tárol, és kérésre tevékenységeket végez. Az objektumokat osztályokban tároljuk. Az osztály az egymáshoz hasonló objektumok általánosítása, gyűjtőfogalma. Az objektumosztály határozza meg a hozzá tartozó objektumok típusát.

3. Egységbezárás C++-ban

A C++ egyik fontos tulajdonsága az egységbezárás, ami röviden az adatok és a metódusok osztályba való összezárasát jelenti. Tulajdonképpen az objektum egységbezárja az állapotot (adattagok értékei) a viselkedésmóddal (műveletekkel). Következmény: az objektum állapotát csak a műveletein keresztül módosíthatjuk.

4. Adatrejtés

Előbb említett egységbe zárással sokkal áttekinthetőbbé tettük a programkódot. Ráadásul az egységbezárással képesek vagyunk az adatrejtésre. Alapvetően minden adat elérhető volt minden függvény számára, (azaz publikus volt). Viszont most a struktúrált felépítés lehetővé teszi nekünk, hogy adatot rejtsünk más függvények elől. A struktúra definícióban megadott private kulcsszó lehetővé teszi nekünk, hogy az adott osztályon belül a tagváltozók és függvények ne legyenek elérhetőek kívülről.

Ha egy private osztályon belül lévő tagváltozót felakarunk használni a private osztályon kívül, akkor azt példányosítással tudjuk megtenni. Objektumnak nevezzük az így létrejött osztálypéldányt.

5. Konstruktur

A konstruktur automatikusan hívódik és inicializálja az objektumot. Egy osztályhoz többféle konstruktort is készíthetünk, és ez függ attól hogi hániféle képpen akarjuk inicializálni a példányokat.

6. Destruktorok

A destruktur a konstruktur ellentéte lényegében. Egy konstrukturban lekötött erőforrásokat destruktornal tudunk felszabadítani. Szintén automatikusan hívódik és az objektum felszabadítása előtt hajtódiék végre.

7. Dinamikus adattagot tartalmazó osztályok

A program írásakor nem tudjuk előre mekkora memóriaterület kell majd nekünk. A túl kevés a legrosszabb opción és a túl sok se valami jó megoldás. Ideálisabb, ha dinamikus a memóriakezelés.

A dinamikus memóriakezelés röviden annyit tesz, hogy a felhasznált memóriaterületek foglalását és felszabadítását mi vezéreljük a programból. Mi döntjük el mekkora memóriaterületet foglalunk éppen és, hogy mikor foglaljuk avagy szabadítjuk fel azt. A C nyelvben két függvény, mégpedig malloc() és a free() függvények segítségével tudjuk megvalósítani a dinamikus adatkezelést.

A "malloc" függvény egy általunk megadott méretű memóriaterületet foglal le és visszaadja annak a területnek a címét. A "free" függvényünk pedig a malloc által lefoglalt címet szabadítja fel.

8. Másolókonstruktur

A másoló konstruktur, mint minden konstruktur az osztály nevét kapja. A másoló jellege az jelenti, hogy ugyanolyan típusú objektummal kívánunk inicializálni, ezért a másoló konstruktur argumentuma is ezen osztály referenciajára. A C++ nyelv definíciója szerint a másoló konstruktur akkor fut le, ha inicializálatlan objektumhoz (változóhoz) értéket rendelünk.

9. Friend függvények és osztályok

Szükség lehet rá, hogy egy függvény, bár nem tagja az osztálynak, mégis hozzáérjen az osztály privát mezőihez. Ezt a friend kulcsszó segítségével tehetjük meg: amennyiben az osztály definíciójába beírjuk a függvény fejlécét a friend kulcsszóval kiegészítve, az adott függvény hozzáér az osztály privát mezőihez is. Ha egy függvény "barátja" egy osztálynak, akkor az barátja az osztály leszármazottainak is, de csak az eredeti osztályban definiált privát mezőkhöz fér hozzá, az utód osztályokban definiáltakhoz nem. Egy friend függvény nem csak egy osztálynak lehet barátja, hanem akár többnek is, és nem kell hogy egyszerű függvény legyen, lehet egy másik osztály tagfüggvénye, vagy akár egy másik osztály is lehet.

10. Statikus tagok

Az osztályban statikusan deklarált adattag nem példányosodik. Pontosan egy példány létezik.

11. Operátorok és túlterhelésük (93-96. oldal)

12.

Dinamikus adattagot tartalmazó osztályok

Dinamikus memóriakezelésre használt függvények a `malloc` és `free` függvények. A `malloc` csak a lefoglalni kívánt tárterület méretét ismeri bájtokban. A `new` operátor egy lefoglalt típusra mutató pointert ad vissza. A `delete` operátor a felszabadítani kívánt objektum destruktörét hívja meg. Tömbök lefogalására alkalmas a `new []`, felszabadításukra a `delete []` operátor. A korábban lefoglalt memóriaterületeket mindig fel kell szabadítani, különben elfolyik a memóriánk.

Másolókonstruktur

A másolókonstruktur az osztály típusával megegyező referenciát vár. Az újonnan létrehozott objektumot egy már meglévő objektum alapján hozza létre (initializálja). Ha a másolókonstruktornak érték szerint adunk át egy paramétert akkor az adott változó lemásolódik, felhasználjuk a függvénytörzsben, majd a függvényből való kilépés után felszabadul. ha referenciát adunk át, akkor az átadott érték változhat. A fordító bitenként fogja másolni az objektumot, hacsak nem írunk neki másolókonstruktort. A bitenként másolás a sekély másolás, a dinamikus adattagok másolása a mély másolás.

Friend függvények és osztályok

A `friend` szó használatával egy osztály feljogosít globális függvényeket és más osztályok tagfüggvényeit, hogy hozzáférhessenek a védett tagjaikhoz. `Friend` osztályok esetében az osztály egy másik osztályt jogói fel, hogy hozzáférhessen a védett tagjaihoz. A `friend` tulajdonság nem öröklődik, és nem is tranzitív. Az inicializálás egy újonnan létrehozott adatszerkezet kezdőértékének beállítását jelenti. Míg az értékadás egy már meglévő adatszerkezetnek ad új értéket.

A konstruktor inicializálási listájába lehet a tagváltozókat inicializálni. Konstruktor (argumentumlista)

Statikus tagok

Osztályok esetében definiálhatunk statikus tagváltozókat, amik az adott osztályhoz (nem az osztály objektumaihoz) tartoznak. Ezeket osztályváltozóknak is nevezhetjük, mert az osztály minden objektumára vonatkozó közös értéket vettek fel. Deklarációjuk az osztályban a `static` kulcsszóval történik, de a statikus tagváltozót ugyanekkor definiálni is kell az osztályon kívül. Ez a hatókör operátorral (`::`) történhet meg, hogy tudjuk melyik osztályhoz tartozik az adott statikus változó. Lehetőség van statikus tagfüggvények definiálására is. A `static` kulcsszót kell megadni az adott tagfüggvényre vonatkozóan. Statikus tagfüggvényekből a nem statikus tagfüggvények és tagváltozók nem érhetők el. A `this` mutató nem értelmezhető statikus függvények törzsében. Akkor érdemes statikus tagváltozókat használni, ha olyan változóra van szükség, amely az osztály minden változójára közös. A statikus tagváltozók minden a globális változókkal együtt inicializálódnak, a `main` függvénybe lépés előtt.

A beágyazott definíciók: az enumeráció, osztály-, struktúra- és típusdefiníciók osztálydefinícióján belüli megadása. A privát részben található definíciók csak az őt tartalmazó osztály tagfüggvényei számára érhetők el, a publikus részben találhatóak mindenki számára elérhetők. A beágyazott osztály nem biztosít speciális jogokat az őt tartalmazó osztálynak és ez visszafelé is ugyanígy igaz.

Operátorok és túlterhelésük (93-96. oldal)

Az operátorok (+, -, *, /, %, stb.) argumentumokkal végeznek műveleteket, az így kapott eredmények a visszatérési értékek. Egy operátor mellékhatásának nevezzük, ha az operátor megváltoztatja az argumentum értékét. Az operátorkiértékelések sorrendjét (erősségét) egy precedenciáblázat tartalmazza. Az operátor túlterhelés saját operátorok definiálását jelenti. A függvények és operátorok közötti különbség csupán a kiértékelési szabályrendszer. Az operátorok tulajdonképpen speciális nevű függvények és kiértékelésük speciális szabályrendszert vesz alapul. célunk, hogy az általunk definiált típusokra is megadhassunk operátor működéseket. Így az argumentumtípusok közül legalább az egyik nem beépített típus lesz. Az operátort tagfüggvényként érdemes definiálni, ha első paramétere olyan típus, amit mi írnunk.

III. rész

Második felvonás

**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.

11. fejezet

Helló, Berners-Lee!

11.1. C++ és Java nyelvek összehasonlítása

Felhasznált könyvek a két nyelv összehasonlításához:

- C++: Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven
- Java: Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II.

A Java nyelv nagyon hasonlít a C és a C++ nyelvekhez, hiszen szintaxisa ezen nyelvekből származtatható. Egy alapvető különbség a Java és a C++ nyelvek között már a fordítás során előjön. Míg a C++ egy natív nyelv (azaz fordítás sorás egy az egybe gépi kódot állít elő), addig a Java egy interpreteres nyelv, azaz a fordítás során egy bajtkódot (Java bytecode) állít elő. Ez a Java bytecode a Java Virtual Machine számára értelmezhető kód. Bármilyen rendszeren amin rajta van a JVM, az előbb említett bajtkód értelmezhető lesz. Egyszerre előnye és hátránya is ez a Java-nak. Mivel a bajtkód előállítása majd annak tovább fordítása időigényes, ezért ez nagy hátrány. Viszont ha azt nézzük, hogy a JVM által egy adott Java kódot bármilyen rendszeren futtathatjuk nagy előnyre teszünk szert. Hiszen egy blackPanther rendszeren megírt C++ programot nem biztos, hogy tudunk futtatni SUSE operációs rendszer alatt.

Szintaxis:

A Java és a C++ nyelv szintaxisa lényegében megegyezik. Viszont

Kifejezések:

Már tudjuk, hogy a C/C++ nyelveknél a részkifejezések kiértékelési sorrendjére semmilyen szabály nem vonatkozik, magyarul: nincs megszabva melyik fog eloször, melyik következ "ore kiértékel" odni. Ez már Java-ban nem így van, Java a kiértékelési sorrend balról jobbra történik.

Típuskonverziók

Mint a C++ esetén, a Javában is van automatikus/kézi típuskonverzió, sőt a Java annyira típusos nyelv, hogy minden kifejezésben megvizsgálja a hatósági program, hogy a kifejezésben található típusok "összeférne-ke" egymással. Például jelez, ha lebegopontos típusú változót szeretnénk egészre konvertálni, ekkor ugye adtavesztés megy végbe.

Objektumok

A Java programozási nyelv alapvető eleme az objektum. Az ilyen nyelveket objektum orientált (OO) programozási nyelveknek nevezzük. Az objektum az adott feladat szempontjából fontos, a valódi világ valamelyen elemének a rá jellemző tulajdonságai és viselkedései által modellezett eleme. Az objektumokkal kapcsolatban valamilyen feladatokat szeretnénk megoldani. A nyelv tervezésekor fontos szempont volt az, hogy az objektumok többé-kevésbé állandóak, de a hozzájuk tartozó feladatok nem, ezért az objektumok kapnak nagyobb hangsúlyt. A mai programok nagyon sok, egymással kölcsönhatásban álló elemből állnak, így nem is igazán programokról, hanem programrendszerkről beszélhetünk.

11.2. Python

A Python egy könnyen tanulható, de hatékony programozási nyelv. Magasszintű adatstruktúrái, az objektumorientáltság egyszerű megközelítése, elegáns szintaxisa, dinamikus típusossága és interpreteres mivolta ideális script-nyelvvé teszi. Kiválóan alkalmas gyors fejlesztői munkákra, nagyobb projektek összefogására. Tulajdonképpen a Python nem a kifejezés szigorú értelmében vett interpreter: a Python byte-kódot fordít és azt futtatja.

A Python nyelv szintaxisa jelentősen egyszerűsített, mivel behúzás alapú. Tehát nincs szükség kapcsoszárájára a blokk elejének és végének jelöléséhez. A blokk addig tart ameddig az értelmező nem talál egy kisebb behúzású sort. Érdekesség még, hogy az utasításokat nem kell ";"-vel elválasztani, azok végét a sor vége jelzi. Persze előfordulhat, hogy egy utasítás nem fér ki egy sorba, ezt az értelmezőnek egy a sor végére írt "\"-jellel jelölhetjük. Ha nem zárunk be egy zárójelet, akkor is a következő sort az utasítás folytatásának veszi. A szintaxisból adódóan nagyon gyors fordítót lehet írni, ezért "néz ki úgy", mintha interpretált lenne a nyelv.

Nagy előnye a nyelvnek, hogy nincs szükség a fordítási fázisra hiszen a Python forrás önmagában elegendő az értelmező számára. Emellett az interpreter könnyen bővíthető C,C++ vagy más C-ből hívható nyelven megírt függvényekkel és adattípusokkal. Segítségével rövid programok írása gyors, egyszerű és jól áttekinthető. Feltűnően gyorsabb programokat hozhatunk létre Python nyelven, mint például C/C++ nyelven.

Python nyelvben nincs szükség explicit módon meghatározni a változók típusát. Futási idő alatt meghatározza minden változó típusát. Megkülönböztet számokat, melyek lehetnek egészek, lebegopontosak, illetve komplexek is, egészek esetén decimális, oktális és hexadecimális szármrendszert is ismer; sztringeket, amit megadhatunk idézójelek vagy aposztrófok között is; enneseket, ami objektumok gyűjteményét jelenti; listákat, ami akár különbözo típusú elemeket is tartalmazhat és szótárakat, ami kullccsal ellátott rendezetlen halmazt jelent. Változók alatt minden az adott objektumra mutató referenciát értjük. A különböző típusok közötti konverzió támogatott. Két fajtáját különböztetném meg a Python beépített típusainak. Ezek a primitívek és az objektumok. A primitív típusok egy értéket jelölnek. Ezek például a logikai értékek és a számok. Az objektumok összetett adattípusok, amik általában más, primitív típusokat kombinálnak. Primitívek: bool, int, float, none. Objektumok: str, list, dict. Változó létrehozásához a változó neve, egy egyenlőség operátor és a változó értéke kell csupán.

A Python biztosítja a hagyományos programnyelvi eszközöket. Tudunk kírni szövegeket, változók értékét a konzolra. Létrehozhatunk if elágazásokat. Ebben eltérés, hogy az else if ágat elif-el jelöljük. A nyelv támogatja a ciklusokat, tehát a while, for ciklusok gond nélkül létrehozhatók. Egyedül a szintaktikájukban térnek el a C/C++-os társaikétől. A for ciklusok esetén érdemes megemlíteni az xrange függvényt, mellyel megadhatjuk, hogy mennyi meddig szeretnénk futtani a ciklusváltozót, és azt is, hogy milyen lépésközzel. Készíthetünk címkeket is, melyet a label kulcszsal jelölünk, és a goto utasítással tudjuk a vezérlést átugratni rá. Létezik egy másik kulcszsó is, a comefrom. Ennek a segítségével a címkeből vissza lehet ugrani a hozzá tartotó comefrom részhez.

12. fejezet

Helló, Arroway!

12.1. Az objektumorientált paradigma alapfoglalmai. Osztály, objektum, példányosítás.

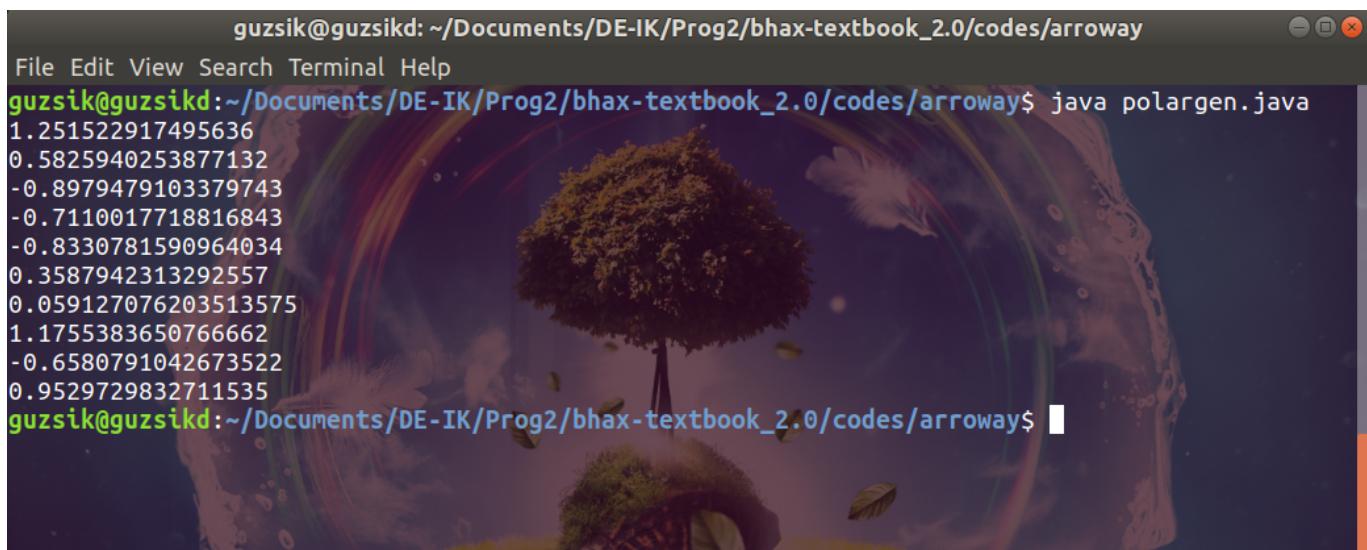
Az osztály az objektum-orientált programozás egyik központi fogalma. Az adattartalmukban esetleg eltérő, de viselkedés szempontjából megegyező objektumokat tekintjük egy osztályba tartozóknak. Az objektumorientált programozási nyelvek rendszerint jelzik is az egyes osztályokat. Ennek kulcsszava a legtöbb esetben a class. Az objektum az objektumorientált programozás egyik alapeleme. Az objektumokat általában információt hordozó, és azokkal műveleteket vagy számításokat elvégezni képes egységeként fogjuk fel. A programozási feladat terében egy elemnek, vagy összetevőnek felel meg, míg számítógépes reprezentációja egy összefüggő adatterületként, és a területet kezelő alprogramok halmazaként képzelhető el. Példányosítás pedig röviden egy adott sablonnal meghatározott adatszerkezet műveletvégzésre alkalmas példányba történő lemásolása, létrehozása.

12.2. OO Szemlélet

Ebben a részben a polártranszformációs normális generátort kellett megírni Java nyelven. Lényege, hogy egy polármódszerrel előállított számot ad vissza. Ennek a matematika háttere nem érdekes, viszont a Java program része már annál inkább. A program objektumorientált szemléletmóddal készült vagyis van benne osztály, attribútumok és metódusok. Lényege, hogy a PolarGen osztály boolean nincsTarolt attribútuma határozza meg a kovetkezo() függvény double visszatérési értékét. Ha nincs tárolt értékünk akkor két számot állítunk elő. Az egyiket eltároljuk egy változóban ebből következik, hogy a nincsTarol értéke módusul majd a másikkal visszatérünk. Ha viszont van tárolt érték akkor egyszerűen azzal tér vissza a kovetkezo() függvény. A kód:

```
import static java.lang.System.*;
public class PolarGen
{
    boolean nincsTarolt = true;
    double tarolt;
    public PolarGen()
    {
```

```
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)
{
    PolarGen pg = new PolarGen();
    for (int i = 0; i < 10; i++)
    {
        out.println(pg.kovetkezo());
    }
}
```



```
guzsik@guzsikd: ~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/arroway
File Edit View Search Terminal Help
guzsik@guzsikd: ~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/arroway$ java polargen.java
1.251522917495636
0.5825940253877132
-0.8979479103379743
-0.7110017718816843
-0.8330781590964034
0.3587942313292557
0.059127076203513575
1.1755383650766662
-0.6580791042673522
0.9529729832711535
guzsik@guzsikd: ~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/arroway$
```

12.1. ábra. PolarGen program eredménye

Hasonló megoldás található az OpenJDK-ban a Random.class-ban. Különbség a synchronized kulcsszó. Lényege, hogy egy időben csak 1 szálon futhat a programkód ezzel megakadályozva a különöző szálakon történő szimultán módosításokat.

12.3. Homokozó

Ebben a feladatban a C++ féle LZWBinfa-t kellett átírni Java nyelvre. A legtöbb függvényt könnyű volt átírni csak a pointereket és referenciákat kellett kiiírtani a kódból. Lényegi rész volt, hogy a működése ne változzon. A működést nem részletezném, mert már prog1-ről is rengetegszer feljött úgyhogy mindenki a "könyökén" jön ki. Java forráskód a Binfáról:

```
package lzwfa;
import java.io.FileInputStream;
public class Binfa
{
    public Binfa()
    {
        fa = gyoker;
    }

    public void egyBitFeldolg(char b)
    {
        if(b == '0')
        {

            if(fa.egyesGyermek() == null)
            {
                Csomopont uj = new Csomopont('0');
                fa.egyesGyermek() = uj;
            }
            else
                fa.egyesGyermek().egyBitFeldolg(b);
        }
        else
            fa.mezesGyermek().egyBitFeldolg(b);
    }
}
```

```
        fa.ujNullasGyermek(uj);
        fa = gyoker;
    }
    else
    {
        fa = fa.nullasGyermek();
    }
}
else
{
    if(fa.egyesGyermek() == null)
    {
        Csomopont uj = new Csomopont('1');
        fa.ujEgyesGyermek(uj);
        fa = gyoker;
    }
    else
    {
        fa = fa.egyesGyermek();
    }
}
}

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());  
        os.print("(");  
        os.print(melyseg - 1);  
        os.println(")");  
        kiir(elem.nullasGyermek(), os);  
        --melyseg;  
    }  
}  
  
protected Csomopont gyoker = new Csomopont('/');
```

```
int maxMelyseg;
double atlag, szoras;

public int getMelyseg()
{
    melyseg = maxMelyseg = 0;
    rmelyseg(gyoker);
    return maxMelyseg - 1;
}

public void rmelyseg(Csomopont elem)
{
    if(elem != null)
    {
        ++melyseg;
        if(melyseg > maxMelyseg)
        {
            maxMelyseg = melyseg;
        }
        rmelyseg(elem.egyesGyermekek());
        rmelyseg(elem.nullasGyermekek());
        --melyseg;
    }
}

public double getAtlag()
{
    melyseg = atlagosszeg = atlagdb = 0;
    ratlag(gyoker);
    atlag = ((double) atlagosszeg) / atlagdb;
    return atlag;
}

public double getSzoras()
{
    atlag = getAtlag();
    szorasosszeg = 0.0;
    melyseg = atlagdb = 0;

    rszoras(gyoker);

    if(atlagdb - 1 > 0)
    {
        szoras = Math.sqrt(szorasosszeg / (atlagdb - 1));
    }
    else
    {
        szoras = Math.sqrt(szorasosszeg);
    }
}
```

```
    return szoras;
}

public void ratlag(Csomopont elem)
{
    if(elem != null)
    {
        ++melyseg;
        ratlag(elem.egyesGyermek());
        ratlag(elem.nullasGyermek());
        --melyseg;
        if(elem.egyesGyermek() == null && elem.nullasGyermek() == null)
        {
            ++atlagdb;
            atlagosszeg += melyseg;
        }
    }
}

public void rszoras(Csomopont elem)
{
    if(elem != null)
    {
        ++melyseg;
        rszoras(elem.egyesGyermek());
        rszoras(elem.nullasGyermek());
        --melyseg;
        if(elem.egyesGyermek() == null && elem.nullasGyermek() == null)
        {
            ++atlagdb;
            szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));
        }
    }
}

public static void usage()
{
    System.out.println("Usage: lzwtree in_file -o out_file");
}

public static void main(String[] args)
{
    if(args.length != 3)
    {
        usage();
        System.exit(-1);
    }

    String inFile = args[0];
```

```
if(!"-o".equals(args[1]))
{
    usage();
    System.exit(-1);
}

try
{
    java.io.FileInputStream beFile = new java.io.FileInputStream(new java.io.File(args[0]));

    java.io.PrintWriter kiFile = new java.io.PrintWriter( new java.io.BufferedWriter( new java.io.FileWriter(args[2])));

    byte[] b = new byte[1];

    Binfa binFa = new Binfa();

    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.egyBitFeldolg('1');
        }
        else
        {
            binFa.egyBitFeldolg('0');
        }
        b[0] <<= 1;
    }

    binFa.kiir(kiFile);

    kiFile.println("depth = " + binFa.getMelyseg());
    kiFile.println("mean = " + binFa.getAtlag());
    kiFile.println("var = " + binFa.getSzoras());

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

} catch (Exception e) {
    System.err.print(e.getMessage());
}
}

}
```

A feladat második része az volt, hogy ezt a Binfa-t emeljük át egy Java Servletbe és irassuk ki a fát egy böngészőbe. Ahhoz, hogy működőképes legyen a Servlet szükség volt egy Tomcat Serverre. Letöltés után integrálni kellett a JRE-hez, amit az Eclipse telepítési könyvtárában találtam. A megfelelő konfiguráció után a 8085-ös porton elértem a Tomcat szervert. Következő lépésként indítottam egy Dinamikus Web Projektet, amit belül csináltam egy LZWBinfaServlet.java servlet file-t. Ezen fájl automatikusan generálta a szükséges metódusokat, konstruktorkat a megfelelő működéshez. A classom a HttpServlet osztályból lett származtatva, illetve kaptunk egy doGet (...) eljárást. Ez az eljárás felel a HTTP GET kéréséért, ami nekünk pont kapóra jött.

Az LZWBinfa forráskódjából minden átírtunk a main függvény kivételével. A main kódjait a doGet(...) eljáráson belül írjuk meg. A kész doGet(...) így néz ki:

```
**
 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse)
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
```

```
{  
    String parameter = request.getParameter("param");  
  
    byte[] b = parameter.getBytes();  
  
    Binfa binFa = new Binfa();  
  
    java.io.PrintWriter kiFile = new java.io.PrintWriter( new java.io. ↵  
        BufferedWriter( new java.io.FileWriter("output.txt")));  
  
    boolean kommentben = false;  
  
    for(int h = 0; h < b.length; ++h)  
    {  
        if(b[h] == 0x3e)  
        {  
            kommentben = true;  
            continue;  
        }  
  
        if(b[h] == 0x0a)  
        {  
            kommentben = false;  
            continue;  
        }  
  
        if(kommentben)  
        {  
            continue;  
        }  
  
        if(b[h] == 0x4e)  
        {  
            continue;  
        }  
  
        for (int i = 0; i < 8; i++)  
        {  
            if((b[h] & 0x80) != 0)  
            {  
                binFa.egyBitFeldolg('1');  
            }  
            else  
            {  
                binFa.egyBitFeldolg('0');  
            }  
            b[h] <<= 1;  
        }  
    }  
}
```

```
binFa.kiir(kiFile);

kiFile.println("depth = " + binFa.getMelyseg());
kiFile.println("mean = " + binFa.getAtlag());
kiFile.println("var = " + binFa.getSzoras());

kiFile.close();

File file = new File("output.txt");
FileInputStream fis = new FileInputStream(file);

// TODO Auto-generated method stub
response.setContentType("text/html");

PrintWriter out = response.getWriter();
out.println("<!DOCTYPE html>");
out.println("<html>");

out.println("<head><title>LZWBinfa</title></head>");
out.println("<body>");
out.println("<h2>Dékány Róbert LZWBinfa Servlet</h2>");
try(BufferedReader br = new BufferedReader(new InputStreamReader(fis))) {
    for(String line; (line = br.readLine()) != null; ) { out.println(
        "<p>" + line + "</p>"); }
}

out.println("</body></html>");
}
```

A String parameter = request.getParameter("param"); változóban lesz eltárolva az URI-ból vett param nevű paraméter értéke. Ezt a karakterláncot fogjuk továbbadni a Binfának. A PrintWriter out = response.getWriter(); egy olyan objektummal tér vissza, ami képes kiírni az outputra jelen esetbe HTML kódokat. A továbbiakban csak kiiíratjuk a Stream tartalmát és elhelyezzük a szükséges HTML-t.

12.4. „Gagyí”

Az ismert formális "while (x <= t && x >= t && t != x);" tesztkérdéstípusra adj a szokásosnál (miszerint x, t az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciája) „mélyebb” választ, írj 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!

Megoldás forrása: [gagyi.java](#)

A fent említett tesztkérdés "szokásosan" Java tesztkérdésként szokott előjönni az univerzumban, hiszen egy nagyon alap és köztudott működésére kérdez rá. Mikor egyenlő két objektum?

Kétfajta egyenlőséget különböztetünk meg, az egyik a "érték szerinti" egyenlőség, a másik az "identitáson alapuló" egyenlőség.

Az "érték-szerinti" egyenlőség pontosan azt jelenti amit elsőre is gondolnánk, két érték egyenlő, valamik által horodozott értékek megegyeznek.

```
import java.lang.Number;
class Gagyil {
    public static void main (String[] args) {
        int x = 128;
        int t = 128;
        while (x <= t && x >= t && t != x);
        System.out.println("infinity");
    }
}
```

A while ciklusunk mindenkor feltétele igaz lesz, így végtelen ciklusba lépünk be. Első kettő feltétel egy-értelmű mivel az x és a t értékeit hasonlítja össze, míg a harmadik feltétel a két változő helyét.

```
import java.lang.Number;
class Gagyil {
    public static void main (String[] args) {
        int x = 127;
        int t = 127;
        while (x <= t && x >= t && t != x);
        System.out.println("infinity");
    }
}
```

Itt viszont már nem lépünk végtelen ciklusba, mert a harmadik feltételünk hamis lesz. Miért? Azért mert az Integer osztály gyorsítótáraz bizonyos értéktartományon belül lévő értékeket. Az említett értéktartomány: [-128;127]. Így minden olyan esetben amikor ebből a tartományból választunk literált, vagy a valueOf metódust meghívva, mi minden, egy már létező objektumra kapunk egy-egy referenciát, tehát megbukik a x != t feltétel.

12.5. Yoda

Írunk olyan Java programot, ami java.lang.NullPointerException-leáll, ha nem követjük a Yoda conditions-t! https://en.wikipedia.org/wiki/Yoda_conditions

Megoldás forrása: [yoda.java](#)

A Yoda Conditions lényege, hogy ellentétben a hagyományos összehasonlítással - ahol a változót hasonlítjuk az adott értékhez - felcserélve történik a művelet. Tehát adott a következő állítás:

```
if( értékVáltozó == 10) { /*...*/ }
```

Ez a hagyományos eljárás. Ugyanez Yoda Conditions környezetben:

```
if( 10 == értékVáltozó) { /*...*/ }
```

Mindkét esetben hasonló eredményeket kapunk. Azonban illik vigyázni, hiszen a legtöbb nyelvben az összehasonlító operátor a dupla egyenlőségjel ($==$), amit sokszor összekevernek az értékadó operátorral, ami az egyedüli egyenlőségjel ($=$).

A feladat szerint egy olyan Java programoz kell írnunk, ami leáll `java.lang.NullPointerException` hibával, amennyiben nem követjük az utóbbi feltételt:

```
public class Yoda {
    public static void main(String[] args) {
        String text = null;
        if ("Text".equals(text))
            System.out.println("Text".equals(text));
        if (text.equals("Text"))
            System.out.println("Text".equals(text));
    }
}
```

A fenti programot lefuttatva a 6. sorban, tehát a második if-nél hibát dob ki, pontosan azt, amit akartunk. Ha egy adott értéket, jelen esetben a Text stringet hasonlíttuk a text változónhoz, nem kapunk kivételt. Viszont ha fordítva csináljuk, és a text változónkat, ami jelen esetben null "értékkel" rendelkezik hasonlíttuk a Text értékhez, hibát kapunk, mivel ha valaminek van értéke, az már alapból nem lehet null.

12.6. Kódolás from scratch

Induljunk ki ebből a tudományos közleményből: <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/bbpalg.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-tanitokjavat/apbs02.html#pi_jegyei (mert ha csak lemásolod, akkor pont az a fejlesztői élmény marad ki, melyet szeretném, ha átélnél).

Megoldás forrása: [pibbp.java](#)

A következő program kiszámolja nekünk, hogy mennyi a pi értéke hexadecimális kifejtésben, amely azért jó, mert képesek vagyunk úgy meghatározni egy helyről a pi értékeit, hogy tudnánk az előtte levő számokat. Ez az eljárás egyáltalán nem újkeletű: 1995-ben talált Bailey-Borwein-Plouffe féle algoritmust alkalmazva könnyen meghatározható.

```
public class PiBBP {
    public PiBBP(int d) {
        String d16PiHexaJegyek;
        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();
}

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));
    }
}
```

A programunk visszaadja a pi értéket a d+1 helyről számítva, azaz a jelen esetben a 1000001. helyről, amely egész pontosan a 6C65E5308. A C nyelv long double változóit használva sokkal pontosabb értékeket is képesek lehetünk meghatározni. De a Java PiBBP osztály segítségével előről is kezdhetjük számolni a pi értékét, ha a main függvényt a következőre módosítjuk:

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

    for(int i=0; i<3000; i+=1) {
        PiBBP piBBP = new PiBBP(i);
        System.out.print(piBBP.toString().charAt(0));
    }
}
```

Ezt alkalmazva megkapjuk a pi első 3000 jegyét hexadecimális értékben.

13. fejezet

Helló, Liskov!

13.1. Liskov helyettesítés sértése

Írunk olyan OO, leforduló Java és C++ kódcsipetet, amely megséríti a Liskov elvet! Mutassunk rá a megoldásra: jobb OO tervezés.

https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf (93-99 fólia) (számos példa szerepel az elv megsértésére az UDPROG repóban, lásd pl. source/binom/Batfai- Barki/madarak/)

[liskh.cpp](#)

[liskovsertve.cpp](#)

[liskh.java](#)

[liskovsertve.java](#)

Liskov helyettesítési elv: minden osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a program helyes működése megváltozna.

Ha S altípusa T-nek, akkor egy T típusú objektum helyettesíthető S típussal anélkül, hogy megváltozna a program helyessége. Magyarul egy osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a szoftver helyes működése megváltozna.

Nem minden rovar bogár, de minden bogár rovar. Ez egy elég ismert közmondás, ami pont téma illik. Kicsit elvonatkoztatnák a rovaroktól és egy egyszerűbb példát hoznék fel.

Konkrét példának vizsgáljuk meg a *Madarak* objektumorientáltságát! Ha meghalljuk a madár szót, akkor egyből egy tollas, repülő állatra gondolunk ami a fejünk felett siklik az égen. Viszont nem minden madár ilyen. Ha a madaraknak alap tulajdonságnak adjuk meg a repülést, akkor gondban leszünk ha ezt a tulajdonságot tovább adjuk az összes madár típusnak. Például ott van a pingvin. Nem repül, mégis madár.

```
class Bird
{
    public void fly()
    {
        System.out.println("Am flying...\n");
    }
}
```

```
class Eagle extends Bird
{
    public void fly()
    {
        System.out.println("Eagle: flying..\n");
    }
}

class Penguin extends Bird
{
}

class Liskov
{

    public static void flyBird(Bird b)
    {
        b.fly();
    }

    public static void main(String[] args)
    {
        Bird theEagle = new Eagle();
        Bird thePenguin = new Penguin();

        flyBird(theEagle);
        flyBird(thePenguin);

    }
}
```

Fenti kódsorunkban gondban vagyunk, hiszen pingvinünk elrepült melegebb éghajlatokra. A hiba a fenti "*tervezésben*" az, hogy mi minden madárról azt feltételeztük, hogy mindegyik képes repülni, így a `fly` funkcionalitást mindegyik örökli. Mi lehet a megoldás?

Megoldás lehet, hogy a repülés tulajdonságot absztraktáljuk. Azaz:

Kivesszük az Őstől a `fly` metódust és bevezetünk két új interfészként szolgáló osztályt. Az egyik `IFlyingBird` lesz, ami deklarálja a virtuális `fly` metódust, másik a `INotFlyingBird` lesz, végül lecseréljük a `flyBird` statikus paraméterének típusát `IFlyingBird`re

```
interface Bird
{

}

interface IFlyingBird extends Bird
```

```
{  
    public void fly();  
}  
interface INotFlyingBird extends Bird  
{  
}  
class Eagle implements IFlyingBird  
{  
    public void fly()  
    {  
        System.out.println("Eagle: flying..\n");  
    }  
}  
class Penguin implements INotFlyingBird  
{  
}  
  
class Liskov  
{  
  
    public static void flyBird(IFlyingBird b)  
    {  
        b.fly();  
    }  
  
    public static void main(String[] args)  
    {  
        Eagle theEagle = new Eagle();  
        Penguin thePenguin = new Penguin();  
  
        flyBird(theEagle);  
        flyBird(thePenguin); //Fordítási hiba!!  
  
    }  
}
```

C nyelven hasonlóképpen működik.

13.2. Szülő-gyerek

Írunk Szülő-gyerek Java és C++ osztálydefiníciót, amelyben demonstrálni tudjuk, hogy az ősön keresztül csak az ős üzenetei küldhetőek!

https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf (98. fólia)

PiBBPBench.java

PiBBPBench.c

PiBBPBench.cs

Ez a feladat csupán azt demonstrálná, hogy nem lehetséges egy adott szülő referencián keresztül, ami egy gyerek objektumára hivatkozik, meghívni gyermeke egy olyan metódusát amit ő maga nem definiált.

C++-ban, Javaban is, ezt polimorfizmussal tudjuk kimutatni, eleve polimorfizmusról beszélünk ha egy szülő mutató vagy referencia egy gyerekére mutat/hivatkozik.

Chapterly Reminder

Polimorfizmus alatt ebben a kontextusban azt értjük, hogy több különböző típusú objektumhoz hozzá tudunk férni egy közös interfészen(Ősön) keresztül, és az Ős típusú változón keresztül meghívhatjuk az Őssel közös metódusokat rajtuk.

A nem Ősök által definiált metódusokhoz nem férhetünk hozzá, ha csak nem *downcastoljuk* az adott objektumot a tényleges típusára. Ez esetben viszont megsértjük az előző feladatban ismertetett Liskov-elvet.

```
#include <iostream>
#include <string>

class Parent
{
public:
    void saySomething()
    {
        std::cout << "Parent says: BLA BLA BLA\n";
    }
};

class Child : public Parent
{
public:
    void echoSomething(std::string msg)
    {
        std::cout << msg << "\n";
    }
};

class App
{
int main()
{
    Parent* p = new Parent();
    Parent* p2 = new Child();

    std::cout << "Invoking method of parent\n";
    p->saySomething();
```

```
    std::cout << "Invoking method of child through parent ref\n";
    p2->echoSomething("This won't work");

    delete p;
    delete p2;

}
};
```

Javaban ugyanígy:

```
class Parent
{
    public void saySomething()
    {
        System.out.println("Parent says: BLA BLA BLA");
    }
}

class Child extends Parent
{
    public void echoSomething(String msg)
    {
        System.out.println(msg);
    }
}

public class App
{
    public static void main(String[] args)
    {
        Parent p = new Parent();
        Parent p2 = new Child();

        System.out.println("Invoking method of parent");
        p.saySomething();

        System.out.println("Invoking method of child through parent ref");
        p2.echoSomething("This won't work");
    }
}
```

13.3. 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!

<https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apas03.html#id561066>

- [PiBBPBench.java](#)
- [PiBBPBench.c](#)
- [PiBBPBench.cs](#)

A BBP programunkat átírtuk objektum orientáltság szempontból. Mivel a C nyelv nem OOP, így fontos hogy egyik nyelven megírt BBP sem lehet OOP, hogy össze tudjuk hasonlítani az összes nyelvet.

A futási idők a következőképpen alakultak:

```
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ javac PiBBPBench.java
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ java PiBBPBench
6
1.352
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ javac PiBBPBench.java
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ java PiBBPBench
7
15.787
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ javac PiBBPBench.java
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ java PiBBPBench
12
193.77
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$
```

Java nyelven

```
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ g++ PiBBPBench.c -o PiBBPBench
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ ./PiBBPBench
6
1.514400
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ g++ PiBBPBench.c -o PiBBPBench
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ ./PiBBPBench
7
17.746486
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ g++ PiBBPBench.c -o PiBBPBench
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ ./PiBBPBench
12
218.402788
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$
```

C nyelven

```
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ 
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ mcs -out:PiBBPBench.exe PiBBPBench.cs
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ mono PiBBPBench.exe
6
1.399068
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ mcs -out:PiBBPBench.exe PiBBPBench.cs
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ mono PiBBPBench.exe
7
16.095648
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ mcs -out:PiBBPBench.exe PiBBPBench.cs
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$ mono PiBBPBench.exe
12
196.745677
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/liskov$
```

C sharp nyelven

13.4. 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 (77-79 fóliát)!

A ciklomatikus komplexitás egy metrika ami, mint minden ehhez a szakmához tartozó metrikai eszköz, az adott dolog, ez esetben metódusok, minőségét hivatott jellemzni.

A bonyolultság kiszámításához külső programokat veszünk igénybe, vagy ha valamilyen *build automation tool*-t használ a projectünk akkor a megfelelő pluginet használva végezzük el a méréseket.

Egy ilyen mérést el is végzünk a már megírt LZWBinaryTree.java programunkra utóbbi módszer segítségével. Ehhez telepítenünk kell az adott Linux disztrónkon a maven csomagot, JDK megléte természetesen szükséges.

Telepítés után elmegyünk oda ahova létre szeretnénk hozni a projektünket, majd létrehozunk egy alap maven projectet a következő parancs kiadásával:

```
mvn archetype:generate \
    -DgroupId=com.monolith \
    -DartifactId=LZW \
    -DinteractiveMode=false
```

A parancs kiadása után letölti automatikusan a *Central Repository*-ból a szükséges csomagokat, kialakítja az alapértelmezett projectet(létrehozva a könyvtárstruktúrát) és egy App.java fájlt ad nekünk a legfontosabb pom.xml mellett ahol be tudjuk állítani projectünk függőségeit/pluginjait/goal-jait stb.. A pom.xml fájlon keresztül kommunikálunk a build systemmel.

A mellékelt Java állományt töröljük, majd ide átmásoljuk az LZW-s programunkat.

Ahhoz, hogy a mérést eltudjuk végezni a **JavaNCSS** plugin fogjuk használni.

Maven

 Nem kell nekünk semmilyen .jar fájlt letölteni ehhez, a maven automatikus letölt és konfigurál(pom.xml alapján) minden megadott függőséget és pluginet, nekünk csupán azt kell megmondani a pom.xml fájlból, hogy mi ezt és ezt akarjuk használni. [Mi minden a Maven?](#)

A [plugin weboldala](#) alapján annak érdekében, hogy működjön a plugin a következőket kell elvégeznünk:

Ez egy reporter plugin, tehát, hogy használni tudjuk reporting tag hierarchiában kell lennie, ezt nekünk nem kell megírnunk külön, hisz az oldalon megadták nekünk a beillesztendő szöveget:

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>javancss-maven-plugin</artifactId>
        <version>2.1</version>
```

```
</plugin>
</plugins>
</reporting>
...
</project>
```

Ekkor a `mvn site` parancs kiadatására elkészülnek a **reportok(!)**. A report goal-k közül vannak olyanok is amik megőlik a sikeres buildet, ha valamelyik metrika elér egy értéket, ráadásul nekünk csak egyetlen egy goal végrehajtása fontos, így megmondjuk `pom.xml`-ben, hogy csak azt az egy goalt tessék futtatni.

```
<project>
...
<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>
```

A report goal egy weblapot fog elkészíteni nekünk, ami tartalmazni fogja többek között az egyes metodusok CNN értékét is. Egyszerűbbség kedvéért mi egyenesen a `pom.xml`-be fogalmaztuk meg a végrehajtandó goalt, így a `mvn package site` parancs kiadásakor végrehajtónak a pluginok goaljai. Ha közvetlenül akarnánk a goal-t meghívni, pl bash alatt, akkor a következő parancsot kell kiadnunk: `mvn javancss:report`.

Az eredmény egy `javancss.html` fájl lesz a `./target/site` mappában. Ha megnyitjuk láthatjuk a mérés eredményeit, köztük a CNN értéket.

A `pom.xml` tartalma:

```
<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.monolith</groupId>
<artifactId>LZW</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>LZW</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>
<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>
```

14. fejezet

Helló, Mandelbrot!

14.1. Reverse engineering UML osztálydiagram

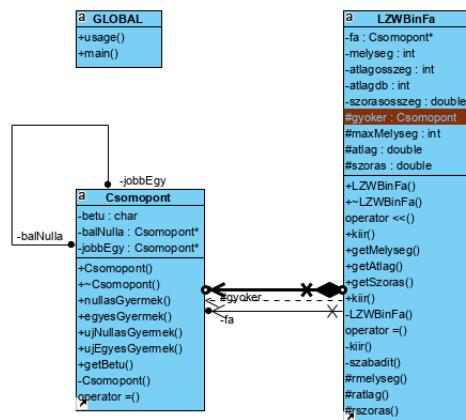
UML osztálydiagram rajzolása az első védési C++ programhoz. Az osztálydiagramot a forrásokból generáljuk (pl. Argo UML, Umbrello, Eclipse UML). Mutassunk rá a kompozíció és aggregáció kapcsolatára a forráskódban és a diagramon, lásd még: https://youtu.be/Td_nlERIEOs.

Megoldás forrása: [Prog1_6.pdf](#) (28-32 fólia)

Az UML egyik legtöbbet használt diagram típusa az osztálydiagram. Arra használjuk az UML osztálydiagramokat, hogy az aktuális programunk objektumorientált vázát illetve szerkezetét lemodellezessük. A modell tartalmazni fogja az osztályokat és a köztük fennálló viszonyokat. Programozás során nagyban elősegíti a programozó csapat munkáját.

Az UML osztálydiagram hasznos, ha egy programozó team nagy projekten dolgozik. Ekkor a team tagjai közötti kommunikációban és az átláthatóságban nagyon hasznos. Megkönnyíti és gyorsítja is a munkafolyamatot. Viszont kisebb projektekre igazából fölösleges, lényegében díszként szolgál. Több idő mire megcsinálod, mint amennyit spórolsz vele.

Egy fizetős szoftverrel, mégpedig a Visual Paradigm nevű programmal készítettem el az UML diagramot. Pár kattintás után gondolkodás nélkül lemodellez a kódunkat. Nekünk csak annyi a dolgunk, hogy kiválasztjuk a kódot.

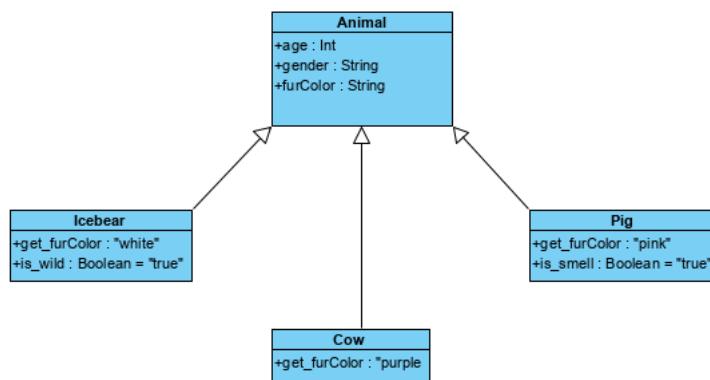


14.1. ábra. Visual Paradigm által létrehozott UML diagram

14.2. Forward engineering UML osztálydiagram

UML-ben tervezünk osztályokat és generálunk belőle forrást!

A diagramból generált kód ebben a [mappában](#) található.



14.2. ábra. Visual Paradigm által Animal UML diagram

Előző feladathoz képest most fordítva kell gondolkodnunk. Igazából most helyes a sorrend. Mégpedig elsőként létrehozunk egy UML osztálydiagramot, majd abból generálunk forráskódot. Ebben a sorrendben érdemes dolgozni egy projekten is. Hiszen az UML diagram segít nekünk fejlesztés közben. Most akkor mikor is gondolkodtunk fordítva?

Lényeg a lényeg, hogy most is Visual Paradigm segítségével dolgozunk. A diagram létrehozása egyszerű. Mégpedig: Diagram > New > Class Diagram és itt az üres (blank) diagramot választjuk ki. Ezután pedig a

Class gombra kattintva tudunk létrehozni osztályokat, azon belül jobb klikkel új attribútumokat. Ha ezekkel megvagyunk mehet az osztályok összekötése és kész is a diagram.

UML class diagramból pedig szintén egy perc alatt létrehozhatjuk a kódot. Mégpedig a következőképpen: Tools > Code > Instant Generator, majd itt kiválasztjuk a nyelvet (Java, C++, C# stb.), kiválasztjuk a diagramunkat és megadjuk a mappát ahova behúzhatja a fájlokat. Végül klick a Generate gombra és kész.

Egy egyszerű osztálydiagramot készítettem, ami a fejezet elején látható.

14.3. Egy esettan

A BME-s C++ tankönyv 14. fejezetét (427-444 elmélet, 445-469 az esettan) dolgozzuk fel!

A könyv az UML diagramról ír, ezt kell feldolgoznunk. Elméleti részei röviden a következők:

Osztályok: ezek a legfontosabb elemei az UML diagramoknak. Téglalappal jelöljük és egy téglalapot három részre tudunk osztani. A téglalap első részén (legfelül) az osztálynév helyezkedik el. Második részén (középen) a tagváltozók és a harmadik részén (azaz legalul) az osztály metódusai és tagfüggvényei helyezkednek el.

Láthatósági szintek szintaxisa: lehet public(+), private (-), protected(#), valamint package(~) is , nyílván azutóbbi c++-ban nem definiált.

A származtatott tagváltozókat "/" jellet jelöljük.

A konstans tagváltozókat read-only tulajdonsággal jelöljük. A tagfüggvényeket szintén láthatósági névvel látjuk el eloször , majd a paraméter listával, végül a visszatérési értékkel. A paraméterlita szintaxisa: irány név: típus = alap érték.

Az irány 3 féle lehet : in(érték szerinti paraméterátadás) out (referenci szerinti) inout(pointert adunk át) Kapcsolatok Asszociáció: Amikor két osztály elakarja érni egymás objektumait,akkor van szükség asszociációra. Vonallal jelöljük. A vonalra kerülnek az érintett objektumok nevei. Valamint egy nyíl jelöli az irányt.

Szerepnév: megmutatja, hogy miként vesz részt az adott osztály az asszociációban. Az asszociációt van két speciális fajtája, a kompozíció és az aggregáció

Aggregáció: Olyan asszociáció ,ami tartalmazást jelöl, jele: egy üres rombusz a tartalmazó oldalán.

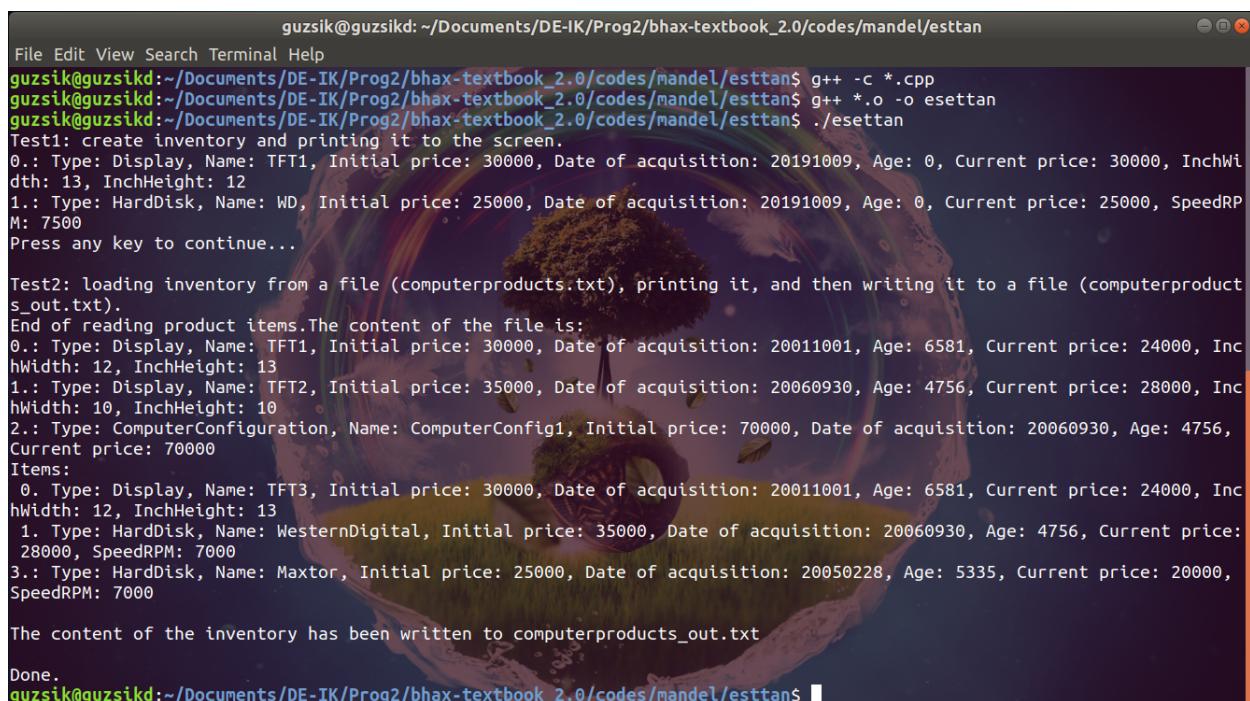
Kompozíció: A tartalmazott objektum pontosan egy tartalmazó objektumhoz tartozik. A tartalmazó és a tartalmazott objektum együtt jön létre, és együtt sz "unik meg.

Gyakorlati része pedig innentől:

A könyv végigvezet minket egy program megírásán. Könyvtárak formájában és az lesz a lényeg, hogy késobb ne keljen átírni a forráskódot nagyon. A megírt program egy informatika cég termékeit fogja tárolni alkatrészenként (megjelenítök, merevlemezek) és összetett termékekkel (számítógép). Az alap osztály amit minden örökölni fog az a Product osztály ez tartalmazza a nevet a termék korát és ki is tudja számolni az árat. Az á függvényt az alosztályokban felüldefiniáljuk a megfelelő termék tulajdonságaira alapozva pl. a merevlemezeti leárazzuk 30 majd 90 nap után. A programnak tudnia kell olvasni fájlóból és azt eltárolnia adatbázisban. Erre írtunk egy void függvényt a Program osztályba.

Soronként fogjuk beolvasni az adatokat és az 1. karakter lesz a termékazonosító. Az áron kívül minden tud tárolni -az árat a termék korából számoljuk ki- ezért az árat majd a gyerek osztályokban felüldefiniálhatjuk.

A nehezebb feladat az összetett termékek tárolása ezt az osztályt is a Product osztályból származtatjuk és azokat az alkatrészeket amikbol felépülnek egy külön vektorban tároljuk el és ehez a vektorhoz írnuk függvényeket amik hozzáadnak alkatrészeket. Ha ezzel elkészültünk akkor már csak az adatok kiírásáról kell gondoskodnunk. Ezt a Product osztályban található virtual void Printparams függvény csinálja. Ezt terméknek megfeleloen felül lehet definiálni. A termékeket nyilván kell tartanunk ezért be kell vezetnünk egy új osztályt ami ezt végzi. A példányosításhoz létrehozunk egy ProductFactory osztályt és a termékek kezelését az ebben található ReadAndCreateProduct függvényre bízzuk.



```
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/mandel/esettan
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/mandel/esettan$ g++ -c *.cpp
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/mandel/esettan$ g++ *.o -o esettan
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/mandel/esettan$ ./esettan
Test1: create inventory and printing it to the screen.
0.: Type: Display, Name: TFT1, Initial price: 30000, Date of acquisition: 20191009, Age: 0, Current price: 30000, InchWidth: 13, InchHeight: 12
1.: Type: HardDisk, Name: WD, Initial price: 25000, Date of acquisition: 20191009, Age: 0, Current price: 25000, SpeedRPM: 7500
Press any key to continue...

Test2: loading inventory from a file (computerproducts.txt), printing it, and then writing it to a file (computerproducts_out.txt).
End of reading product items.The content of the file is:
0.: Type: Display, Name: TFT1, Initial price: 30000, Date of acquisition: 20011001, Age: 6581, Current price: 24000, InchWidth: 12, InchHeight: 13
1.: Type: Display, Name: TFT2, Initial price: 35000, Date of acquisition: 20060930, Age: 4756, Current price: 28000, InchWidth: 10, InchHeight: 10
2.: Type: ComputerConfiguration, Name: ComputerConfig1, Initial price: 70000, Date of acquisition: 20060930, Age: 4756, Current price: 70000
Items:
0. Type: Display, Name: TFT3, Initial price: 30000, Date of acquisition: 20011001, Age: 6581, Current price: 24000, InchWidth: 12, InchHeight: 13
1. Type: HardDisk, Name: WesternDigital, Initial price: 35000, Date of acquisition: 20060930, Age: 4756, Current price: 28000, SpeedRPM: 7000
3.: Type: HardDisk, Name: Maxtor, Initial price: 25000, Date of acquisition: 20050228, Age: 5335, Current price: 20000, SpeedRPM: 7000
The content of the inventory has been written to computerproducts_out.txt
Done.
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/mandel/esettan$
```

14.3. ábra. Esettan futtatva

14.4. BPMN

Rajzolunk le egy tevékenységet BPMN-ben!

Segítség: [Prog2_7.pdf](#) (34-47 fólia)

Az UML-hez hasonló workflow leíró eszköz, amely könnyen értelmezhető grafikus jelölérendszer biztosít az üzleti folyamatok ábrázolásához. Szabványos eszközök számít az üzleti folyamatok, valamint a webes szolgáltatások modellezéséhez. A BPMN fő célja egy olyan üzleti folyamat jelölérendszer biztosítása, amely minden stakeholder számára értelmezhető, az üzleti elemzőktől kezdve a szoftverfejlesztőkön át a vállalati vezetőkig, ezáltal a kommunikációt nagyban megkönyítve. A másik célja, hogy az XML alapú nyelvek, melyeket üzleti folyamatok végrehajtására terveztek, vizualizálhatók legyenek egy szabványos jelölérendszerrel. A BPMN folyamat-orientált megközelítéssel dolgozik. Három alapobjektumát a tevékenységek (activities), események (events), és az átjárók (gateways) jelentik.

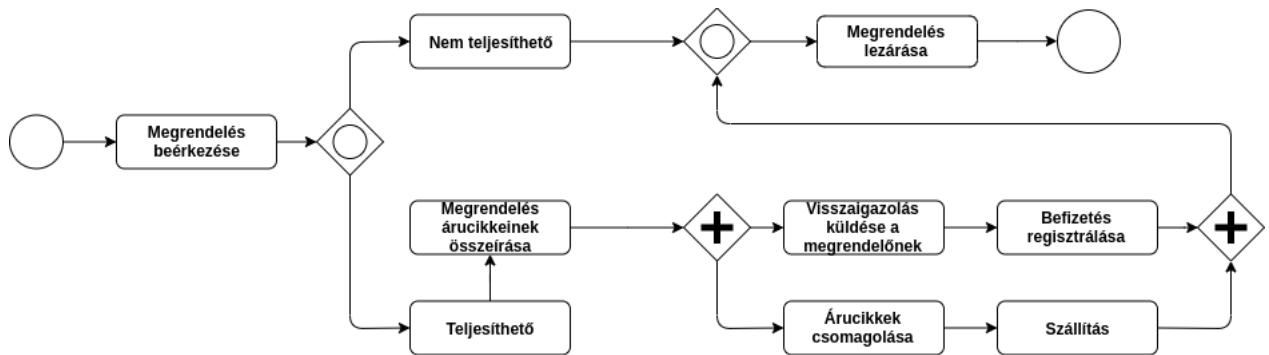
Egy esemény a vizsgált folyamat során következik be, van kiváltója, illetve valamilyen eredménye vagy hatása (pl. szállítmány beérkezte). A tevékenységek valamilyen általános művelet végrehajtását jelentik (pl. számla kiállítása, kamatok számítása). Az átjárók logikai kapcsolóként (és/vagy/xor műveletek) szolgálnak

a tevékenységfolyamok szétválasztása vagy egyesítése során. A BPMN jelölőrendszer számos objektumot tartalmaz, melyeket a referencia ismertet.

A BPMN eredménye egy üzleti folyamatokat is érthetően ábrázoló diagram, melyet Business Process Diagram-nak (BPD) neveznek. Vállalati munkafolyamatok modellezéséhez, meg kell adni egy kezdő-eseményt, további eseményeket és tevékenységeket, üzleti döntésekkel (workflow elágaztatása átjárókkal), valamint a kimenetet és eredményeket.

A BPMN üzleti folyamat modellező standarddá vált. Népszerűségét könnyű elsajátíthatóságának és értelmezhetőségének köszönheti.

Egy általam elkészített kisebb BPMN diagram látható az alábbi képen. Egyszerűen elkészíthető például a [draw.io](#) oldalon. Pár kattintás csupán és kész is.



14.4. ábra. BPMN egy üzleti tevékenységről

15. fejezet

Helló, Chomsky!

15.1. Encoding

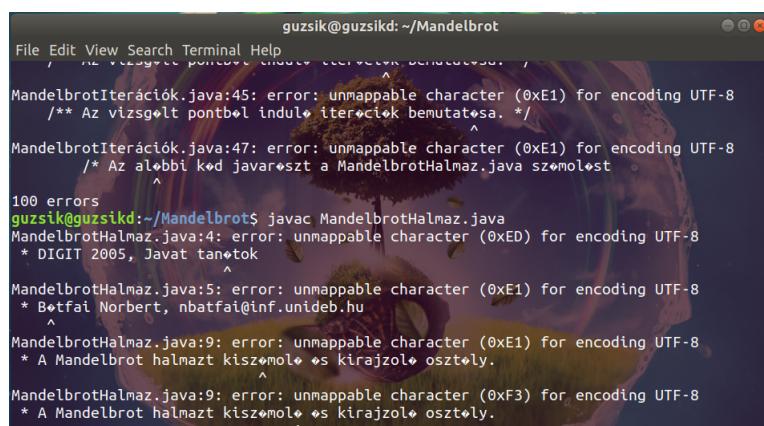
Fordítsuk le és futtassuk a Javat tanítok könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezetes betűket! [adatok.html](#)

A feladatot a fent linkelt oldalon található "letölthető segédanyagok" között lévő [zip](#) állományon belül található kódokból van értelme kivitelezni.

Ha nincs kedved mazsolázni a zip állományból, akkor közvetlenül itt vannak a szükséges források:

- [MandelbrotHalmaz.java](#)
- [MandelbrotHalmazNagyító.java](#)
- [MandelbrotIteráció.java](#)

Kíváncsiságból a forrásfájlokat megpróbáltam hagyományos módon lefuttatni a parancssorban. Az alábbi képen látható, hogy mit kaptam eredményül. A kép egy kis részlet és rengeteg hibaüzenetet kapunk.



The screenshot shows a terminal window with the following text output:

```
guzsik@guzsikd:~/Mandelbrot$ javac MandelbrotHalmaz.java
MandelbrotHalmaz.java:45: error: unmappable character (0xE1) for encoding UTF-8
    /* Az vizsgolt pontból induló iterációk bemutatása. */
                                         ^
MandelbrotHalmaz.java:47: error: unmappable character (0xE1) for encoding UTF-8
    /* Az alebbi kód javaröszt a MandelbrotHalmaz.java szövegben. */
                                         ^
100 errors
guzsik@guzsikd:~/Mandelbrot$ javac MandelbrotHalmaz.java
MandelbrotHalmaz.java:4: error: unmappable character (0xED) for encoding UTF-8
 * DIGIT 2005, Javat tanított
                                         ^
MandelbrotHalmaz.java:5: error: unmappable character (0xE1) for encoding UTF-8
 * Bétfai Norbert, nbafai@inf.unideb.hu
                                         ^
MandelbrotHalmaz.java:9: error: unmappable character (0xE1) for encoding UTF-8
 * A Mandelbrot halmazt kiszámoltam és kirajzoltam osztály.
                                         ^
MandelbrotHalmaz.java:9: error: unmappable character (0xF3) for encoding UTF-8
 * A Mandelbrot halmazt kiszámoltam és kirajzoltam osztály.
                                         ^
```

15.1. ábra. Encoding proba

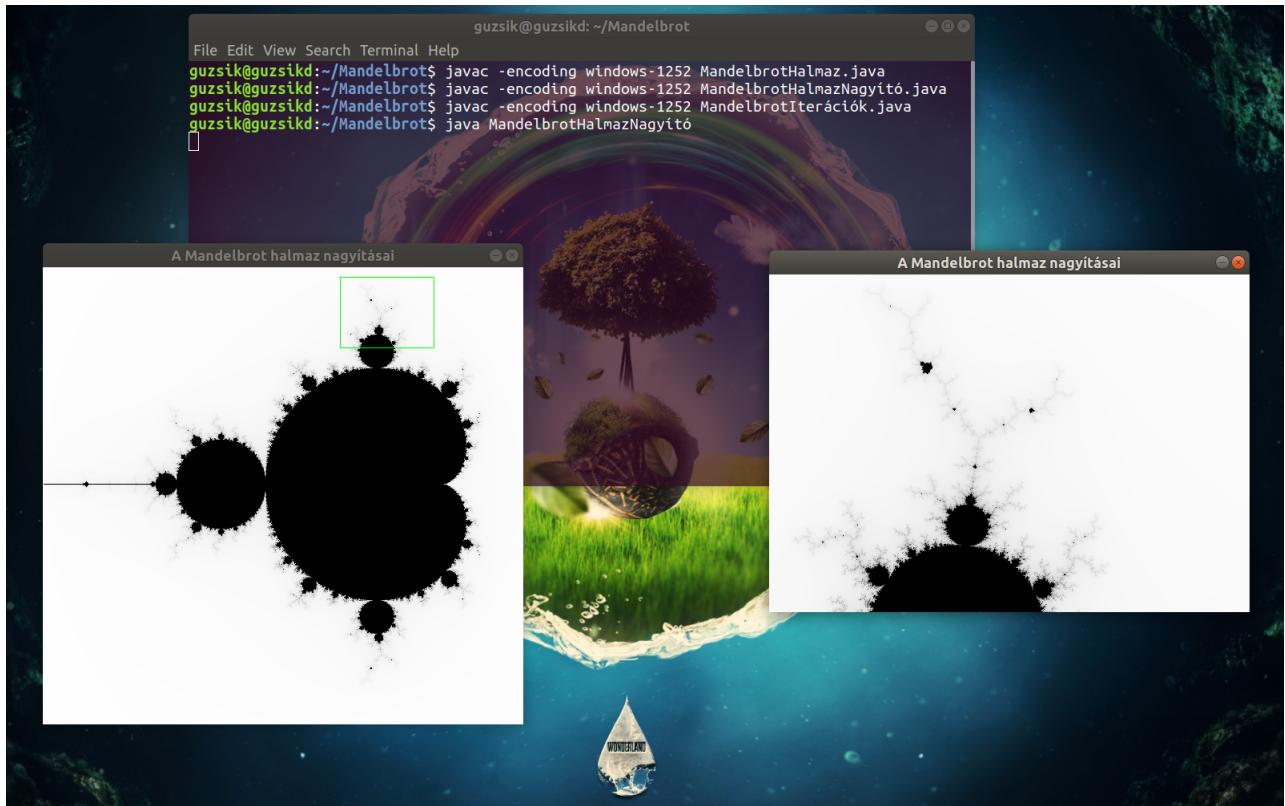
A `javac` fordító alapvetően egy bizonyos karakterkészlettel rendelkező forrásokra számít, mikor megkapja az inputot. Ezt az alapértelmezett karakterkészletet a rendszerbeállításokból örökli. Tehát a rendszerbeállításokban az általunk megadott karakterkészlet alkotja a `javac "default"` avagy alapértelmezett karakterkészletét.

Esetünkben az alapértelmezett karakterkészlettel nem megyek semmire, mert az ékezes betűkkel nem tud mit kezdeni a `javac` fordítónk...

Mi a teendőnk? A `javac` fordítónak közvetlenül is megtudjuk adni, hogy milyen karakterkészlettel dolgozon egyes inputok esetén.

Először is rá kellett jönni, hogy milyen karakterkészlettel rendelkeznek pontosan a forráskódok. Az ékezes betűk miatt valamelyik latin kódolás kell most nekünk. Pár próbálkozás után a Windows Latin1 karakterkészlete sikert aratott. Így a feladat megoldásához `-encoding windows-1252` utasítást kell egyszerűen a `javac` fordítónak megadni.

Az alábbi képen látható, hogy a megfelelő karakterkészlet megadásával hibátlanul lefut a Mandelbrot nagyítónk.



15.2. ábra. Encoding siker

15.2. Full screen

Készítsünk egy teljes képernyős Java programot! Tipp: https://www.tankonyvtar.hu/en/tartalom/tkt/javatanitok-javat/ch03.html#labirintus_jatek

Tipp alapján készített feladat.

Megoldás forrása: [labirintusjatek](#)

A feladat megoldásához a Tanár Úr által mellékelt programot, vagyis a labirintus játékot élesztettem fel, amely alapból teljes képernyos móddal rendelkezik. A teljes képernyős mód kódját a LabirintusJáték.java fájlban tudjuk megnézni, a teljesKépernyosMód metóduson belül:

```
public void teljesKépernyosMód(GraphicsDevice graphicsDevice)
{
    int szélesség = 0;
    int magasság = 0;
    // Nincs ablak fejléc, keret.
    setUndecorated(true);
    // Mi magunk fogunk rajzolni.
    setIgnoreRepaint(true);
    // Nincs átméretezés
    setResizable(false);
```

A fentiek után megnézzük továbbá, hogy át tudunk-e kapcsolni teljes képernyos módba:

```
boolean fullScreenTamogatott = graphicsDevice.isFullScreenSupported();
```

Ha át tudunk kapcsolni, akkor kapcsoljuk is át, és fullscreen "exkluzívba" váltunk.

```
if(fullScreenTamogatott) {
    graphicsDevice.setFullScreenWindow(this);
```

Ezután bekérjük a képernyő jellemz „oit: szélesség, magasság, színmélység, kéfrissítés; majd kiírjuk azokat

```
java.awt.DisplayMode displayMode
    = graphicsDevice.getDisplayMode();
szélesség = displayMode.getWidth();
magasság = displayMode.getHeight();
int színMélység = displayMode.getBitDepth();
int frissítésiFrekvencia = displayMode.getRefreshRate();
System.out.println(szélesség
    + "x" + magasság
    + ", " + színMélység
    + ", " + frissítésiFrekvencia);
```

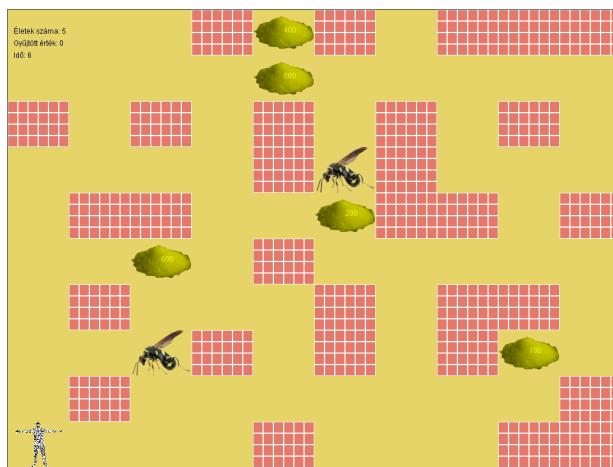
Elkérjük a képernyő lehetséges beállításait, majd megnézzük, hogy támogatja-e az 1024x768-as képernyő felbonntást, mivel a labirintus játékot ekkora felbontáshoz írtuk.

```
java.awt.DisplayMode[] displayModes
    = graphicsDevice.getDisplayModes();
boolean dm1024x768 = false;
for(int i=0; i<displayModes.length; ++i) {
    if(displayModes[i].getWidth() == 1024
        && displayModes[i].getHeight() == 768
        && displayModes[i].getBitDepth() == színMélység
        && displayModes[i].getRefreshRate()
```

```
    == frissítésiFrekvencia) {  
    graphicsDevice.setDisplayMode(displayModes[i]);  
    dm1024x768 = true;  
    break;  
}  
}
```

Ha nem elérhető az 1024x768 felbontás, tudassuk ezt a felhasználóval, egyébként folytathatjuk a kép felbontásának átállításával:

```
if(!dm1024x768)  
    System.out.println("Nem megy az 1024x768, de a példa képméretei ehhez a ←  
                      felbontáshoz vannak állítva.");  
} else {  
    setSize(szélesség, magasság);  
    validate();  
    setVisible(true);  
}  
createBufferStrategy(2);  
bufferStrategy = getBufferStrategy();  
}
```



15.3. ábra. Labirintus

15.3. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Lásd vis_prel_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, textúrázás, a színtek jobb elkülönítése, kézreállóbb irányítás.

Megoldás forrása:

A program fordításához szükségünk van a lib-boost csomagra és a freeglut3-ra.

Ezeket egyszerűen letölthetjük az alábbi parancsal:

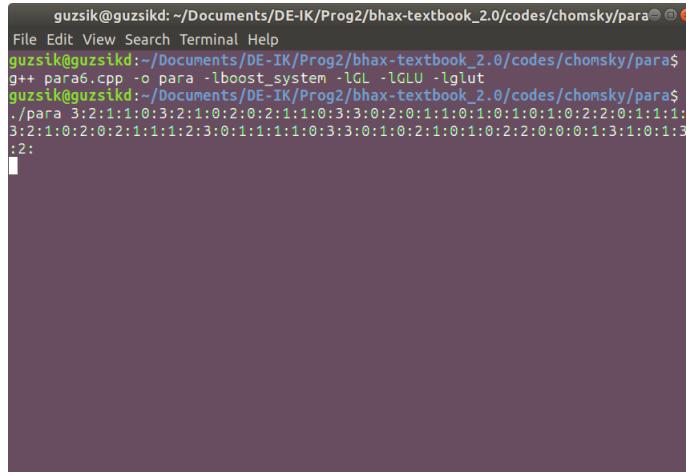
- sudo apt-get install libboost-all-dev
- sudo apt-get install freeglut3-dev

Miután megvagyunk a csomagok telepítésével a g++ para6.cpp -o para -lboost_system -lGL -lGLU -lglut parancs segítségével fordíthatjuk a programunkat.

Ha már lehetett választani, hogy mit írunk át rajta elsőként a TAB gombot vezettem be, hogy ne csak számok segítségével lépkedhessünk a kockák között.

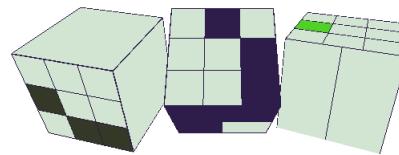
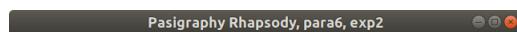
Másodikként szintén billentyű funkciókat variáltam. Mégpedig megfordítottam a nyilak által irányított forgási irányt. Egyből egyszerűbb forgatni arra amerre akarjuk.

Picit a megjelenésébe, azaz a színvilágba is belepiszkáltam.



```
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/chomsky/para$ g++ para6.cpp -o para -lboost_system -lGL -lGLU -lglut
guzsik@guzsikd:~/Documents/DE-IK/Prog2/bhax-textbook_2.0/codes/chomsky/para$ ./para
./para 3:2:1:1:0:3:2:1:0:2:0:2:1:1:0:1:3:3:0:2:0:1:1:0:1:0:1:0:1:0:2:2:0:1:1:1:3:2:1:0:2:0:2:1:1:1:2:3:0:1:1:1:0:3:3:0:1:0:2:1:0:1:0:2:2:0:0:1:3:1:0:1:3:2:
```

15.4. ábra. Paszigráfia Rapszódia OpenGL futtatás



15.5. ábra. Paszigráfia Rapszódia OpenGL

15.4. Perceptron osztály

Dolgozzuk be egy külön projektbe a projekt Perceptron osztályát! Lásd <https://youtu.be/XpBnR31BRJY>

Megoldás forrása:

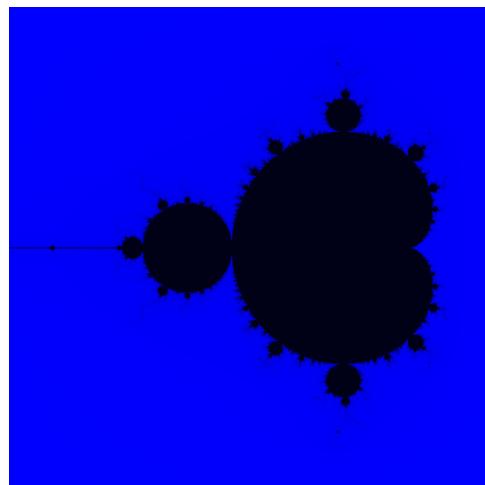
- mandel.cpp
 - main.cpp
 - ml.hpp

A feladathoz szükséges telepíteni a libpng12 és a libpng12-dev könyvtárakat. Ezen felül png++ könyvtár is szükséges nekünk, amit a forrásfájlból tudunk telepíteni.

Elsőként az indítási első argumentumként megadott fájlból beolvassuk az image típusú képet. Beolvasás után a png++ könyvtár segítségével mégpedig a get_width() és a get_height() függvények segítségével lekérjük a beolvasott kép magasságát és szélességét.

Mivel létre lett hozva a mlp.hpp ebbol használom a Perceptron osztályt, ezt példányosítottam a kép méretével. Majd egy dupla for ciklus segítségével, ami végig fut a kép szélességén és magasságán, majd beállítja a $i * \text{kép szélesség} * j$ pixelt az i és j pozíión lévő pirosra.

15.6. ábra. Perceptron futtatás



15.7. ábra. Perceptron kimenet

16. fejezet

Helló, Stroustrup!

16.1. JDK osztályok

Írunk olyan Boost C++ programot (indulj ki például a fénykardból) amely kilistázza a JDK összes osztályát (miután kicsomagoltuk az src.zip állományt, arra ráengedve)!

Megoldás forrása: [boost](#) mappápan

Kis alapok a feladat megoldásához: [Boost \(C++ libraries\)](#)

Feladatunk szerint Bátfa Tanár úr FUTURE projektjének programkódjai közül a fénykard programból kellett kiindulni. Ezt sikerült is fellelni mégpedig [itt](#)

Amiért nekünk ebből a fénykard programból kellett kiindulni, az a `read_acts` függvény. Mégpedig azért, mert ez a függvény nagyon hasonló dolgokat művel, mint amit mi is elszeretnénk érni a programunkkal. A példa függvényünk egy adott kiterjesztésű fájlokat keres mialatt végigmegy egy állománszerkezeten, majd a bennük található információkat kiolvassa és eltárolja.

Nekünk igazából nagyon hasonló a feladatunk, mivel egy állománszerkezet bejárása közben `.java` fájlokat keresünk.

Nézzük is a feladatot:

Miután indukáltuk a megfelelő osztályokat, egy változót deklaráltunk ami az osztályok számának tárolására szolgál.

Ezek után egy `read_classes` eljárás következik. Ez a következőképpen működik: Egy `if` feltételvizsgálaton belül először is eldöntjük, hogya fejlhearchiában hol állunk. Ezt egy `is_regular_file()` függvény segítségével tesszük meg. Ha nem könyvtárak, hanem hagyományos fájlok állnak rendelkezésre akkor lehet a vizsgálat, hogy van-e `.java` kiterjesztésű fájl. Ha igen, akkor növeljük a számlálónkat. Ha könyvtárakkal vagyunk körbevéve, nem pedig fájlokkal, akkor az `is_directory` függvénytellyel egy for ciklusba lépünk. Rekurzívan meghívjuk a `read_classes` függvényt és így jutunk el lépésről lépére a fájlokhoz.

A `main` függvényün a következőképpen alakult: tartalmazza a `read_classes` függvényhívást, itt hozzuk létre az osztályokat tároló vektort és itt íratjuk ki hány osztályt találtunk az `src` mappápan.

Futtatni a következő parancsokkal tudjuk:

- g++ boost.cpp -o bejaro -lboost_system -lboost_filesystem -lboost_program -std=c++14
- ./boost src

16.2. Másoló-mozgató szemantika

Kódcsipeteken (copy és move ctor és assign) keresztül vesd össze a C++11 másoló és a mozgató szemantikáját, a mozgató konstruktort alapozd a mozgató értékkedásra!

Megoldás forrása:

•

A megoldáshoz a klasszikusnak számító LZWBInfa kódunkat használtam.

Fontos még az elején, hogy tiszta vizet öntsünk a pohárba. Tisztázzuk le, hogy amikor egy objektum létrehozásakor értéket adunk az adott objektumnak, akkor hívódik meg a mozgató konstruktor. Például.: LZWBInFa binFa = binFa_regi; Viszont ha egy már inicializált objektumnak (LZWBInFa binFa2;) adunk új értéket (binFa2 = binFa_regi;), akkor beszélünk másoló/mozgató értékkedásról.

Lássuk, hogy hogyan kell elkészíteni a másoló konstruktort. Mivel konstruktorról beszélünk, ezért a neve megegyezik az osztály nevével. A vezérlés akkor adódik át erre a konstruktorra, ha magával megegyező típusú referenciát adunk át.

```
LZWBInFa::LZWBInFa (const LZWBInFa & forras) {
    std::cout << "Copy ctor" << std::endl;
    gyoker = new Csomopont('/');
    gyoker->ujEgyesGyermek(masol(forras.gyoker->
        egyesGyermek(), forras.fa));
    gyoker->ujNullasGyermek(masol(forras.gyoker->
        nullasGyermek(), forras.fa));
    if (forras.fa == forras.gyoker) {
        fa = gyoker;
    }
}
```

Látható, hogy ez a konstruktor egy LZWBInFa konstans referenciát vár paraméterként. Mivel ilyenkor nem hívódik meg az alap konstruktor, emiatt a gyöker pointerünknek értéket kell adni, különben memóriacímzési hibát kaphatunk. Ha ezzel kész vagyunk, akkor létre kell hozni a gyökértol kiindulva az egyes gyermekeket, ezért van szükség a ujEgyesGyermek és ujNullasGyermek függvényekre. Mivel egy már elkészített fából indulunk ki, ezért annak a mintájára készítjük el a gyermekeket. Ebben a másol függvény van segítségünkre. Ennek két paramétere van, az egyik a forrás gyökerének a gyermeke, a másik pedig a forrás fa mutatója, mivel azt is szeretnénk átmásolni, hogy a fa mutató éppen hol áll az eredeti fában.

```
Csomopont* LZWBInFa::masol (Csmopont* elem, Csmopont* regi_fa) {
    Csmopont* ujelem = nullptr;
    if (elem != nullptr) {
```

```

ujelem = new Csomopont (elem->getBetu());
ujelem -> ujEgyesGyermek(masol(elem->egyesGyermek(), regi_fa));
ujelem -> ujNullasGyermek(masol(elem->>nullasGyermek (), regi_fa)) ←
;
if (regi_fa == elem) {
    fa = ujelem;
}
return ujelem;
}

```

A masol egy Csomopont mutatót ad vissza. Elso lépésben létrehozzuk a kés "obb átadni kívánt csomópon-tot. Majd ellenorizzük, hogy az eredeti f csomópontjának van-e értéke, vagy null pointer. Utóbbi esetben szimplán visszaadunk egy null pointert. Ellenkezo esetben az ujelem pointernek átadunk egy az eredeti csomópont alapján inicializált csomóponthoz tartató memóriacímet. Ezután újra meghívjuk a masol függ-vényt annak érdekében, hogy elkészítsük az ujelem gyermekeit. Ezt addig folytatjuk, ameddig az eredeti fa végére nem érünk. Abban az esetben, ha a régi fához tartozó fa mutató a masol függvénynek átadott elemre mutat, akkor az új fánk fa mutatóját ezen elem alapján létrehozott csomópontra állítjuk. Végezetül pedig visszatérünk a csomóponttal. Ahhoz, hogy a fa mutatót a gyoker-re is állíthassuk, a másoló konstruktörben is ellenorizzük, hogy hova mutat az eredeti fa mutatója.

Most pedig a másoló értékadás következik:

```

LZWBinFa & LZWBinFa::operator= (const LZWBinFa & forras) {
    std::cout << "Copy assaignement" << std::endl;

    gyoker->ujEgyesGyermek(masol(forras.gyoker-> egyesGyermek(), forras ←
        .fa));
    gyoker->ujNullasGyermek(masol(forras.gyoker-> nullasGyermek(), ←
        forras.fa));

    if (forras.fa == forras.gyoker) {
        fa = gyoker;
    }
    return *this;
}

```

Lényegében egy operátor túlterheléséről van szó. Szemantikailag különbség a konstruktőrhoz képest, hogy a másoló értékadásnak van visszatérési értéke, jelen esetben LZWBinFa referencia. Ennek megfelelően visszaadunk egy mutatót arról az objektumról, amely az egyenloség bal oldalán volt. Ezen kívül a már ismert eljárást követjük.

A C++11 másoló szemantikájának megismerése után folytassuk a mozgató szemantikával, azon belül is a mozgató konstruktőrral. A mozgató konstruktur paraméteréül egy jobbértékreferenciát vár. A feladat szerint a mozgató értékadásra kell alapoznunk.

```

LZWBinFa::LZWBinFa (LZWBinFa&& forras)
{
    std::cout<<"Move ctor\n";
    gyoker = nullptr;
    *this = std::move(forras); //ezzel kényszerítjük ki, hogy a mozgató ←
        értékadást használja
}

```

}

Az alapkoncepciója az a mozgató értékkadásra alapozásnak, hogy lényegében felcseréljük a két fa gyökér mutatójának értékét. Ennek érdekében az újonan létrehozni kívánt fa gyökerét null mutatóvá tesszük, majd meghívjuk a mozgató értékkadást. Ehhez a `std::move` függvényre van szükségünk, amely bemenetéül kapott paramétert jobbértékreferenciává alakítja. Mivel a `this` egy már inicializált objektumot jelöl, ezért nem a mozgató konstruktor, hanem a mozgató értékkadás hívódik meg.

```
LZWBinFa& LZWBinFa::operator= (LZWBinFa&& forras)
{
    std::cout<<"Move assignment ctor\n";
    std::swap(gyoker, forras.gyoker);
    return *this;
}
```

A mozgató értékkadás feladata a már korábban említett érték csere, melyet a `std::swap` valósít meg. Mivel megcsérélődik a `gyoker` és a `forras.gyoker` által mutatott tárterület, ezért a mozgatás teljes egészében megvalósul. Hiszen a `gyoker` egy null mutató, emiatt a csere után a `forras.gyoker` is null mutató lesz, vagyis az eredeti fa "töröldött". Természetesen ez nem szó szerint igaz, hiszen csak egy másik pointeren keresztül hivatkozunk a már korábban a memóriában lefoglalt és tárolt fára.

16.3. Hibásan implementált RSA törése

Készítünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: [Prog2_3.pdf](#) (71-73 fólia) által készített titkos szövegen.

Megoldás forrása: [rsa](#) mappában

Az első publikációk óta sok kutató, köztük maguk a szerzők is keresték az RSA algoritmus gyenge pontjait. Az eltelt huszonöt év vizsgálódásai egy sor nagyon érdekes elméleti és gyakorlati támadási módszert eredményeztek, azonban eddig egyik sem oldotta meg az RSA általános megfejthetőségét, mivel a számítástechnika rohamos fejlődése ezeket a támadásokat technikával kompenzálta. Jelen dolgozat célja, hogy rámutasson arra, hogy bár a támadások egyenként nem rendítik meg (bizonyos feltételek teljesülése esetén!) az RSA biztonságát, összességükben olyan arzenált képviselnek, amelyek állandó szemelőtt tartása jelentősen megnehezíti (lelassítja) az RSA tömeges alkalmazását.

Az RSA-eljárás nyílt kulcsú (vagyis „aszimmetrikus”) titkosító algoritmus, melyet 1976-ban Ron Rivest, Adi Shamir és Len Adleman fejlesztett ki (és az elnevezést nevük kezdőbetűiből kapta). Ez napjaink egyik leggyakrabban használt titkosítási eljárása. Az eljárás elméleti alapjait a moduláris számelmélet és a prím-számelmélet egyes tételei jelentik.

Bővebben [itt](#) tudsz utána nézni az RSA titkosításnak, ha jobban bele szeretnél mélyülni. Esetleg még a [wiki](#) áll a rendelkezésre, ami könnyen értelmezhető.

A feladathoz fontos tudni az alapokat a `BigInteger` osztályról. A JDK részét képzi, mégpedig a `java.math` csomag osztálya a `BigInteger` osztály. Leggyakrabban kódolók és kódtörők elkészítéséhez használják és maga a típus egy 32-bites egészből áll.

A nyilvános és titkos kulcs generálását az [RSA.java](#) végzi.

[RSA2.java](#) kódunk pedig a következőképpen néz ki:

A KulcsPar osztályunk felel a titkosításhoz szükséges kulcspárok előállításáról. A p és q számokból kapjuk a modulust. A fontosabb számjaink: m a modulo, e a kitevő és d az inverze.

A main-en belül titkosítunk. Először is megadjuk a titkosítatlan szöveget, majd pedig egy for ciklus segítségével végigmegyünk a szövegünkön és feltöljük azt az ASCII kódjával. Itt a for cikluson belül található egy modPow függvény, ami megkapja a hatványkitevőt és a modulust.

A titkosított szöveget vissza is kell fordítanunk, ami a következőképpen működik. A modPow függvényünk első paraméterként inverzet kap, így az előzőhöz képest fordítva működik. Mégpedig BigIntegerből újra ASCII kód sorozatot csinál. A bufferbe a visszafordított szöveg kerül, és ezt is íratjuk ki.

16.4. Változó argumentumszámú ctor

Készítsünk olyan példát, amely egy képet tesz az alábbi projekt Perceptron osztályának bemenetére és a Perceptron ne egy értéket, hanem egy ugyanakkora méretű „képet” adjon vissza. (Lásd még a 4 hét/Perceptron osztály feladatot is.)

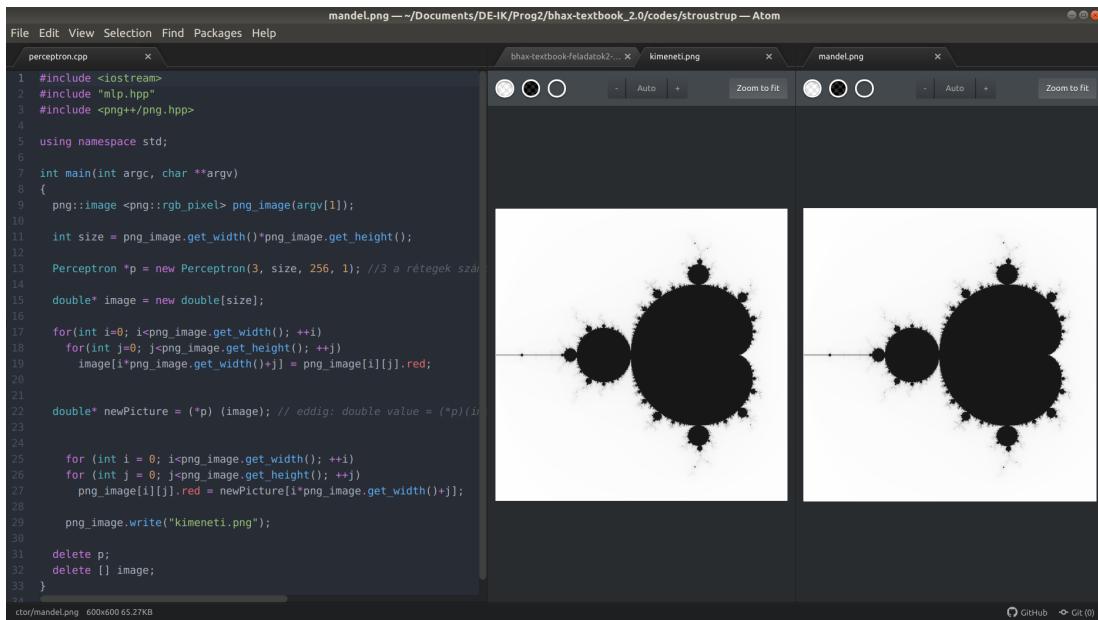
Megoldás forrása: [ctor mappápan](#)

Szükségünk van a mandel.cpp fordításához és futtatásához a libpng, libpng++ könyvtárakra és a png++ png.hpp header fájljára. A könyvtárakat egyszerűen parancssorban tudjuk telepíteni, míg a header fájlhoz lekell szednünk a png++ telepítőállományokat.

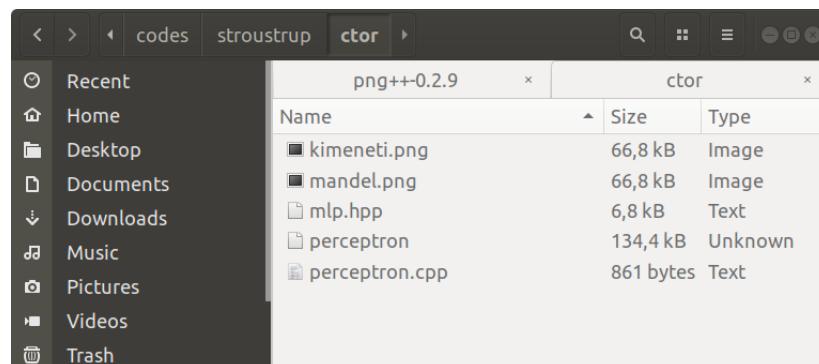
A perceptron az egyik legegyszerűbb előrecsatolt neurális hálózat. A main.cpp segítségével fogjuk szimulálni a hiba-visszaterjesztéses módszert, mely a többrétegű perceptronok (MLP) egyik legfőbb tanítási módszere. Ahhoz, hogy ezt fordítani és futtatni tudjuk később, szükségünk lesz az mlp.hpp header filera, mely már tartalmazza a Perceptron osztályt.

Az előző program futtatásával létrejött Mandelbrot png ábrát fogjuk beimportálni. A header filenak köszönhetően megadhatjuk a rétegek számát, illetve a neuronok darabszámát. A beolvasásra kerülő kép piros részeit a lefoglalt tárba másoljuk bele. A mandel.png alapján új képet állítunk elő, mely megkapja az eredeti kép magasságát és szélességét. A visszakapott értékeket megfeleltetjük a blue értékeknek.

Az első felvonás Perceptron feladatához képest módosításokat kell végezniünk a header file-on is, ugyanis új képet akarunk előállítani. A double pointer () operátor már egy tömböt térít vissza, melynek segítségével bele tudunk nyúlni a képhez. Az utolsó units tömb értékei átkerülnek a paraméterként kapott tömbbe, az eredeti és az új kép egyforma méretű lesz.



16.1. ábra. Futtatás után az új és régi kép



16.2. ábra. Mandelbrot.png képek mérete

17. fejezet

Helló, Gödel!

17.1. Gengszterek

Gengszterek rendezése lambdával a Robotautó Világbajnokságban. Kis segítség a következő [videóban](#) (8:05-től)

Miért akarunk így rendezni? Miért ne? Természetesen mi is implementálhatnánk egy gyorskeresést, vagy egy szimpla beszúrós rendezést, viszont nincs értelme állandóan beimplementálni ilyeneket a projectünkbe, használunk kész, jól implementált, letesztelt algoritmusokat(gyakoriakat)!

Egy elég gyors ("nagy ordó" $N^* \log(n)$), összehasonlító alapon működő rendezést biztosít nekünk a standard könyvtár az algorithm -ban található std::sort révén.

Ahhoz, hogy a lambda(névtelen metódus) alapon történő rendezést megvalósítsuk a 3 paraméteres verzióját kell használnunk, ami vár a rendezendő kollekcióra két iterátort, amik rendre a kollekció elejére és végére mutatnak, majd egy függvényt ami az összehasonlítás alapját fogja definiálni.

A lambda szintaxisa (C++ esetén) a következő részekből adódik:

- [] "capture block", itt megadhatunk a lambdát befoglaló metódus scopejában elérhető változókat elérés céljából, elkaphatjuk őket akár referencia alapján is, alapértelmezetten másolódnak.
- () kerülnek a lambda paraméterei, mi esetünkben a gengszterek pozícióját jelképező x,y koordináták.

Végül jön maga a függvény body:

```
std::sort(gangsters.begin(), gangsters.end(), [this, cop] (unsigned x ←
    ,
    unsigned y)
{
    return dst(cop,x) < dst(cop,y);
});
```

Ennek eredménye az lesz, hogy a rendőrök a legközelebbi gengsztert fogják üldözőbe venni.

17.2. C++11 Custom Allocator

A [CustomAlloc](#)-os példa, lásd C forrást az UDPROG repóban!

A feladat szerint csinálni kell egy egyedi allokátort. Az alap allokátor a `std::allocator<T>`. Jelen példában a pointer `allocate(size_type n)` függvény felel a memória lefoglalásáért. Itt visszatérésre kerül egy `T` típusú lefoglalt `n * T` méretű memória cím. Ez előtt a függvény kiírja a lefoglalás méretét, és a típusát. Fontos felszabadítani a már nem használt memóriát is, ezért felelős a `std::deallocate<T>` függvény, ami felszabadítja a lefoglalt memóriát.

```
using namespace std;
template<typename T>
struct CustomAlloc {
    using size_type = size_t;
    using value_type = T;
    using pointer = T *;
    using const_pointer = const T*;
    using reference = T &;
    using const_reference = const T &;
    using difference_type = ptrdiff_t;
    CustomAlloc() {}
    CustomAlloc(const CustomAlloc&) {}
    ~CustomAlloc() {}
    pointer allocate(size_type n) {
        int s;
        char* p = abi::__cxa_demangle(typeid(T).name(), 0, 0, &s);
        std::cout << "Allocating "
        << n << " object of "
        << n * sizeof(T)
        << " bytes. "
        << typeid(T).name() << " = " << p
        << " hívások száma "
        << std::endl;
        free(p);
        return reinterpret_cast<T*> (new int[n * sizeof(T)]);
    }
    void deallocate(pointer p, size_type n) {
        delete[] reinterpret_cast<int*> (p);
        std::cout << "Deallocating "
        << n << " object of "
        << n * sizeof(T)
        << " bytes. "
        << typeid(T).name() << " = " << p
        << " hívások száma "
        << std::endl;
    }
};
```

Teszteléshez feltöltöttem a `CustomAlloc` típusú vektort értékekkel, és erre vektorra meghívásra kerül a `reserve` függvény, ami elindítja a folyamatot.

```
int main() {
    vector< int, CustomAlloc<int> > v;
    v.reserve(10);
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    v.push_back(5);
    v.push_back(6);
    v.push_back(7);
    v.push_back(8);
    v.push_back(9);
    v.push_back(10);
    for (int x : v) {
        cout << x << "\n";
    }
    return 0;
}
```

Az allokálás megoldható a boost könyvtár segítségével is, amihez a boost/interprocess/allocators/ a könyvtár szükséges. A memóri manageléséhez szükséges a boost/interprocess/managed_shared_memory könyvtár, és mivel vektorokkal dolgozik a kód, ezért kell még a boost/interprocess/containers/vector könyvtár is.

Elsőnek csinálni kell egy megosztott memóriát, ez a segment nevet kapja. Ezután létre kell hozni az allokátort, ami az Allocator nevet kapja, ez alapján kerül meghatározásra a vektor. Majd végül az allokátort példányosítani kell, ami az alloc_inst nevet kapja, argumentnek meg a lefoglalt memória van megadva. Teszt esetnek létrehoztam egy TP típusú alloc_inst allokátor általi vektort, majd feltöltöm elemekkel.

```
boost::interprocess::managed_shared_memory segment(boost::interprocess::open_or_create, "shm", 1024*10*1024);

typedef boost::interprocess::allocator<int, boost::interprocess::managed_shared_memory::segment_manager> Allocator;
typedef boost::interprocess::vector<int, Allocator> TP;
const Allocator alloc_inst(segment.get_segment_manager());

TP *v = segment.construct<TP>("shm") (alloc_inst);
v->push_back(1);
v->push_back(2);
v->push_back(3);
v->push_back(4);
v->push_back(5);
```

17.3. STL map érték szerinti rendezése

Például: [fénykard.cpp](#)

Hasonlóan az előző feladathoz kollekciókat fogunk rendezni lambda függvény segítségével, viszont itt már egy összetettebb gyűjtemény kerül a műtőasztalra: a map, azaz egy táblázat, vagy szótár ami kulcs-érték párokból álló rekordok gyűjteménye.

What is the catch? Nem tudunk rendezni közvetlenül egy std::map szerkezetet, hiszen, mint a Java HashMap esetében az elemek bejárás nem feltétlenül történik sorrendben, tehát létre kell hoznunk egy őt reprezentáló vektort ami a std::map kulcs-érték párait tartalmazza. Java esetén itt egyszerű dolgunk van, csupán az entrySet tagmetódust hívjuk meg.

Ahhoz, hogy beszélni tudjunk róla itt egy implementációja a Fénykard projectből:

```
std::vector<std::pair<std::string, int>> sort_map ( std::map <std::string, int> &rank )
{
    std::vector<std::pair<std::string, int>> ordered;
    for ( auto & i : rank ) {
        if ( i.second ) {
            std::pair<std::string, int> p {i.first, i.second};
            ordered.push_back ( p );
        }
    }
    std::sort (
        std::begin ( ordered ), std::end ( ordered ),
        [ = ] ( auto && p1, auto && p2 ) {
            return p1.second > p2.second;
        }
    );
    return ordered;
}
```

A rendezni kívánt map rekordjai egy sztring és egy int típusú kulcspárból állnak, tehát a segédvektor ezek kombinációját kell tartalmazni: ezt az std::pair szerkezzettel valósítjuk meg.

Az ordered vektort feltöljtük a map rekordjaival majd megkezdődik az előbbi feladatban ismertetett rendezés.

Ahogy említve is volt, van lehetőségünk referenciként vagy másolatként is elkapni külső változókat, mi itt most explicit megmondjuk hogy az összes elérhető változót szeretnénk felhasználni lambda függvényünkben, viszont csak másolatként(=).

A rendezés minden két rekordot viszgál egyszerre, amelyiknek nagyobb a kulcsa az kerül minden shiftelésre.

17.4. Alternatív Tabella rendezése

Mutassuk be az [alternatív tabella](#) a programban a java.lang Interface Comparable<T> szerepéit!

Újabb rendezéses vizekre evezünk, most már Javában mutatjuk be a custom rendezéseket.

Ebben a feladatban, mivel saját osztályainak szeretnénk rendezni, implementáljuk a Comparable Java interfészét, hiszen ha ráengednénk egy sort függvényt egy olyan adatszerkezetre aminek elemei a mi osztályunkból példányosított példányok nem tudna mit csinálni a JVM, nem tudja hogy ezek az elemtípusok milyen szempont alapján lehetnek összehasonlítva egymással.

Az összehasonlítás alapját nekünk kell magunk kimondani explicit módon.

Ezt az Alternatív Tabella projecten mutatjuk meg, ez magyar focicsapatok egy alternatív rangsorát tartalmazza. Tehát csapatokat fogunk rendezni, ehhez nyílvánvalóan implementálni kell egy olyan mechanizmust, Comparable révén, ami lehetővé teszi az összehasonlítást két példány között.

```
class Csapat implements Comparable<Csapat> {
    protected String nev;
    protected double ertek;
    public Csapat(String nev, double ertek) {
        this.nev = nev;
        this.ertek = ertek;
    }
    public int compareTo(Csapat csapat) {
        if (this.ertek < csapat.ertek) {
            return -1;
        } else if (this.ertek > csapat.ertek) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

A Java nagyon olvasmányos nyelv, így nem szükséges nagyon szókimondó magyarázatra, hogy pontosan mi is történik itten.

A Csapatokat úgy szeretnénk rendezve látni, hogy a legjobb legyen a legvégén a kollekcióban, a legyenébb pedig az elején, tehát növekvő sorrendet szeretnénk.

Ehhez overrideolnunk kell az említett interfész compareTo metódusát.

Ha a vizsgált rekord értéke kisebb mint a másik akkor térjünk vissza -1 értékkel, ha nagyobb akkor +1 értékkel, egyébként nullával.

Ez a három szám egyértelműen megmondja, hogy milyen relációban állnak a vizsgált elemek ez alapján pedig el lehet döntenи melyik a "dominánsabb" példány.

Ha példázé x86-os Assemblyben hasinlítünk össze két számot akkor is pozitív/negatív vagy nulla értékeket kapunk vissza.

17.5. GIMP Scheme hack

Ha az előző félévben nem dolgoztad fel a témat (például a mandalás vagy a króm szöveges dobozosat) akkor itt az alkalom!

Megoldás videó: https://youtu.be/OKdAkI_c7Sc

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

A kódot innen érheted el: [Chrome](#)

Feladatunk végeredménye A GIMP képszerkesztő egy kiterjesztése lesz. Ahhoz, hogy futtassuk a .scm fájlunkat be kell helyezni a GIMP scripts könyvtárába.

Előző feladatban Scheme-ben írtunk szkriptet, most pedig Script-fu szkriptet írunk. A Script-fu egy Scheme dialektuson alapuló nyelv, ami a GNU Image Manipulation program nyelve. A Script-fu segítségével lényegében automatizálni tudunk egy chrome hatású effektet, következő feladatban pedig ennek segítségével fogunk mandalát is készíteni.

Minden egyes GIMP funkció (forgítás, nagyítás, tükrözés, stb.) benne van egy adatbázisban. A Script-fu nyelvben tudjuk használni lekérdezéssel és meghívással ezeket a GIMP függvényeket. Az adatbázis GUI-n keresztül elérhetjük az összes függvényt, illetve böngészhetünk benne. Ezáltal könnyebb a meló.

A forráskód röviden:

```
(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)
)
```

A színek manipulálásához szükséges adatokat itt tároljuk. Ez az eljárás lényegében egy tömböt ad vissza. A "let*" segítségével hozunk létre lokális változókat.

```
; (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)
)
)
```

Egy rekurziós függvényt láthatunk ami kiveszi a megadott indexű elemet a listából. A "car" a lista első elemét adja vissza, itt akkor ha az index egyenlő 1-gyel, ha ez nem teljesül akkor levonunk az "x" értékből 1-t és meghívódik a "cdr" ami a listát adja vissza az első elem nélkül.

Következő rekurziós függvényünk az krómosítani kívánt szöveg szélességét határozza meg, ez kell a rajzoláshoz. Láthatjuk, hogy immár GIMP-s függvényekhez folyamodunk. Meghatározzuk a szöveg szélességét felhasználva a gimp-text-get-extents-fontname első visszatérési értékét, ami a szélesség. Ezt értékül adjuk a text-width változónknak a set! parancsal.

Az előbb megírt keresési eljárást felhasználva pedig megszerezzük a második visszatérési értékét is, ami a magasság.

```
; (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)))
```

Megkezdjük a tényleges krómosítási algoritmus implementálását az előbb említett cikk alapján.

A helyi változók deklarálása és inicializálása után megkezdjük a lépések megvalósítását.

Először létre kell hoznunk egy réteget, nem üres réteget hozunk létre, mert egy általunk definiált Űrlapon megkérjük a felhasználót, hogy adjon meg valamilyen karakterláncot amit szeretne krómosítani néhány paraméter mellett.

Folytatva az első lépését, felhasználva a bekért paramétereket, létrehozunk egy fekete háttérű képet, amin az input szöveg színé fehér lesz. Töröljük a kijelölést, hogy elkerüljük az előreláthatatlan következményeit ha meghagyjuk.

```
;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)
```

Az angolul *Gaussian* szóval megilletett elmosást alkalmazzuk a jelenlegi képre, hogy kisimítsuk a szöveget, tehát ne legyen olyan élesek a betűk.

A 3. lépésben beállítjuk, hogy a szövegek lekanyarítottak legyenek élei a megfelelő színekre vonatkozó konfigurációs paraméterek testreszabásával, majd a 4. lépésben az enyhített elmosást alkalmazunk.

```
;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)
)
```

Invertáljuk a kijelölést, hogy létrehozzunk egy átlátszó réteget, tehát eltűnik a fekete háttér. A szövegre alkalmazzuk a színátmenetet, amit megadott a user, majd alkalmazzuk a BumpMap függvényt rá megfelelő paraméterekkel. A 9. lépésben történik meg a csoda: létrehozunk egy polinomot, egy görbületet(spline), aminek tulajdonságait állítgatva, különböző módon kerül alkalmazva a képre a króm effektus. Végén átméretezzük a képünket, majd megjelenítjük az eredmény a usernek.

```
) ;(script-fu-bhax-chrome-border "Bátf41 Haxor Stream" "Sans" 160 1920 1080 ←
400 '(255 0 0) "Crown molding" 7
;(script-fu-bhax-chrome-border "Programozás" "Sans" 110 768 576 300 '(255 0 ←
0) "Crown molding" 6

(script-fu-register "script-fu-bhax-chrome-border"
"Chrome3-Border2"
"Creates a chrome effect on a given text."
"Norbert Bátfai"
"Copyright 2019, Norbert Bátfai"
"January 19, 2019"
")
```

```
SF-STRING      "Text"      "Bátf41 Haxor"
SF-FONT        "Font"       "Sans"
SF-ADJUSTMENT "Font size" '(160 1 1000 1 10 0 1)
SF-VALUE       "Width"     "1920"
SF-VALUE       "Height"    "1080"
SF-VALUE       "New width" "400"
SF-COLOR      "Color"     '(255 0 0)
SF-GRADIENT   "Gradient"  "Crown molding"
SF-VALUE       "Border size" "7"
)
(script-fu-menu-register "script-fu-bhax-chrome-border"
  "<Image>/File/Create/BHAX"
)
```

Itt látható kód azért kell, hogy a GIMP menüjébe beépíthessek a szkript hívását. Egy űrlapot jelenít meg, ami segítségével bekéri a szükséges paramétereket a felhasználótól. Mi a kódban egy alapértelmezett beállítást is megadunk. Végül megadjuk a szkriptünk elérési útvonalát a gui-n belül az utolsó függvényhívással.



Chrome effekt így néz ki

18. fejezet

Helló, !

18.1. FUTURE tevékenység editor

Javítsunk valamit a ActivityEditor.java JavaFX programon! ([link](#))

Itt láthatjuk működésben az alapotot: [twitch](#)

A program futtatásához szükségünk van a Qt és az OpenCV könyvtárakra. Ezeket beszerezve, Linux alatt a kapott utasításokat követve futtathatjuk a programot.

A feladatunk, hogy javítani kellett a programon. Használat során figyeltem, hogy milyen hibák avagy problémák azok, amik kiküszöbölése egyszerűbbé teheti a program használatát. Észrevettem, hogy mikor egy altevékenységet akarok létrehozni, akkor egy mappán belül csak egyet tudok. Ha többet szeretnék létrehozni, mint egy akkor hibaüzenetet dob.

Az alábbi kódcsipetben megtaláltam azokat a sorokat, amik az új altevékenység létrehozásáért felelősek.

```
java.io.File f = new java.io.File(  
    file.getPath() + System.getProperty("file.separator") + "Új altevékenység ←  
    ");  
if (f.mkdir()) {  
    javafx.scene.control.TreeItem<java.io.File> newAct  
    // rr.println("Cannot create " + f.getPath())rr.println("Cannot create " ←  
    +  
    // f.getPath())rr.println("Cannot create " + f.getPath())rr.println(" ←  
    Cannot  
    // create " + f.getPath()) = new javafx.scene.control.TreeItem<java.io. ←  
    File>(f,  
    // new javafx.scene.image.ImageView(actIcon));  
    = new FileTreeItem(f, new javafx.scene.image.ImageView(actIcon));  
    getTreeItem().getChildren().add(newAct);
```

A `java.io.File.mkdir()` a fájl "elérési útvonalát" kell tartalmaznia a `java.io.File`-nak és ha ezen meghívjuk az `mkdir()` tagfügvényt, létrehozza ezt a mappát. Észre lehet venni miközben átfutjuk a dokumentumot, hogy ha már létezik az elérési útvonalon egy mappa, akkor az `mkdir` nem tud új mappát létrehozni az adott elérési útvonalon ugyanazzal a névvel. Tehát megtaláltuk a lényegében szemantikai hibát, ami azt okozza hogy a program névváltoztatás nélkül próbálja létrehozni új altevékenységekért. Itt kell korrigálnunk, hogy kijavítsuk a felfedezett problémákat és létre tudunk hozni több új mappát.

```
int i = 1;
while (true) {
    java.io.File f = new java.io.File(
        file.getPath() + System.getProperty("file.separator") + "Új ←
        altevékenység" + i);

    if (f.mkdir()) {
        javafx.scene.control.TreeItem<java.io.File> newAct
        // rr.println("Cannot create " + f.getPath())rr.println("Cannot ←
        create " +
        // f.getPath())rr.println("Cannot create " + f.getPath())rr.println ←
        ("Cannot
        // create " + f.getPath()) = new javafx.scene.control.TreeItem<java ←
        .io.File>(f,
        // new javafx.scene.image.ImageView(actIcon));
        = new FileTreeItem(f, new javafx.scene.image.ImageView(actIcon));
        getTreeItem().getChildren().add(newAct);
        break;
    } else {
        i++;
        System.err.println("Cannot create " + f.getPath());
    }
}
});
```

Egy ciklusba helyezve a mappa készítését, szimplán hozzárakjuk az elérési útvonal végére az "i" változónk értékét, amivel azt számoljuk, hogy hányszor próbálkoztunk már mappát létrehozni. Amennyiben a mappa már létezik, újra kezdjük egyel nagyobb értékkel és azt rakjuk a mappa nevének a végére. Amennyiben a mappa sikeresen létrejött a ciklusból egy break utasítás segítségével kiléünk.

18.2. OOCWC Boost ASIO hálózatkezelése

Mutassunk rá a scanf szerepére és használatára! [carlexer.ll](#)

Röviden az OOCWC (robocar World Championship) programról:

Az OOCWC egy nemrégi azaz néhány évvel ezelőtt készített platform. Lényegében forgalomirányítási algoritmusok kutatása és a robotautók terjedésének vizsgálata volt a platform célja. A Robocar City Emulator fejlesztők számára tette lehetővé, hogy új elméleteket és modelleket fejleszthessenek ki illetve teszteljenek. Ami nekünk ebből az egészből kell, az a carlexer és ebből is a `sscanf` függvény amit fel kell dolgoznunk.

A mellékelt fájl helyett, jobban tudjuk érzékelni a `std::sscanf` működését a kliens kód alapján, hiszen pontosan amiatt használjuk amit a fent látunk.

```
while (std::sscanf (data+nn, "<OK %d %u %u %u>%n", &idd, &f, &t, &s, &n) == ←
    4)
{
    nn += n;
    gangsters.push_back (Gangster {idd, f, t, s});
```

{}

A sscanf függvény addig dolgozza fel a formázott stringből az adatokat, amíg meg nem kapja a Gangster négy argumentumát. Alapvetően két részből állnak: egy úgynévezett bufferbol és egy formatból. A buffer egy pointer lesz egy karakterstringre, amiből kiolvásásra kerül majd az adat. A format határozza meg, az adatok hogyan kerüljenek olvasásra. Úgynévezett format-specifikátorokból áll, ezek rendre százalékkel kezdődnek. Esetünkben ilyennel találkozhatunk: <%d %u %u %u>%n. A d az integerekre illeszkedik, míg a u az unsigned integerre, vagyis az elojele nélküli integerekre. Az utolsó, n pedig arra fog szolgálni, hogy számon tartsa a már beolvasott karakterek számát. Az elobb mintáknak megfelelő adatok rendre a & jellet bevezetett változókban kerülnek tárolásra.

18.3. SamuCam

Mutassunk rá a webcam (pl. Androidos mobilod) kezelésére ebben a projektben: [SamuCam](#)

A SamuCam.cpp-ben látható, hogy a videoCapture open függvénye nyitja meg a VideoStream-et. Mivel alapból telefonos használatra íródott a program, ezért a VideoStream helyett 0-t kell írnunk, ha webkamerával szeretnénk dolgozni.

Ezután beállítjuk a kamerakép szélességét, magasságát és FPS értékét. A CascadeClassifier alapvetően tárgyak felismerését segíti, jelen esetben pedig ez teszi lehetővé a kameraképen látható arcok feldolgozását. A helyes működés érdekében le fogunk rántani a GitHub-ról egy frontal face XML-t, melyben az értékek a kamerával szemben elhelyezett arcképek felismerését biztosítja.

Képkockánként kerül beolvasásra a kamerakép a read függvényben. Ha kap egy képkockát, akkor első lépésként átméretezi, majd szürkére átszínezi. Az equalizeHist függvény kiegyenlíti a szürke képkocka hisztogramját.

A detectMultiScale függvény segítségével történik a képkockán lévő arcok keresése. Ha talált egy arcot, akkor az alapján létrehozunk egy QImage-t. A faceChanged egy signal, bekövetkezte után az arc köré rajzolunk egy keretet, és egy újabb QImage-t készítünk. Ha pedig a webcamChanged signal bekövetkezik, akkor 80 ms-t követően következhet egy újabb képkocka beolvasása.

A SamuCam futtatásához először le kell szednünk a GitHub projektet, majd pedig a korábban említett XML fájlt. A Qt keretrendszer segítségével létrehozzuk a Makefile-t, ezután pedig indíthatjuk is a programot.

18.4. BrainB

Mutassuk be a Qt slot-signal mechanizmust ebben a projektben: [esport-talent-search](#)

A BrainB feladata a tehetségtalálatás az esportban, az egyes játékokban előforduló 'karakterelvesztést' fogja előidézni, a karaktervesztés akkor következik be amikor egyszerre annyi minden történik a képernyőn, hogy nem tudjuk már követni hogy hol is van a karakterünk. A program 10 percig fut, ezalatt az idő alatt az a feladatunk hogy a bal egérkombot lenyomva Samu Entropy tartson az egeret. A program futás közben statisztikát készít az eredményeinkről amit a program végeztével megtekinthetünk.

Ehhez a programozh is szükségünk van a Qt és az OpenCV könyvtárakra. A git repó klónozása után qmake-vel lemakeljük a BrainB.pro fájlt. Ez után a programot megnyitva, egy olyan "játékot" kapunk, amiben a Samu nevű entitást kell követnünk az egérrel, közben pedig újak jönnek be a képbe.

A feladatban a slot-signal mechanizmust kell bemutatnunk, ami egy olyan folyamat, amely lehetőséget biztosít különböző eventek összekapsolására, így átláthatóbbá és könnyebben kezelhetővé téve a program háttérbeli feldolgozását.

A BrainBWin.cpp fájlban a BrainBWin funkcióban használjuk a mechanizmust.

```
BrainBWin::BrainBWin ( int w, int h, QWidget *parent ) : ←
    QMainWindow ( parent )
{
    statDir = appName + " " + appVersion + " - " + QDate::currentDate() ←
        .toString() + QString::number ( QDateTime::←
        currentMSecsSinceEpoch() );
    brainBThread = new BrainBThread ( w, h - yshift );
    brainBThread->start();
    connect ( brainBThread, SIGNAL ( heroesChanged ( QImage, int, int ) ) ←
        ),
        this, SLOT ( updateHeroes ( QImage, int, int ) ) );
    connect ( brainBThread, SIGNAL ( endAndStats ( int ) ),
        this, SLOT ( endAndStats ( int ) ) );
}
```

Ez azt jelenti, hogyha meghívjuk például a heroesChanged függvényt, akkor helyette az updateHeroes hívódjon meg.

19. fejezet

Helló, Lauda!

19.1. Port scan

Mutassunk rá ebben a port szkennelő forrásban a kivételkezelés szerepére! tankonyvtar.hu

A kapott linken az alábbi kódot láthatjuk, ami a feladathoz szükséges kódot jelenti.

```
public class KapuSzkenner {
    public static void main(String[] args) {
        for(int i=0; i<1024; ++i)
            try {
                java.net.Socket socket = new java.net.Socket(args[0], i);
                System.out.println(i + " figyeli");
                socket.close();
            } catch (Exception e) {
                System.out.println(i + " nem figyeli");
            }
    }
}
```

Maga a program elég egyszerű, kivételkezelés és socket programozás az alapja az egésznek. Egy for ciklusban zajlik le az egész, a `java.net.Socket()` osztály készít egy adott socketet, majd megpróbálja hozzárendelni annyi porthoz, amennyit megadtunk a ciklusban(ez jelen esetben 0-1023 portokat nézi meg). Ha nem figyeli a gép az adott portot, kapunk egy kivételt, ha nem kapunk kivételt, akkor pedig kiírjuk, hogy az adott porton figyelik az adott IP címet, majd bezárjuk a socketet a `socket.close()` metódussal. Futtatás-kor argumentumnak megadjuk az IP címet, amit tesztelni szeretnénk, de figyeljünk arra, hogy csak saját IP-címen teszteljük, mert egy weboldal könnyen veszélyként kezeli, ha ráengedjük a programunkat

Tanulságok, tapasztalatok, magyarázat...

19.2. AOP

Szój bele egy átszövő vonatkozást az első védési programod Java átiratába! (Sztenderd védési feladat volt korábban.)

Az AOP jelentése, Aspect-oriented programming, azaz askpektus orientált programozás, ezt a nevet azért kapta, mert segítségével különbözo aspektusokból tudjuk megfigyelni, hogy hogyan is viselkedik a kód, "anélkül hogy abba belenyúlnánk. Az OO-hoz képest egy magasabb szintű absztrakciót vezet be. Azért hasznos, mivel nincs szükség a kódunkba belenyúlkálni és ott átírni/beleírni dolgokat, hanem egy külön fájlba megírjuk, majd egy complier segítségével összefüzzük a fo állományunkkal. Futtatás után pedig minden két fájlnak külön-külön olvasható lesz a logja.

A feladat a LZWBInFa Java átiratába kell belefűzni egy AspectJ-s scriptet. Ami a kiir függvényen fog változtatni, még pedig annyiban hogy az inorder (eredeti) kiíratás mellett preorderben is kiírásra kerül a kimenet

Ennek a megvalósításához, létrehozzunk egy pointcut-ot ami itt a feladat miatt a kiir() függvény lesz, és egyben az Aspect számára ezek lesznek a joinpoint-ok. Így az fog történi hogy a call() függvény hívással meghívjuk a kiir függvényt ami már az AspectJ-s kód szerint is le fog zajlani, míg a kiir függvényig minden ugyanúgy lemege az LZWBInFa-ba, majd ezek után az after() függvény segítségével minden az eredeti lefutást mint pedig az AspectJ-s "módosítást", azaz a preorderben kiírást is megejelenítjük. Így tehát eredményképpen az eredeti inorder kiírás mellé preorderben is látható lesz a kimenet, anélkül hogy az eredeti forrásban bármit is módosítottunk volna.

```
package binfa;
import java.io.FileNotFoundException;
import java.io.IOException;
public aspect order {
    int melyseg = 0;
    public pointcut travel(LZWBInFa.Csomopont elem, java.io.PrintWriter os)
        : call(public void LZWBInFa.kiir(LZWBInFa.Csomopont, java.io.←
            PrintWriter)) && args(elem,os);
    after(LZWBInFa.Csomopont elem, java.io.PrintWriter os) throws IOException ←
        : travel(elem, os)
    {
        java.io.PrintWriter kiPre = new java.io.PrintWriter(
            new java.io.BufferedWriter(new java.io.FileWriter("preorder.txt")));
        melyseg = 0;
        preorder(elem, kiPre);
        kiPre.close();
    }
    public void preorder(LZWBInFa.Csomopont elem, java.io.PrintWriter p) {
        if (elem != null) {
            ++melyseg;
            for (int i = 0; i < melyseg; ++i) {
                p.print("---");
            }
            p.print(elem.getBetu());
            p.print("(");
            p.print(melyseg - 1);
            p.println(")");
            preorder(elem.egyesGyermek(), p);
            preorder(elem.nullasGyermek(), p);
            --melyseg;
        }
    }
}
```

```
}
```

A fenti forrás maga az *.aj fájlunk, mint azt fentebb említettem az elején pointcutot létrehozza valamint a call segítségével meghívja a szükséges függvényt, majd meghatározza a kimenetet az after függvény törzsében. Magát az inorderbol preorderbe történő átalakítást pedig a preorder függvényünk végzi, ha a szükséges fájlokkal rendelkezünk (LZWBinFa.java, order.aj, text.txt...), azok fordítása után, s futattása utána megfog jelenni a preorder.txt-nk ami tartalmazni fogja a binfa preorder bejárása szerinti kiírást.

19.3. Android Játék

Írunk egy egyszerű Androidos „játékot”! Építkezzünk például a 2. hét „Hello, Android!” feladatára!

Az első játékhoz nyúlok vissza, amit tanultam programozni, mégpedig a számkitalálós játékhoz. A játék egészen egyszerű. Sorsolunk egy random számot 1 és 100 között amit nekünk ki kell találni. A kitalálás menete a következő. Tippelünk egyet szintén 1 és 100 között és a gép megmondja, hogy a kitalálandó szám kisebb-e vagy nagyobb, mint a tippünk. Aztán lehet folytatni addig amíg el nem találjuk a számot.

Nézzük magát a kódot:

```
<resources>
    <string name="app_name">guessingGame</string>
    <string name="start_msg">Találd ki a számot! (1 és 100 között)</string>
    <string name = "too_high">Kisebb számmal próbálkozz!</string>
    <string name = "too_low">Nagyobb számmal próbálkozz!</string>
</resources>
```

A játékunk egyik legfontosabb alkotó eleme, az az üzenet amelyet a játékos egy-egy tipp után kap. Ezek az üzenetek a strings.xml fájlban kerülnek létrehozásra string formában.

Sok xml kiterjesztésű fájl áll rendelkezésünkre a játékunk kicsinosítása érdekében (ilyen például a colors.xml, dimens.xml) de ezeket most békénhagyjuk.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.ssaurel.higherlower.MainActivity">

    <TextView
        android:id="@+id/msg"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp"
        android:text="@string/start_msg"
        android:textSize="22sp"
        android:textStyle="bold"/>
```

```
<EditText  
    android:id="@+id/numberEnteredEt"  
    android:layout_width="200dp"  
    android:layout_height="wrap_content"  
    android:layout_below="@+id/msg"  
    android:layout_centerHorizontal="true"  
    android:layout_marginTop="50dp"  
    android:inputType="number"/>  
  
<Button  
    android:id="@+id/validate"  
    android:layout_width="200dp"  
    android:layout_height="wrap_content"  
    android:layout_below="@+id/numberEnteredEt"  
    android:layout_centerHorizontal="true"  
    android:layout_marginTop="60dp"  
    android:text="Ellenoriz" />  
</RelativeLayout>
```

Az `activity_main.xml` fájlunkban kerül létrehozásra a felhasználói interfész. A `TextView` lesz felelős a felhasználónak címzett üzenet megjelenítéséért. Az `EditText` teszi lehetővé a játékos számára, hogy egy adott tippet (számot) be tudjon írni. A `Button` biztosítja az ellenorzsere szolgáló gomb működését.

```
package com.example.guessinggame;  
import android.os.Bundle;  
import androidx.appcompat.app.AppCompatActivity;  
import android.view.View;  
import android.widget.Button;  
import android.widget.EditText;  
import android.widget.TextView;  
import android.widget.Toast;  
import java.util.Random;  
  
public class MainActivity extends AppCompatActivity implements View.OnClickListener {  
    public static final int MAX_NUMBER = 100;  
    public static final Random RANDOM = new Random();  
    private TextView msgTv;  
    private EditText numberEnteredEt;  
    private Button validate;  
    private int numberToFind, numberTries;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        msgTv = (TextView) findViewById(R.id.msg);  
        numberEnteredEt = (EditText) findViewById(R.id.numberEnteredEt);  
        validate = (Button) findViewById(R.id.validate);  
        validate.setOnClickListener(this);  
        newGame();  
    }
```

```
    }
    @Override
    public void onClick(View view) {
        if (view == validate) {
            validate();
        }
    }
    private void validate() {
        int n = Integer.parseInt(numberEnteredEt.getText().toString());
        numberTries++;
        if (n == numberToFind) {
            Toast.makeText(this, "Gratulálok! Megvan a szám! " + numberToFind ←
                + Összesen " + numberTries + " próbálkozásra volt szükséged", ←
                Toast.LENGTH_SHORT).show();
            newGame();
        } else if (n > numberToFind) {
            msgTv.setText(R.string.too_high);
        } else if (n < numberToFind) {
            msgTv.setText(R.string.too_low);
        }
    }
    private void newGame() {
        numberToFind = RANDOM.nextInt(MAX_NUMBER) + 1;
        msgTv.setText(R.string.start_msg);
        numberEnteredEt.setText("");
        numberTries = 0;
    }
}
```

Nézzük a MainActivity.java fájlunkat: mindenek előtt importáljuk a szükséges könyvtárakat. Létrehozásra kerülnek a változóink és konstansaink. Deklarálásra kerül az onCreate metódusunk is.

A onClick metódus felel az ellenőrzés gomb működéséért. A validate metódusban fog eltárolódni a felhasználó által bevitt szám n változóban. Egy számláló értékét addig növeljük, ameddig ki nem tudjuk találni a gondolt számot, majd ha kitaláljuk egy gratuláció fogad minket, illetve megtudjuk azt is, hogy hány tippbol tudtuk kitalálni. Meghívásra kerül a newGame metódus, ezzel új játéket kezdve. Ha nem sikerül kitalálnunk a gondolt számot pedig segítséget kapunk a játéktól (ahogy már a feladat elején ki lett fejtve). A newGame metódusban történik a random szám generálása és tárolása. Üzenetet továbbít a játékosnak, és nullázza a számlálót.

19.4. Junit teszt

A [labormeres_oththon_avagy_hogyan_dolgozok_fel_egy_pedat](#) poszt kézzel számított mélységét és szórását dolgozd be egy Junit tesztbe (sztenderd védési feladat volt korábban).

Érdemes lenne azzal kezdeni, mi is a JUnit? Miért is jó nekünk a használata? A JUnit nem más, mint egy keretrendszer amit egységeszteléshez szokás használni Java nyelv mellé. Utóbbi fogalom kis magyarázat-

ra szorul: akkor beszélünk egységesztelésről, amikor egy adott kóddal együtt az adott kódot tesztel „o osztállyal” együtt kerül fejlesztésre.

```
public class BinfaTest {  
    LZWBinFa binfa = new LZWBinFa();  
    @org.junit.Test  
    public void tesBitFeldolg() {  
        for (char c : "01111001001001000111".toCharArray())  
        {  
            binfa.egyBitFeldolg(c);  
        }  
    }  
}
```

AZ elso két sorban semmiféle újdonságot nem találunk. Azonban a harmadik sorban már láthatunk számunkra eddig ismeretlen dolgokat, ezért a kód elemzését kezdjük azzal! A @-cal kezdődő sor jelöli annak a metódusnak a kezdetét, amit a JUnit tesztfuttatójának futtatnia kell. Viszont fontos, ha több ilyen metódus is van, azok végrehajtásának sorrendje nem lesz egyértelmű, ezért úgy érdemes kialakítani ezeket a teszteket, hogy függetlenek legyenek.

Egy teszeset @ annotációval kezdődik, törzsében pedig a tesztelend „o metódus kerül meghívásra és a vég-” rehajtás eredményeként kapott tényleges eredményt az elvárt eredménnyel össze kell vetni

Léteznek egyébként másfajta annotációk is, ezekről itt olvashatunk. Nem olyan rég óta van lehetőségünk parametrizált teszteket is végezni, azonban ez már egy haladóbb tech-” nika, mi nem foglalkozunk vele.

A törzsben található a tesBitFeldolg függvényünk, ami a teszesetünk lesz. Az egyBitFeldolg függvénytel karakterenként dolgozzuk fel a megadott tömböt.

Az assertEquals függvénytel pedig megvizsgáljuk, mennyi is az eltérés - vagyis inkább, hogy megegyezik a két érték - a várható mélység, átlag és szórás valamint ezek tényleges értékei között. A függvény rendre, minden esetben három paraméterrel rendelkezik. Az elso az az érték, amit ’kapunk kellene’, a második a saját programunk által kapott érték, míg a harmadik a ’hibahatár’ lényegében, vagyis hogy mennyi lehet a maximális eltérés az előbb említett két érték között.

20. fejezet

Helló, Calvin!

20.1. MNIST

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel, [hello_samu_a_tensorflow](#)-ból Háttérként ezt vetítsük le: [no-programming-programming](#)

Leegyszerűsítve egy kézzel írott arab számjegyeket tartalmazó adatbázis. Az adatbázis összesen 60000 darab 28x28 pixel méretű, greyscale, PNG képállományt tartalmaz. Ezt az adatbázist (a 60000 képet) két részre tudjuk bontani. Első rész a tanítási, ez 50000 darab képet foglal magába. A maradék 10000 pedig a tesztelési képeket tartalmazza. Az első résznek köszönhetően tanulja meg a gép a számjegyeket, majd a második részben ellenőrzi a tanulásnak a sikerességét.

Fussuk át a kódot:

```
import keras
from keras.datasets import fashion_mnist
from keras.layers import Dense, Activation, Flatten, Conv2D, MaxPooling2D
from keras.models import Sequential
from keras.utils import to_categorical,np_utils
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import os
```

A programunk elején importáljuk a megoldáshoz szükséges könyvtárakat. Ha ezek nem találhatóak meg a gépünkön akkor a `pip install 'library_neve'` parancs segítségével tudjuk letölteni.

```
(train_X,train_Y), (test_X,test_Y) = tf.keras.datasets.mnist.load_data()
```

Lehetővé teszi számunkra az adatbázissal való tevékenykedést. A `load.data()` segítségével töltjük be a több tízezer képállományt amellyekkel a későbbiekben dolgozni fogunk. A betöltött képállományok a vektorban kerülnek tárolásra.

```
train_X = train_X.reshape(-1, 28,28, 1)
test_X = test_X.reshape(-1, 28,28, 1)
```

Következnek azon vektorok elkészítése amelyeket a tanításra és tesztelésre való számok tárolására fogunk használni. 28 db, 28 db elemet tartalmazó vektorra bontjuk az adatainkat. Első paraméterünk -1, magyarul ezt minden tagra eljátszuk.

```
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255
test_X = test_X / 255
```

Annak érdekében, hogy a lehető leggyorsabb tanítási folyamatot elérjük a vektorok típusait átállítjuk, és elosztjuk a megadott számokkal , hogya a képeket alkotó képpontok értéke mostmár megfelelő legyen számunkra.

```
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)

model = Sequential()
```

A one_hot encoding segítségével a modell nem tud kategorikus adatokkal dolgozni. Ennek köszönhetően 0-sok és 1-esek sorozatára fogjuk felbontani az adott számot. Majd beállítjuk a model típusát is.

```
model.add(Conv2D(64, (3,3), input_shape=(28, 28, 1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(64, (3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())
model.add(Dense(64))

model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras. ←
    optimizers.Adam(), metrics=['accuracy'])
```

Az add függvényel tudunk a modellünkhez újabb rétegeket hozzáadni. Első sorban például egy konvolúciós réteget adunk hozzá a modellünkhez. Aminek az első paramétere a neuronok száma , második a detektor, harmadik pedig az input shape, ami jelenleg a 28x28 greyscale állomány lesz. Majd aktiváljuk a ReLu-t (Rectified Linear Unit) azaz a helyesbített lineáris egység. Következő sorban találkozhatunk a pool_size-zal amiben az kerül feldolgozásra, hogy mennyi adat kerüljön feldolgozásra, ennek a két argumentuma van, egy függőleges illetve egy vízszintes érték. Majd a compile függvényel megindítjuk a tanulási folyamatot.

```
model.fit(train_X, train_Y_one_hot, batch_size=64, epochs=1)

test_loss, test_acc = model.evaluate(test_X, test_Y_one_hot)
print('Test loss', test_loss)
print('Test accuracy', test_acc)
```

```
predictions = model.predict(test_X)

print(np.argmax(np.round(predictions[0])))
plt.imshow(test_X[0].reshape(28, 28), cmap = plt.cm.binary)
plt.show()
print(np.argmax(np.round(predictions[1])))
plt.imshow(test_X[1].reshape(28, 28), cmap = plt.cm.binary)
plt.show()
img = Image.open('one.png').convert("L")
img = np.resize(img, (28,28,1))
im2arr = np.array(img)
im2arr = im2arr.reshape(1,28,28,1)
print(np.argmax(np.round(model.predict(im2arr))))
plt.imshow(im2arr[0].reshape(28,28),cmap = plt.cm.binary)
plt.show()
img = Image.open('one.png').convert("L")
img = np.resize(img, (28,28,1))
```

A fit függvény segítségével beállítjuk, hogy mik lesznek a tanítás jellemzői. A batch_size lesz a tanulási sebesség. Az epochs pedig az , hogy a folyamat hányszor kerüljön végrehajtásra. Kiíratásra kerül a tanulási folyamat során keletkező veszteség, illetve a pontosság értéke is. Eltároljuk a prediction-öket. Megjelenítjük az első illetve a második feldolgozott képallokányunkat is a gép általi predictonnal együtt. A harmadik kép a mi képünk lesz a Image.open() függvény segítségével.

20.2. CIFAR-10

Az alap feladat megoldása, +saját fotót is ismerjen fel [link](#)

A CIFAR egy gépi tanulási adatkészlet, ami képek segítségével lehet tanítani, hogy felismerje, hogy minden képen mi található. 32x32 felbontású képekkel dolgozik. Az alap adatszerkezetében több millió kép található, de lehet használni saját képeket is, és így lehet tanítani.

Elsőnek is be kell tanítani a programot a képek felismerésére. Ehez a képekből bináris álmányt kell készíteni, amit a program értelmezni tud. Ezért kell egy olyan algoritmus, ami becsomagolja a képet.

```
rom PIL import Image
import numpy as np
im = Image.open('image.png')
im = (np.array(im))
r = im[:, :, 0].flatten()
g = im[:, :, 1].flatten()
b = im[:, :, 2].flatten()
label = [1]
out = np.array(list(label) + list(r) + list(g) + list(b), np.uint8)
out.tofile("out.bin")
```

A betanítás után jöhet a kép felismerése. Ezt az elkészített bináris álmányra kell elvégezni, úgy hogy a példaprogram cifar10_eval.py fájlt kicsit módosítani kell, hogy ne számoljon pontosságot, mert a feladat szempontjából teljesen lényegtelen, itt csak az kell, hogy melyik csoportba sorolható bele a kép.

```
# while step < num_iter and not coord.should_stop():
# predictions = sess.run([top_k_op])
predictions = sess.run([top_k_op])
print(sess.run(logits[0]))
classification = sess.run(tf.argmax(logits[0], 0))
cifar10classes = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck"]
print(cifar10classes[classification])
true_count += np.sum(predictions)
step += 1
# Compute precision @ 1.
precision = true_count / total_sample_count
# print('%s: precision @ 1 = %.3f' % (datetime.now(), precision))
```

Itt látszik is a módosítás, hogy a pontosság ki van kommentelve, és meg vannak adva a felismeréshez szükséges osztályok. Az ellenőrzés csak egyszer hajtódik végre, majd ahogy az előző MNIST feladatnál is itt is meg kell keresni a legnagyobb indexel rendelkező értéket, majd meghívásra kerül rá a sess.run funkció, így elkezdődik a felismerés és visszatér a megtalált értékkel, utána a cifar10classes tömbből kikérdezi a típusát, majd kiírja.

Mivel csak egy darab képet ismertetünk fel a programmal ezért még a cifar10.py fájlban módosítani kell a batch_size értékét 1-re.

20.3. Android telefonra a TF objektum detektálója

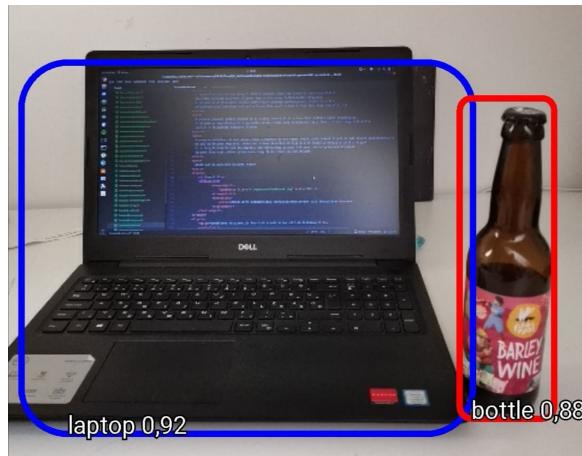
Telepítsük fel, próbáljuk ki!

A Tensorflow-ról röviden annyit lehetne mondani, hogy egy tárgyfelismerő applikáció, ami több-kevesebb sikkerrel felismer egyszerűbb vagy hétköznapiabb tárgyat. A feladat kivitelezéséhez minden adott volt (android oprendszeres telefon és wifi). Egyszerűen csak lekellett tölteni a Tensorflow appot a Google Playről, majd használni azt.

A program működés közben használja az eszköz kameráját és a kamerából látható képet feldolgozza. A folyamatos képfeldolgozás a gyengébb vassal rendelkező eszközökön lassú lehet, hiszen nagy erőforrást igényel a folyamatos képkocka elemzés.

A program működése röviden annyi, hogy a kamerán bejövő képet rövid időközönként elemzi és már ismert objektumokat keres rajta. Ha egy objektumot megtalál, akkor azt színes kerettel körbejelöli és nevén is nevezi a talált tárgyat. Ez a tárgyfelismerés nem túl biztos. Sok hétköznapi tárgyat felismer, és helyesen meg is nevez. Viszont akad azért bőven példa arra, hogy félre ismer egy objektumot.

Alább pár pillanatkép működés közben:



20.1. ábra. Látható, hogy felismer egyszerre akár több tárgyat is



20.2. ábra. Csoda, hogy látok

IV. rész

Irodalomjegyzék

20.4. Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

20.5. C

[KERNIGHANRITCHIE] Kernighan, Brian W. & Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

20.6. C++

[BMECPP] Benedek, Zoltán & Levendovszky, Tihamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

20.7. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf , 2004.

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPORG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEAHCackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésekért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPORG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségen született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.