

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Kiss Marcell

Copyright (C) 2019, Kiss Marcell

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Kiss, Marcell	2020. december 9.	

REVISION HISTORY

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

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	5
2.3. Változók értékének felcserélése	7
2.4. Labdapattogás	7
2.5. Szóhossz és a Linus Torvalds féle BogomIPS	9
2.6. Helló, Google!	9
2.7. 100 éves a Brun tétel	11
2.8. A Monty Hall probléma	11
3. Helló, Chomsky!	13
3.1. Decimálisból unárisba átváltó Turing gép	13
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	13
3.3. Hivatkozási nyelv	13
3.4. Saját lexikális elemző	14
3.5. l33t.1	14
3.6. A források olvasása	17
3.7. Logikus	18
3.8. Deklaráció	18

4. Helló, Caesar!	21
4.1. <code>int **</code> háromszögmátrix	21
4.2. C EXOR titkosító	21
4.3. Java EXOR titkosító	21
4.4. C EXOR törő	22
4.5. Neurális OR, AND és EXOR kapu	25
4.6. Hiba-visszaterjesztéssel perceptron	25
5. Helló, Mandelbrot!	27
5.1. A Mandelbrot halmaz	27
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	28
5.3. Biomorfok	29
5.4. A Mandelbrot halmaz CUDA megvalósítása	31
5.5. Mandelbrot nagyító és utazó C++ nyelven	31
5.6. Mandelbrot nagyító és utazó Java nyelven	31
6. Helló, Welch!	32
6.1. Első osztályom	32
6.2. LZW	33
6.3. Fabejárás	36
6.4. Tag a gyökér	37
6.5. Mutató a gyökér	40
6.6. Mozgató szemantika	41
7. Helló, Conway!	43
7.1. Hangyaszimulációk	43
7.2. Java életjáték	46
7.3. Qt C++ életjáték	46
7.4. BrainB Benchmark	49
8. Helló, Schwarzenegger!	51
8.1. Szoftmax Py MNIST	51
8.2. Szoftmax R MNIST	51
8.3. Mély MNIST	51
8.4. Deep dream	51
8.5. Robotpszichológia	52

9. Helló, Chaitin!	53
9.1. Iteratív és rekurzív faktoriális Lisp-ben	53
9.2. Weizenbaum Eliza programja	53
9.3. Gimp Scheme Script-fu: króm effekt	53
9.4. Gimp Scheme Script-fu: név mandala	53
9.5. Lambda	54
9.6. Omega	54
 III. Második felvonás	 55
10. Helló, Arroway!	57
10.1. OO szemlélet	57
10.2. Homokozó	58
10.3. „Gagyí”	59
10.4. Yoda	60
10.5. Kódolás from scratch	61
 11. Helló, Liskov!	 62
11.1. Liskov helyettesítés sértése	62
11.2. Szülő-gyerek	65
11.3. Anti OO	66
11.4. deprecated - Hello, Android!	66
11.5. Hello, Android!	66
11.6. Hello, SMNIST for Humans!	67
11.7. Ciklomatikus komplexitás	67
 12. Helló, Mandelbrot!	 69
12.1. Reverse engineering UML osztálydiagram	69
12.2. Forward engineering UML osztálydiagram	71
12.3. Egy esettan	74
12.4. BPMN	74
12.5. BPEL Helló, Világ! - egy visszhang folyamat	75
12.6. TeX UML	75

13. Helló, Chomsky!	76
13.1. Encoding	76
13.2. OOCWC lexer	76
13.3. l334d1c45	76
13.4. Full screen	77
13.5. Paszigráfia Rapszódia OpenGL full screen vizualizáció	77
13.6. Paszigráfia Rapszódia LuaLaTeX vizualizáció	78
13.7. Perceptron osztály	78
14. Helló, Stroustrup!	80
14.1. JDK osztályok	80
14.2. Másoló-mozgató szemantika	81
14.3. Hibásan implementált RSA törése	81
14.4. Változó argumentumszámú ctor	84
14.5. Összefoglaló	85
15. Helló, Gödel!	86
15.1. Gengszterek	86
15.2. C++11 Custom Allocator	87
15.3. STL map érték szerinti rendezése	87
15.4. Alternatív Tabella rendezése	88
15.5. Prolog családfa	89
15.6. GIMP Scheme hack	89
16. Helló, !	90
16.1. FUTURE tevékenység editor	90
16.2. OOCWC Boost ASIO hálózatkezelése	90
16.3. SamuCam	92
16.4. BrainB	93
16.5. OSM térképre rajzolása	95
17. Helló, Lauda!	96
17.1. Port scan	96
17.2. AOP	97
17.3. Android Játék	98
17.4. Junit teszt	98
17.5. OSCI	99

18. Helló, Calvin!	100
18.1. MNIST	100
18.2. Deep MNIST	104
18.3. CIFAR-10	105
18.4. Android telefonra a TF objektum detektálója	109
18.5. SMNIST for Machines	114
18.6. Minecraft MALMO-s példa	114
19. Helló, Berners Lee!	115
19.1. C++ és Java	115
19.2. Python	120
IV. Irodalomjegyzék	121
19.3. Általános	122
19.4. C	122
19.5. C++	122
19.6. Lisp	122

Ábrák jegyzéke

11.1. Példa a komplexitásra	68
12.1. 1.	69
12.2. 2.	70
12.3. 3.	70
12.4. 4.	71
12.5. 5.	72
12.6. 6.	72
12.7. 7.	73
12.8. 8.	73
12.9. 9.	74
12.10.10.	75
13.1. 1.	78
14.1. RSA	84
18.1. 1.	102
18.2. 2.	103
18.3. 3.	104
18.4. 4.	106
18.5. 5.	107
18.6. 6.	108
18.7. 7.	108
18.8. 8.	110
18.9. 9.	111
18.10.11.	112
18.11.12.	113

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás videó:

Megoldás forrása:[itt](#)

Tanulságok, tapasztalatok, magyarázat... A végtelen ciklus, olyan ciklus amely, soha nem ér véget, ezzel terhelve a processzort. A 100 százalékos terhelést egy magon a legkönnyebb végrehajtani, ehhez egy sima, egyszerű végtelen ciklusra van szükségünk, amely addig fut folyamatosan míg le nem állítjuk. A 0%-os terheléshez, a végtelen ciklusba szükséges egy "sleep" függvény, ami úgymond "elaltatja" azt a szálát amelyet a végtelen ciklus használna, így a processzor mag terhelése 0%-os lesz. Ahhoz hogy minden magot 100%-on dolgoztasson, szükségünk lesz az OpenMP-re és a "#pragma omp paralell" sorra. Ez röviden több szálon dolgoztatja a programot, így elérve, hogy minden processzormag 100%-osan legyen kihasználva.

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

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

Megoldás videó:

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

```
Program T100
{
    boolean Lefagy(Program P)
    {
        if(P-ben van végtelen ciklus)
            return true;
        else
```

```
    return false;
}

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{

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

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

    main(Input Q)
    {
        Lefagy2(Q)
    }

}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

Tanulságok, tapasztalatok, magyarázat... A T100 egy programot kap meg bemenetként, amiről neki el kell dönteni, hogy az legáll-e, vagy nem. Tehát a T100-as kap egy programot inputként, és megnézi, hogy az általa kapott program kódjában található-e végtelen ciklus. Ha ezt sikerült megállapítania kapunk egy igaz vagy hamis kimenetet. Ezt az outputot kapja meg majd a T1000-es, amely ha a igaz a bemenet, lefagy, ha hamis, akkor pedig bekerül egy végtelen ciklusba. Eddig minden oké, de mi történik akkor, ha a T1000-nek saját magát adjuk oda bemenetként? Ha ő azt látja, hogy van saját magában végtelen cilus, akkor le fog fagyni, ha nincs, akkor viszont bekerül egy végtelen ciklusba. Emiatt az ellentmondás miatt nem lehet ilyen programot írni. Vagy legalábbis eddig még senkinek nem sikerült

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat... Ennek a legegyszerűbb módja, ha matematikai művelettel cseréljük fel a két változó értékét. Én itt összeadás és kivonás segítségével hajtottam ezt végre, de ugyanúgy lehetséges szorzás/osztással is. Az én verzióm lényege, hogy megadom a programnak "a" és "b" értékét, jelen esetben 10 és 5. Majd "a" értékét átírom "a" és "b" összegére így "a" értéke jelenleg 15. Következő lépésben "b" értékének megadom, hogy "a" és "b" különbsége legyen, tehát 15-5, így a "b" értéke már 10, tehát féig megvagyunk. Az utolsó lépés, hogy "a" értékét ismét átíratom a programmal az $a=a-b$ művelettel, azaz 15-10, így a értéke 5 lesz. Szóval a két változó értékét egyszerűen három matematikai művelettel felcseréltem.

2.4. Labdapattogás

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

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

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/labda.c>

Tanulságok, tapasztalatok, magyarázat: Kezdeként include-oljuk szükséges headereket.

```
WINDOW *ablak;  
ablak = initscr ();
```

Ebben a programban, a megjelenítéshez a 'usleep' és a 'clear' a két legfontosabb függvény. Ahhoz hogy a labda pattogni tudjon, tudnunk kell hogy mekkora az ablak mérete, ezt az initscr() függvény használatával tudjuk meg. Ezután kell létrehozunk pár változót, amikben az aktuális pozíciót, az x és y tengelyen lévő lépésközoeket, valamint az ablak méretét tároljuk majd.

```
int x = 0; //x tengelyen lévő jelenlegi pozíció  
int y = 0; //y tengelyen lévő jelenlegi pozíció  
  
int xnov = 1; //x tengelyen lévő lépésköz  
int ynov = 1; //y tengelyen lévő lépésköz  
  
int mx; //ablak szélessége  
int my; //ablak magassága
```

Ezután létrehozunk egy végtelen ciklust, amiben a labdánk 'pattogni' fog. Majd a getmaxyx függvénynek megadjuk az ablakban lévő értékeket, a mvprintw pedig a labdát fogja mozgatni a megadott értékekre.

```
getmaxyx ( ablak, my , mx );  
mvprintw ( y, x, "O" );
```

A labdát mostmár az x és y értékének egyel növelésével tudjuk mozgásra bírni. Ebben játszik szerepet a usleep függvény, mivel ezzel tudjuk állítani, hogy ez milyen gyorsan történjen. (A usleep millisecundumokban számol) A clear függvény pedig törli a labda előző helyzetét és tényleg úgy látjuk mintha pattogna, nem pedig egy csíkot húz maga után. Az ifekkel tudjuk meghatározni azt, hogy az ablak szélénél a labda visszaforduljon, ezt úgy érjük el, hogy a lépésközt -1-el szorozzuk.

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

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Progl/blob/master/szohossz.c>

Tanulságok, tapasztalatok, magyarázat...

```
#include <stdio.h>

int main()
{
    int hossz=0;
    int n=0x01;
    do
    {
        hossz++;
    }while (n<=1);
    printf("Szohossz: %d bit\n",hossz);
    return 0;
}
```

Hasonló programot írtunk tavaly c++-ban. Ez a program úgynevezett shiftelés segítségével dönti el, hogy hány bitből áll a szó. Tehát addig lépeget míg az első szám 0-a nem lesz. Ez az ún. BogoMIPS-es shiftelés módszer. A BogoMIPS egy CPU sebességmérő, amit Linus Torvalds, (Linux kernel atyja) írt meg. Lényegében a program feladata az, hogy leméri mennyi idő alatt fut le, kapunk tőle egy értéket, amely alapján el lehet dönteni, hogy milyen gyors a processzor.

2.6. Helló, Google!

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

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Progl/blob/master/pagerank.c>

Tanulságok, tapasztalatok, magyarázat... A PageRank algoritmust Larry Page és Sergey Brin fejlesztették ki 1998-ban. Ez az algoritmus a mai napig a Google keresőmotorjának a legfontosabb része. Ez egy olyan rendszer, amely arról szól, hogy melyik oldal milyen prioritást élvez, és hogy ezek az page-ek melyik másik page-re mutatnak. Így tudja az algoritmus meghatározni, hogy melyik az a legrelevánsabb oldal amely keresésünknek legjobban megfelel. Ez a kód ezt az algoritmus mutatja be, viszont csak egy 4 weblapos hálózaton. A weboldalak kapcsolatát egy mátrixban tároljuk el.

```
double L[4][4] = {
```

```
{0.0, 0.0, 1.0 / 3.0, 0.0},  
{1.0, 1.0 / 2.0, 1.0 / 3.0, 1.0},  
{0.0, 1.0 / 2.0, 0.0, 0.0},  
{0.0, 0.0, 1.0 / 3.0, 0.0}
```

Ezt a mátrixot megkapja a pagerank függvény argumentumként.

```
void  
pagerank(double T[4][4]){  
    double PR[4] = { 0.0, 0.0, 0.0, 0.0 };  
    double PRv[4] = { 1.0/4.0, 1.0/4.0, 1.0/4.0, 1.0/4.0};  
  
    int i, j;  
  
    for(;;){  
  
        for (i=0; i<4; i++){  
            PR[i]=0.0;  
            for (j=0; j<4; j++){  
                PR[i] = PR[i] + T[i][j]*PRv[j];  
            }  
        }  
  
        if (tavolsag(PR,PRv,4) < 0.0000000001)  
            break;  
  
        for (i=0;i<4; i++){  
            PRv[i]=PR[i];  
        }  
    }  
  
    kiir (PR, 4);  
}
```

A 'PRv' blockban az oldalak első, eredeti értékét tároljuk, a PR blockban pedig a mátrixműveletet tároljuk. Ez a mátrixművelet egy szorzás, amely az 'L' és a 'Prv' block szorzását jelenti. Ennek a szorzásnak az eredményeként kapjuk meg az oldalak pagerank értékét. A 'kiir' függvénnyel íratjuk ki egyenként a weboldalak eredményét.

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

```
}  
}
```

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/Primek_R A Brun tétel azt mondja, hogy az ikerprímek reciprokösszege egy bizonyos összeghez konvergál, szóval közel ér hozzájuk, de soha nem éri el magát a számot. A Tételt Viggo Brun, norvég matematikus dolgozta ki.

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

Tanulságok, tapasztalatok, magyarázat... A Monty Hall probléma alapja egy amerikai tv show (Let's Make a Deal), amely arról szól, hogy a játék végén mutatnak a játékosnak 3 ajtót. A 3 ajtó közül kettő mögött nincs semmi, vagy csak egy értéktelen tárgy van, 1 mögött viszont értékes nyeremény. A játékos választ egy ajtót, de még nem nyitják ki, hanem a műsorvezető kinyitja az egyik olyan ajtót, ami mögött nem az értékes nyeremény van. Ekkora a játékos eldöntheti, hogy szeretne-e változtatni döntésén és kinyitni a másik ajtót, vagy marad a választottjánál. Itt felmerül a kérdés, hogy egyáltalán megéri-e váltani. A válasz meglepő módon igen. Ez azért paradoxon, mert ellentmond a józan paraszti észnek. Mivel elvileg mikor rábökünk egy ajtóra akkor 1/3-ad az esélye, hogy jóra mutattunk, ezen nem változtat ha változtatunk. Vagy mégis? Mikor rámutattunk egy ajtóra még 2/3 valószínűséggel nem volt mögötte semmi, viszont kinyitották az egyik üres ajtót. Innentől biztos, hogy a nyeremény az egyik ajtó mögött van. Tehát így a nyeremény 2/3-ad eséllyel a másik ajtó mögött van. Először meg kell adnunk hogy hány kísérlet lesz. Ezt randommá a "sample" függvénnyel tesszük, amelynek meg kell adni, hogy hánytól hányig generáljon számokat és hogy hányszor tegye ezt. A replace=T pedig megengedi az ismétlődést a számoknál. Az adatokat különböző blokkokban tároljuk. A 'kiserlet' blokkban azt tároljuk, hogy mikor, hová melyik ajtó mögött található a díj, a 'jatekos' blokkban pedig a játékos döntését tároljuk. A 'musorvezeto' függ a nyeremény helyzetétől és hogy a player mit választ. Egy for ciklussal végignézzük az összes játékot, az if segítségével dönti el a műsorvezető, hogy melyik ajtót kell kinyitnia. Az if függvényünk egyik érse azt nézi meg, hogy a játékos, jól választott-e, tehát azt az ajtót, ahol a fődíj van. Ha jól választott a 'mibol' a másik két ajtó egyike lesz, ha viszont nem jól választott, akkor az else azt mondja, hogy azt az ajtót legyen ami mögött nem a nyeremény van és amit nem a játékos választott. Ha ez megvan, ezt az információt műsorvezető megkapja és egy üres ajtót fog kinyitni. Most jön a játékos, hogy akar-e változtatni a döntésén. A 'nemvaltoztatesnyer' akkor vesz csak fel értéket, ha a játékos jól választott és úgy dönt, hogy nem is választ másik ajtót. Ha viszont üres ajtóra mutatott, és úgy határoz hogy változtat, akkor kelleni fog a for ciklus ismét. A 'holvalt' annak

az ajtónak az értéke, se a műsorvezető nem nyitott ki, és a játékos sem válaszotta, majd ezt az értéket veszi át a 'változtat' tömb. Megnézi, hogy a játékos által kiválasztott ajtó és a nyertes ajtó megegyezik-e, és ha igaz értéket kap, akkor a 'valtoztatesnyer' íródik ki. Végül a 'valtoztatesnyer' és a 'nemvaltoztatesnyer' hosszát meg kell néznünk, hogy el tudjuk dönteni, hogy melyik a melyik a nagyobb.

DRAFT

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfiával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/C99.C>

Tanulságok, tapasztalatok, magyarázat...

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

```
int main()
```

```
{  
    for(int a=5; a>10; a++);  
    return 0;  
}
```

C szabvány fejlődésével egyre több funkciót kapott, ám ezek a funkciók nem kompatibilisek visszafelé. A fenti kód például c99-ben lefordul c89-ben viszont nem. Ennek oka, hogy C89-ben még nem lehetett a for ciklusban a ciklusfejben történő ciklusváltozót deklarálni.

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/lex.c>

Tanulságok, tapasztalatok, magyarázat:

A lex forráskódunk 3 részből áll: Az 1. rész: definíciós rész, ahol headereket unclude-olhatunk, változókat deklarálhatunk A 2. rész: Itt a szabályok vannak megadva Ennek két része van, az egyik a reguláris kifejezések, a másik pedig az ezekhez a kifejezésekhez tartozó utasítások A 3. rész: Ez egy c kód Az első és a harmadik rész majd átkerül a generált forrásba is. Tehát az első részben include-oljuk a headereket, és változókat deklarálunk. A második rész, ahogy leírtam a szabályokat tartalmazza.

```
{digit}*(\.{digit}+)? {++realnumbers;  
printf("[realnum=%s %f]", yytext, atof(yytext));}
```

Itt található egy regex, amit előző félévben Operációs rendszerek órán tanultunk. Ez arra az inputokra illeszkedik amik számokkal kezdődnek, ez akárhányszor előfordulhat. (A * ezt jelzi regexben) Majd a zárójeles rész viszont csak 1-szer vagy 1-szer sem fordulhat elő (ezt a ? jelzi) A 3 rész:

```
int  
main ()  
{  
    yylex ();  
    printf("The number of real numbers is %d\n", realnumbers);  
    return 0;  
}
```

Itt hívjuk meg a lexikális függvényt a yylex segítségével, majd egy printf-fel kiíratjuk az eredményt.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/l33t.c>

Tanulságok, tapasztalatok, magyarázat:

A l33t nyelv lényege annyi, hogy a szavakban lévő betűket, valamilyen más karakterekre cseréljük, ezek lehetnek pl számok is akár. Ahogy fentebb láthattuk, itt is három részre oszlik a kódunk. Kezdeként include-oljuk a szükséges headereket, majd define-oljuk a L337SIZE-t, tehát ha valahol hivatkozva lesz rá, akkor a mellette lévő értékeket fogja használni

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

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

Ezután létrehozunk egy 'cipher' nevű struktúrát, amely egy char c-ből és egy char c* pointerből áll.

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

Aztán létrehozuk a l337d1c7 block-ot, amely azt tartalmazza, hogy melyik betűt milyen karakterekkel helyettesíthetünk.

```
} l337d1c7 [] = {

{'a', {"4", "4", "@", "/-\\\"}},
{'b', {"b", "8", "|3", "|\"}},
{'c', {"c", "(", "<", "{"}},
{'d', {"d", "|)", "|]", "|\"}},
{'e', {"3", "3", "3", "3\"}},
{'f', {"f", "|=", "ph", "|#\"}},
{'g', {"g", "6", "[", "[+"}},
{'h', {"h", "4", "|-|", "[-\"}},
{'i', {"1", "1", "|", "!"}},
{'j', {"j", "7", "_|", "_/"}},
{'k', {"k", "|<", "1<", "|{"}},
{'l', {"1", "1", "|", "|_"}},
{'m', {"m", "44", "(V)", "|\\\"/\"}},
{'n', {"n", "|\\\"|", "/\\\"/", "/v\"}},
{'o', {"0", "0", "()", "[\"}},
{'p', {"p", "/o", "|D", "|o\"}},
{'q', {"q", "9", "O_", "(,)"}},
{'r', {"r", "12", "12", "|2\"}},
{'s', {"s", "5", "$", "$\"}},
{'t', {"t", "7", "7", "'|'"}},
{'u', {"u", "|_|", "(_) ", "[_"]}},
```

```

{'v', {"v", "\\/", "\\/", "\\/"}}},
{'w', {"w", "VV", "\\//\\/", "(/\\)"}}},
{'x', {"x", "%", ")(", ")(("}}},
{'y', {"y", "", "", ""}}},
{'z', {"z", "2", "7_", ">_"}}},

{'0', {"D", "0", "D", "0"}}},
{'1', {"I", "I", "L", "L"}}},
{'2', {"Z", "Z", "Z", "e"}}},
{'3', {"E", "E", "E", "E"}}},
{'4', {"h", "h", "A", "A"}}},
{'5', {"S", "S", "S", "S"}}},
{'6', {"b", "b", "G", "G"}}},
{'7', {"T", "T", "j", "j"}}},
{'8', {"X", "X", "X", "X"}}},
{'9', {"g", "g", "j", "j"}}}

```

A következő részben, mivel pontot használunk, így minden kakarkert vizsgálnia kell a programnak. A program megvizsgálja a karaktereket, ha megtalálja a cipher tömbben, akkor generál egy random számot, ami alapján eldönti, hogy az 'l337d1c7' tömbben hányadik karakterrel helyettesítse. Ha nem találja meg akkor az eredeti megakpott karaktert írja ki változatlanul. Mint látjuk ha a random szám kisebb mint 91 akkor az első karakterrel helyettesíti, ha kisebb mint 95, akkor a másodikkal és így tovább.

```

. {

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

            found = 1;
            break;
        }
    }
}

```

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

Az utolsó rész itt is egy C forráskód, amiben a lexelést indítjuk el, ugyanúgy mint az előző programunknál.

```
int
main()
{
    srand(time(NULL)+getpid());
    yylex();
    return 0;
}
```

3.6. A források olvasása

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

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

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



Bugok

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

i.

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

ii.

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

iii.

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

iv.

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

v.

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

vi.

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

vii.

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

viii.

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

Megoldás forrása:

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat:

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

```
 $\$ (\backslash \text{forall } x \backslash \text{exists } y ((x < y) \wedge (y \text{ \textit{prím}})) \wedge (\neg (y \text{ \textit{prím}}))) \leftarrow$   
 $\$$ 
```

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

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

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

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

Tanulságok, tapasztalatok, magyarázat:

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciája
- egészek tömbje
- egészek tömbjének referenciája (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;`
- `int *b = &a;`
- `int &r = a;`
- `int c[5];`
- `int (&tr)[5] = c;`
- `int *d[5];`
- `int *h ();`
- `int *(*l) ();`
- `int (*v (int c)) (int a, int b)`
- `int ((*z) (int)) (int, int);`

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/dek.c>

Tanulságok, tapasztalatok, magyarázat: A programozási nyelvekben az egészeket ábrázoló adattípus az integer (röviden int). A következőket vezetjük be a programban:

- ```
int a;

// Ez az egész változó (int típusú)
```
- ```
int *b = &a;

//ez az egész változóra (a) mutató mutató (*b)
```

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

//Ez az egész típusú változó (a) a referenciája (&r)
- ```
int c[5];
```

//Ez az egészek tömbje (5 elemű)
- ```
int (&tr)[5] = c;
```

//Ez az előző (c) elem referenciája (&tr)
- ```
int *d[5];
```

//A (d) egy tömb, amely 5 darab egészre mutató mutatót ← tartalmaz
- ```
int *h ();
```

//A h függvény az a egészre mutató mutatót visszaadó ← függvény
- ```
int *(*l) ();
```

//Ez egészre mutató mutatót visszaadó függvényre mutató ← mutató
- ```
int (*v (int c)) (int a, int b);
```

//Ez az egészet visszaadó (ami a c) és két egészet kapó ( ← Az a és b) függvényre mutató mutatót (\*v) visszaadó, ← egészet kapó függvény
- ```
int ((*z) (int)) (int, int);
```

//Ez pedig a függvénymutató (az első int előtti ((*z) ← egy egészet visszaadó (legelső 'int' a ((*z) után) ← és két egészet kapó (az utolsó 2 'int') függvényre ← mutató mutatót visszaadó (Ez a sima (*z)), egészet ← kapó függvényre

4. fejezet

Helló, Caesar!

4.1. int ** háromszögmátrix

Megoldás videó:

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/thematic_tutorials/bhax_textbook_IgyNeveldaProgCaesar/tm.c

Tanulságok, tapasztalatok, magyarázat:

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/e.c>

Tanulságok, tapasztalatok, magyarázat: Kezdsnek szokás szerint include-oljuk a szükséges headereket, majd define-oljuk a buffer méretet (256) és a maximum kulcsot (100). A mainben fellelhető 'argc' és 'argv' argumentumokat és azok számát tárolja. Ezután deklarálunk két char típusú tömböt a 'buffer'-t és a 'kulcs'-ot, majd két integer típusú változót a 'kulcs_index'-et és az 'olvasott_bajtok'-at. Aztán deklarálunk kell még egy integert a 'kulcs_meret'-et, aminek az értéke az argv 2. elemének nagysága lesz. Ezután ezt a 'strncpy' függvénnyel átmásoljuk a kulcs változóba. Ha ezzel megvagyunk a következő lépés, hogy beolvassuk a byte-okat, ameddig a bufferben van elég hely neki (ez 256 byte lesz). Amíg az 'i' kisebb mint az elemenként olvasott byte-ok, azokat a (kulcs[kulcs_index])-el kezeljük. Végül kiíratjuk a kapott byte-okat egy fájlba.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/exor.java>

Tanulságok, tapasztalatok, magyarázat: Itt java nyelven írjuk meg ugyan azt az EXOR titkosítót. A java egy objektumorientált nyelv, amely szintaxisát a C-től és a C++-tól örökölte, viszont egyszerűbb objektummodellekkel rendelkezik azoknál. A java alkalmazásokat általában bájt kód formájává alakítják, de lehet natív kódot is készíteni belőle. A program ugyan azt csinálja mint az előző c nyelven íródott program. Megkapjuk az adatokat, amelyeket EXOR használatával byte-onként titkosítunk, majd kiíratjuk őket.

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/t.c>

Tanulságok, tapasztalatok, magyarázat: Kezdeként szokás szerint include-oljuk a megfelelő headereket, majd definiálnunk kell pár függvényt. Az 'atlagos_szohossz' függvény kiszámolja a bemenet átl. szóhosszát.

```
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;
}
```

A tiszta_lehet azt nézi meg, hogy a szöveg amit megejtettünk vele, feltört-e vagy nem. A szöveg általában, akkor tiszta, vagy tört ha benne vannak a 'hogy' 'nem' 'az' és a 'ha'szavak, és vizsgáljuk az átlagos szóhosszt is

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

    double szohossz = atlagos_szohossz (titkos, titkos_meret);

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

Ha ennek nem felel meg a szöveg, akkor nem fogjuk tudni feltörni. Következik az exoros eljárás, ahol megkapjuk a 'kulcs'-ot, a 'kulcs_meret'-et, a 'titkos'-t, a 'tikos_meret'-et és a 'buffer'-t. Ezután lefuttatunk egy for ciklust az összes karakteren.

```
void
exor (const char kulcs[], int kulcs_meret, char titkos[], int ←
      titkos_meret, char *buffer)
{

    int kulcs_index = 0;

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

        buffer[i] = titkos[i] ^ kulcs[kulcs_index];
        kulcs_index = (kulcs_index + 1) % kulcs_meret;
    }
}
```

Aztán következik az exor törés, ami ugyan azokkal az adatokkal dolgozik, mint az előző folyamat.

```
void
exor_tores (const char kulcs[], int kulcs_meret, char titkos[],
            int titkos_meret)
{

    char *buffer;

    if ((buffer = (char *)malloc(sizeof(char)*titkos_meret)) == NULL)
    {
        printf("Memoria (buffer) falióra\n");
        exit(-1);
    }

    exor (kulcs, kulcs_meret, titkos, titkos_meret, buffer);

    if(tiszta_lehet (buffer, titkos_meret))
    {
        printf("Kulcs: [%c%c%c%c%c%c%c%c]\nTiszta szoveg: [%s]\n",
               kulcs[0],kulcs[1],kulcs[2],kulcs[3],kulcs[4],kulcs[5],kulcs ←
               [6],kulcs[7], buffer);
    }

    free(buffer);
}
```

A main függvényben deklarálnunk kell néhány 'char' és egy 'int' típusú változót.

```
int
main (void)
```

```
{
    char kulcs[KULCS_MERET];
    char titkos[MAX_TITKOS];
    char *p = titkos;
    int olvasott_bajtok;
```

Ezután következik a titkos fájl 'berántása' egy while ciklussal, majd egy for ciklussal következik a maradék hely nullázása a titkos bufferben.

```
// titkos fájl berantasa
while ((olvasott_bajtok =
    read (0, (void *) p,
        (p - titkos + OLVASAS_BUFFER <
            MAX_TITKOS) ? OLVASAS_BUFFER : titkos + MAX_TITKOS - p)))
    p += olvasott_bajtok;

// maradék hely nullazasa a titkos bufferben
for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
    titkos[p - titkos + i] = '\\0';
```

Ezután következik a 8 db for ciklus, amelyekkel előállítjuk a kulcsokat, majd minddel megpróbáljuk a törést külön külön. Ha valamelyik működik és megfelel a 'tisztá_lehet' függvénynek, akkor kiírja a helyes kulcsot és a feltört szöveget.

```
// osszes kulcs eloallitasa
//int ii, ji, ki, li, mi, ni, oi, pi; //-5.1-es példa: ezek cikluson kívül definiált változók
#pragma omp parallel for private(kulcs,ii, ji, ki, li, mi, ni, oi, pi)
share(p, titkos)
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)
                                {
                                    //printf("%p/n", kulcs); //-5.2-es példa:kulcámának tömbök számának nyomonkövetése

                                    kulcs[0] = ii;
                                    kulcs[1] = ji;
                                    kulcs[2] = ki;
                                    kulcs[3] = li;
                                    kulcs[4] = mi;
                                    kulcs[5] = ni;
                                    kulcs[6] = oi;
                                    kulcs[7] = pi;
```

```
        exor_tores (kulcs, KULCS_MERET, titkos, ←  
                    p - titkos);  
    }  
  
    return 0;  
}
```

4.5. Neurális OR, AND és EXOR kapu

R

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

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

Tanulságok, tapasztalatok, magyarázat: A neuronok feladata az, hogy valamilyen jeleket továbbítsanak. A neuronoknak 3 rétegük van: - bementi réteg - kimeneti réteg - rejtett réteg A bementi réteg feladata, hogy továbbadja a kapott adatokat a többi résznek. A kimeneti rétegben találhatóak meg a függvények és a kimeneti neuronok. Míg a rejtett részben zajlanak a lényeges folyamatok. A jel továbbítása egy küszöbértéktől függ, ha elérjük ezt a küszöbértéket akkor indul csak el a folyamat. A programban lévő a1 és a2 sorok, azok fix sorok, nem változnak. Az 'OR' 'AND' és 'EXOR' sorok valamilyen logikai művelettel jöttek létre. A különböző sorok utáni 'data.frame' paranccsal data frame-eket hozunk létre, ami lehetővé teszi, hogy a kapott adatokat táblázat formájában tároljuk.

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/SylwerStone-perceptron/main.cpp>

Tanulságok, tapasztalatok, magyarázat: A perceptron a megserséges intelligenciában használt neuron egyik legelterjedtebb változata, ami úgymond egy alakfelismerő gép, amelynek az a feladata, hogy véges számú kísérlet alapján osztályozni tudja a bináris alakzatokat. Itt a mandelbrot halmaz alapján generált kép RGB kódját rakjuk be a perceptron inputjába. A programhoz tehát szükségünk lesz a mandelbrot halmaz által generált png-re, az ml.hpp-re és a main.cpp-re. A main.cpp forráskódjának elején include-oljuk például az ml.hpp, amely a Perceptron osztályt tartalmazza.

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

Ezután létrehozunk egy üres png-t

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

Egy változóban tároljuk el a kép méretét, majd létrehozuk a Perceptron-t, amelyben azt adjuk meg, hogy hány rétegünk legyen (itt 3 lesz), és hogy azokon a rétegeken hány neuron legyen. Az utolsóba 1-et rakunk, mivel ez adja majd az eredményünket

```
int size = png_image.get_width() * png_image.get_height();  
  
Perceptron* p = new Perceptron (3, size, 256, 1);
```

A memóriába másoljuk a for ciklusok segítségével a kép piros pixeleit. Majd a Perceptron osztály operátora adja meg nekünk az eredményt, amit a cout-at kiíratunk

```
for (int i = 0; i<png_image.get_width(); ++i)  
    for (int j = 0; j<png_image.get_height(); ++j)  
        image[i*png_image.get_width() + j] = png_image[i][j].red;  
  
double value = (*p) (image); //ez adja vissza az eredményt  
  
std::cout << value << std::endl; //ezzel íratjuk ki az eredményt
```

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/mandelbrot.cpp>

Tanulságok, tapasztalatok, magyarázat:

A Mandelbrot halmaz felfedezője Benoît Mandelbrot volt, akiről a halmaz a nevét is kapta. A Mandelbrot halmaz elemei a komplex számok. Ha ezeket a komplex számokat ábrázoljuk, a komplex számsíkon, akkor különös formájú alakzatokat kapunk. A fenti c++ programmal tudjuk ezt megtenni, amely egy ábrát készít nekünk. Nézzük meg a programot: Először is include-oljuk a szükséges header fájlokat, aztán meg kell adnunk, hogy melyik fájlba szeretnénk menteni a képet, ha ezt nem adjuk meg kapunk egy hibaüzenetet.

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

int main (int argc, char *argv[])
{
    if (argc != 2) {
        std::cout << "Hasznalat: ./mandelpng fajlnev";
        return -1;
    }
}
```

Ezután megadunk egy értékkészletet a függvénynek, valamint megadjuk hogy milyen magas és széles legyen a képünk. Meg kell adnunk még az iterációs határt is.

```
double a = -2.0, b = .7, c = -1.35, d = 1.35;
int szelesseg = 600, magassag = 600, iteraciosHatar = 1000;
```

Ezután létrehozuk a png fájl-t amibe majd a mandelbrot halmaz kerül berajzolásra

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

Ezután a program végigmegy a koordináta rendszer pontjain

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

int iteracio = 0;
std::cout << "Szamitas";

for (int j=0; j<magassag; ++j) {
    //sor = j;
    for (int k=0; k<szelesseg; ++k) {

        reC = a+k*dx;
        imC = d-j*dy;
        // z_0 = 0 = (reZ, imZ)
        reZ = 0;
        imZ = 0;
        iteracio = 0;

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

            ++iteracio;
        }
    }
}
```

Meg kell adnunk valamilyen szint a pixeleknek, aztán berajzoljuk a kapott Mandelbrot-halmazt abba az üres kép fájlba, amit az elején létrehoztunk.

```
kep.set_pixel(k, j, png::rgb_pixel(255-iteracio%256,
                                     255-iteracio%256, 255-iteracio%256));

    }
    std::cout << "." << std::flush;
}

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat: Ez a program annyiben különbözik az előzőtől, hogy megadhatunk 8 paramétert parancssori argumentumként (Ha nem adjuk meg, akkor az alapértelmezettet fogja használni), illetve itt két változó helyett csupán egyet használunk a komplex számok tároláshoz. Valamint ennél színebb ábrát fogunk kapni mint az előzőnél. Ehhez csupán a complex könyvtárra van szükségünk és máris spóroltunk magunknak 1 változót.

```
// 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;
```

Amit még tud a program, hogy %-ban látjuk a folyamat állapotát.

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

5.3. Biomorfok

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

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

Tanulságok, tapasztalatok, magyarázat:

A Biomorfokat Clifford A. Pickover fedezte fel, a Julia halmaz kutatása alatt. Ugyanis megírt egy programot, az előbb említett halmaz megjelenítésére, ám a programkódban volt valami hiba. Ezáltal a hiba által fedezte fel ezeket az úgynevezett biomorfokat. A Julia halmaz egyébként részhalmaza a Mandelbrot halmaznak, annyi különbséggel, hogy míg a Juliában a "c" konstansként szerepel, addig a Mandelbrotban már változóként.

Nézzük a programot: Include-oljuk kezdésként a szükséges headereket. Itt látjuk, hogy 8 helyett már 10 parancssori argumentumunk van, amiket itt sem kötelező megadni, szimplán az alapértelmezett értékeket fogja használni a program.

```
{
    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 szelesseg magassag n a b c ←
                d reC imC R" << std::endl;
    return -1;
}
```

Ezután hozzuk létre az üres png-t, valamint azt, hogy mekkora lépésközünk legyen.

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

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

Mint mondtam, a c konstansként szerepel, ezért a cc a cikluson kívül van.

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

Ez az a bug, ami miatt létrejött a program:

```
if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
```

Végül ugyan azt csináljuk mint az előző programoknál: beállítjuk a pixelek színét és kiíratjuk 1 fájlba.

```
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;
```

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása:

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

Megoldás forrása:

Megoldás videó:

Megoldás forrása:

5.6. Mandelbrot nagyító és utazó Java nyelven

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 eltérve kiszámolt szám.

Megoldás videó:

Megoldás forrása: c++ :<https://github.com/SylwerStone/Prog1/blob/polargen/polargen.cpp> java: <https://github.com/SylwerStone/Prog1/blob/polargen/polargen.java>

Tanulságok, tapasztalatok, magyarázat... térj ki arra is, hogy a JDK forrásaiban a Sun programozói pont úgy csinálták meg ahogyan te is, azaz az OO nemhogy nem nehéz, hanem éppen természetes neked!

Először include-oljuk a szükséges headereket. Aztán létrehozunk egy 'Polargen'-nek elnevezett osztályt, ezen belül is egy public és egy private részt (a public osztályon kívül is, a private pedig csak osztályon belül elérhető) és megadjuk, hogy még nincs eltárolt szám.

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

```
private:
    bool nincsTarolt;
    double tarolt;
```

A generátor kap egy random seedet, majd a 'kovetkezo' függvény megnézi, hogy van e tárolt szám. Ha nincs akkor létrehoz kettőt, amelyek közül az egyiket elmenti a másikkal pedig return-öl. A másikat akkor adja vissza, ha már volt tárolt szám.

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

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

        tarolt = r * v2;
        nincsTarolt = !nincsTarolt;

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

A program utolsó része pedig legenerál nekünk 10 véletlenszerű számot.

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

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

    return 0;
}
```

6.2. LZW

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

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/z.c>

Tapasztalatok: Az LZW (Lempel-Ziv-Welch) algoritmust, amely egy veszteségmentes tömörítési algoritmus 1984-ben publikálta Terry Welch. (A Abraham Lempel és Jacob Ziv által fejlesztett LZ78 algoritmus továbbfejlesztéseként) Legfőbb felhasználása: Unix Compress programja, Gif, és a PDF tömörítő algoritmus között is szerepel. Az LZW a bemeneti adatokból egy úgynevezett binfát épít, olyan módon hogy végig, hogy megnézi van-e 1-es vagy 0-ás oldal, ha nincs, akkor létrehoz egyet és visszaugrik a gyökérre, ha van akkor vagy az 1-es vagy a 0-ás oldalra lép és addig megy lefelé míg létre tud hozni egy újat. A while ciklus hozza létre a fát, a bemenetet olvasva. Ha az első bit 0, akkor megnézzük hogy van-e 0-ás ág, ha nincs, akkor létre kell hozni egyet, majd visszamegyünk a gyökérre, ha van, akkor a bal oldalra a 0-ra ugrunk. Ha a bemenet 1, akkor ugyan ezt csináljuk ellenkezőleg.

Kód: Először létrehozunk egy struktúrát, amely egy értékből, és annak gyerekeire mutató mutatókból áll.

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

Ezután helyet kell foglalni, a változóknak, és visszakapunk egy pointert, ami a lefoglalt területre mutat. Ha nincs elég memória, akkor error-t kapunk és a program kilép.

```
BINFA_PTR
uj_elem ()
{
    BINFA_PTR p;

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

A main elején hozzuk létre a gyökeret, amit '/'-el jelölünk. Jelenleg nincs gyereke, szóval a pointernek NULL értéket vesznek fel. A fa mutatót pedig a gyökérre állítjuk rá.

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

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

Ezután olvassuk be a bemenetet és itt jön létre a fa. Itt történik az amit fentebb írtam: Megnézzük, hogy a bemenet 1 vagy 0. Ha például 1 és nincs ilyen gyerek, akkor létrehoz egyet, a gyerekei pointerét NULL-RA állítjuk, a fa mutatót pedig visszaállítjuk a gyökerre. Ha viszont már van ilyen gyerek, akkor rálépünk arra és a következő bitet vizsgáljuk.

```
while (read (0, (void *) &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;
        }
    }
}
```

A main következő részében íratjuk ki a fát. A szabadit függvény felszabadítja a lefoglalt memóriát, a kiir függvény pedig inorder módon kiíratja a fát a standard outputra.

```
printf ("\n");
    kiir (gyoker);
    extern int max_melyseg;
    printf ("melyseg=%d", max_melyseg);
    szabadit (gyoker);
}
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;
    }
}

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

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: A preorder és az inorder bejárás közötti különbség annyi, hogy preorderben először a fa gyökerét dolgozzuk fel, majd bejárjuk a fa bal oldalát aztán a jobb oldalát. A postorder eljárásban pedig a preorderrel ellenkezőleg, előbb a fa bal oldalát járjuk be, aztán a jobb oldalát, és végül legutoljára járjuk be a fa gyökerét. Ehhez csak a kiir függvényt kell módosítanunk az előzőhöz képest. A postorder bejárásnál a for ciklust az utolsó helyre raktuk, a két gyerek feldolgozása utánra, így előbb a jobb és bal oldali gyerek, aztán a gyökér kerül feldolgozásra.

```
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-1);
        --melyseg
    }
}

```

Preordernél ellenkezőleg, a for ciklus kerül legelőre, így előbb a gyökér kerül feldolgozásra, aztán a bal és jobb oldali gyerekei.

```

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-1);
        kiir(elem->jobb_egy);
        kiir(elem->bal_nulla);

        --melyseg
    }
}

```

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása: https://progpater.blog.hu/2011/03/31/imadni_fogjatok_a_c_t_egy_emberkent_tiszta_szivbol

A c++-os kód, ugyan azt csinálja mint a C-s változata, csupán leegyszerűsödött maga a kód és ezáltal egyszerűbben olvashatóvá is vált. Először is a struktúrát átírjuk osztályá.

```

class LZWBinFa
{

```

```
public:
LZWBinFa (char b = '/') : betu (b), balNulla (NULL), jobbEgy (NULL)  ←
{
};
~LZWBinFa () {};
```

Ezután jön a bemenet vizsgálata, ami annyiben különbözik a c-s verziótól, hogy van egy operátorunk, amellyel a bemnetet shifteljük a fába. Itt új csomópontnak a 'new'-val tudunk területet foglalni. Tehát ha nincs még 0/1-es csomópontunk a new-val foglalunk neki területet, majd az ujNullasGyermek/ujEgyesGyermek függvény segítségével adjuk a fához.

```
void operator<<(char b)
{
    if (b == '0')
    {
        // van '0'-s gyermeke az aktuális csomópontnak?
        if (!fa->nullasGyermek ()) // ha nincs, csinálunk
        {
            Csomopont *uj = new Csomopont ('0');
            fa->ujNullasGyermek (uj);
            fa = &gyoker;
        }
        else // ha van, arra lépünk
        {
            fa = fa->nullasGyermek ();
        }
    }
    else
    {
        if (!fa->egyedGyermek ())
        {
            Csomopont *uj = new Csomopont ('1');
            fa->ujEgyedGyermek (uj);
            fa = &gyoker;
        }
        else
        {
            fa = fa->egyedGyermek ();
        }
    }
}
```

Nézzük a private részt: A Csomopont értékét a konstruktorban '/'-re állítjuk, a gyermekei pedig 'NULL' értéket kapnak. A nullasGyermek és egyesGyermek a bal és jobb gyerekre mutató pointert adnak vissza. Az 'ujNullasGyermek' és az 'ujEgyedGyermek' a gyermekek pointerét állítja rá a paraméterként adott csomópontra.

```
private:
    class Csomopont
    {
    public:

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

        Csomopont *nullasGyermekek () const
        {
            return balNulla;
        }

        Csomopont *egyenesGyermekek () const
        {
            return jobbEgy;
        }

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

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

        char getBetu () const
        {
            return betu;
        }

    private:

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

Végül nézzük a main-t: Itt a beolvasás történik egy while ciklus segítségével.

```
int
main ()
```

```
{
    char b;
    LZWBinFa gyoker, *fa = &gyoker;

    while (std::cin >> b)
    {
        if (b == '0')
        {
            // van '0'-s gyermeke az aktuális csomópontnak?
            if (!fa->nullasGyermek ()) // ha nincs, csinálunk
            {
                LZWBinFa *uj = new LZWBinFa ('0');
                fa->ujNullasGyermek (uj);
                fa = &gyoker;
            }
            else // ha van, arra lépünk
            {
                fa = fa->nullasGyermek ();
            }
        }
        else
        {
            if (!fa->egyenesGyermek ())
            {
                LZWBinFa *uj = new LZWBinFa ('1');
                fa->ujEgyenesGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->egyenesGyermek ();
            }
        }
    }

    gyoker.kiir ();
    gyoker.szabadit ();

    return 0;
}
```

6.5. Mutató a gyökér

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

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/Prog1/blob/master/z3a7agg.cpp>

A gyökércsomópontot át kell írunk mutatóvá.

```
Csomopont *gyoker;
```

Majd a konstruktorban a fa pointert rá kell állítani a fa gyökerére.

```
LZWBinFa ()  
{  
    gyoker = new Csomopont ( '/' );  
    fa = gyoker;  
}
```

Mivel a gyökér mostmár mutató típusú, így az összes helyen ahol pontokat használtunk, azokat nyilakkal kell felcserélnünk.

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

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása: A másoló szemantika tömören annyi, hogy a kapott bináris fát értékül adja az eredeti fának, lemásolva annak összes értéket. A mozgató szemantika működése: az original fa gyökerét felcseréli annak a fának a gyökerével amelyet értékként megkapunk, és ezeknek a gyerekeit átállítja nullpointerre, hogy az ezután lefutó konstruktor miatt ne törlődjön az eredeti fa.

```
LZWBinFa ( LZWBinFa && regi ) {  
    std::cout << "LZWBinFa move ctor" << std::endl;  
  
    gyoker.ujEgyenesGyermekek ( regi.gyoker.egyenesGyermekek() );  
    gyoker.ujNullasGyermekek ( regi.gyoker.nullasGyermekek() );  
  
    regi.gyoker.ujEgyenesGyermekek ( nullptr );  
    regi.gyoker.ujNullasGyermekek ( nullptr );  
}  
LZWBinFa& operator = (LZWBinFa && regi)
```

```
{
    if (this == &regi)
        return *this;

    gyoker.ujEgyesGyermek ( regi.gyoker.egyesGyermek() );
    gyoker.ujNullasGyermek ( regi.gyoker.nullasGyermek() );

    regi.gyoker.ujEgyesGyermek ( nullptr );
    regi.gyoker.ujNullasGyermek ( nullptr );

    return *this;
}
```

Az `std::move` függvény lényegében nem mozgat semmit, szimplán az átadott argumentumot tesszük vele jobbértékké és kikényszerítjük, hogy a mozgató értékadást használja. Aztán az új fát kiíratjuk

```
LZWBinFa binFa2 = std::move(binFa);

kiFile << binFa2;
kiFile << "depth = " << binFa2.getMelyseg () << std::endl;
kiFile << "mean = " << binFa2.getAtlag () << std::endl;
kiFile << "var = " << binFa2.getSzoras () << std::endl;
```

Az eredeti binfát már nem fogjuk tudni majd kiíratni, mivel annak gyökerét kinulláztuk.

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

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

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

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

Tanulságok, tapasztalatok, magyarázat. Ez a program egy ún. "hangyabojt" szimulál, vagyis a hangyákat és azok útjait. Az "Ant" class a hangya tulajdonságait tartalmazza: Kordináták (x, y), hova tart (dir)

```
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;
```

Az "Antwin" classben az ablak magasságát és szélességét, a hangyákat tartalmazó cellák magasságát szélességét tároljuk. A "keyPressEvent" a gomblenyomásokat kezeli, a "closeEvent" az ablak bezárásáért felelős, a "paintEvent" pedig a hangyák színeit alakítja.

```
class AntWin : public QMainWindow
{
    Q_OBJECT

public:
    AntWin(int width = 100, int height = 75,
           int delay = 120, int numAnts = 100,
           int pheromone = 10, int nbhPheromon = 3,
           int evaporation = 2, int cellDef = 1,
           int min = 2, int max = 50,
           int cellAntMax = 4, QWidget *parent = 0);

    AntThread* antThread;

    void closeEvent ( QCloseEvent *event ) {

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

    void keyPressEvent ( QKeyEvent *event )
    {

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

    }

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

private:

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



```
public slots :  
    void step ( const int &);  
  
};
```

Az AntThread-ben tároljuk például a hangyák számát egy cellában, az evaporation mennyiségét, a pheromonok számát. Itt található meg a run és a finish funkció is. Itt található még az "isRunning" függvény, amely a nevéből következtethetően megnézi, hogy fut-e aztán visszaad egy igaz vagy hamis értéket. Pár privát függvény pl: a "newDir" (Új irány a hangyáknak), a "moveAnts" (Hangyák mozgatása) és a "setPheromone" (Pheromone mennyisége beállítása)

```
class AntThread : public QThread  
{  
    Q_OBJECT  
  
public:  
    AntThread(Ants * ants, int ***grids, int width, int height,  
              int delay, int numAnts, int pheromone, int nbrPheromone,  
              int evaporation, int min, int max, int cellAntMax);  
  
    ~AntThread();  
  
    void run();  
    void finish()  
    {  
        running = false;  
    }  
  
    void pause()  
    {  
        paused = !paused;  
    }  
  
    bool isRunning()  
    {  
        return running;  
    }  
  
private:  
    bool running {true};  
    bool paused {false};  
    Ants* ants;  
    int** numAntsinCells;  
    int min, max;  
    int cellAntMax;  
    int pheromone;  
    int evaporation;  
    int nbrPheromone;
```

```
int ***grids;
int width;
int height;
int gridIdx;
int delay;

void timeDevel();

int newDir(int sor, int oszlop, int vsor, int voszlop);
void detDirs(int irany, int& ifrom, int& ito, int& jfrom, int& jto );
int moveAnts(int **grid, int row, int col, int& retrow, int& retcol, ←
    int);
double sumNbhs(int **grid, int row, int col, int);
void setPheromone(int **grid, int row, int col);

signals:
    void step ( const int &);

};
```

7.2. Java életjáték

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

Megoldás videó:

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

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: https://gitlab.com/kissmarcell98/bhax/tree/master/attention_raising/QT

Tanulságok, tapasztalatok, magyarázat... Az életjáték John Conway nevéhez fűződik, aki a Cambridge egyetem egyik matematikusa volt. A játék egy ún. 'nullszemélyes' játék, tehát a játékos feladata szimp-lán annyiben kimerül, hogy egy kezdőalakzatot megad majd figyeli a történéseket. A "játék" lépéseinek eredményét a számítógép számolja, tehát a 'játékosnak' úgymond semmi teendője nincs a kezdőalakzat kiválasztásán kívül. A játék szabályai: A sejt ha 2 vagy 3 szomszédja van túléli a kört. A sejt, ha kettő-nél kevesebb, vagy háromnál több szomszédja van akkor elpusztul. Új sejt akkor keletkezik, ha egy üres cellának pontosan 3 szomszédja van. A 'sejtablak.h'-ban és a 'sejtablak.cpp'-ben megtalálható a SejtAblak class, ami azért felelős, hogy a programunk kirajzolódjon.

```
#include <QtGui/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();
    // Egy sejt lehet élő
    static const bool ELO = true;
    // vagy halott
    static const bool HALOTT = false;
    void vissza(int racsIndex);

protected:
    // Két rácsot használunk majd, az egyik a sejttér állapotát
    // a t_n, a másik a t_n+1 időpillanatban jellemzi.
    bool ***racsok;
    // Valamelyik rácsra mutat, technikai jellegű, hogy ne kelljen a
    // [2][][]-ból az első dimenziót használni, mert vagy az egyikre
    // állítjuk, vagy a másikra.
    bool **racs;
    // Megmutatja melyik rács az aktuális: [racsIndex][][]
    int racsIndex;
    // Pixelben egy cella adatai.
    int cellaSzelesseg;
    int cellaMagassag;
    // A sejttér nagysága, azaz hányszor hány cella van?
    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;
};
```

A 'sejtszal.cpp' és 'sejtszal.h'-ban pedig a SejtSzal class található meg, ami a szabályokat tartalmazza. Tehát ez alapján jön létre új sejt, hal meg egy, vagy éppen marad életben

```
#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;
    // Megmutatja melyik rács az aktuális: [rácsIndex][][]
    int racsIndex;
    // A sejttér két egymást követő t_n és t_n+1 diszkrét időpillanata
    // közötti valós idő.
    int varakozas;
    void idoFejlodes();
    int szomszedokSzama(bool ***racs,
                       int sor, int oszlop, bool allapot);
    SejtAblak* sejtAblak;
};
```

A sikókilövő pedig szintén a sejtablak.cpp-ben található meg. Itt egyesével rajzolja ki a sejteket a megadott koordinátákra.

```
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;

}
```

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: <https://github.com/nbatfai/esport-talent-search>

Tanulságok, tapasztalatok, magyarázat: A BrainB Benchmark azt teszteli, hogy mennyire tudunk figyelni a karakterünkre, mennyi idő alatt veszítjük el, illetve ha elvesztettük, mennyi idő alatt találjuk meg újból. A

feladat annyi, hogy a kurzort rajta kell tartanunk a saját "karakterünkön", ám közben egyre több másik "new" karakter jelenik meg, így nehezedik egyre jobban a dolgunk. Ha elveszítjük a karaktert, akkor kevesebb új karakter jelenik meg, hogy könnyebb legyen megtalálni az eredetit. A folyamat 10 percig tart, majd a végén megkapod az eredményt. Ezeket az eredményeket összehasonlítva tudjuk összehasonlítani egyes egyének képességeit pl. Esport játékokon.

DRAFT

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.2. Szoftmax R MNIST

R

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.4. Deep dream

Keras

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.5. Robotpszichológia

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Megoldás forrása:

9.2. Weizenbaum Eliza programja

Éleszd fel Weizenbaum Eliza programját!

Megoldás videó:

Megoldás forrása:

9.3. 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

Tanulságok, tapasztalatok, magyarázat...

9.4. Gimp Scheme Script-fu: név mandala

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

Megoldás videó: https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelese_a_scheme_programozasi_nyelv

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

Tanulságok, tapasztalatok, magyarázat...

9.5. Lambda

Hasonlítsd össze a következő programokat!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9.6. Omega

Megoldás videó:

Megoldás forrása:

DRAFT

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.

DRAFT

10. fejezet

Helló, Arroway!

10.1. OO szemlélet

A módosított polártranszformációs normális generátor beprogramozása Java nyelven. Mutassunk rá, hogy a mi természetes saját megoldásunk (az algoritmus egyszerre két normálist állít elő, kell egy példánytag, amely a nem visszaadottat tárolja és egy logikai tag, hogy van-e tárolt vagy futtatni kell az algot.) és az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua.! <https://arato.inf.unideb.hu/batfai.norbert/UDPROG> (16-22 fólia) Ugyanezt írjuk meg C++ nyelven is! (lásd még UDPROG repó: source/labor/polargen)

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/prog2/tree/master/prog2%20feladatok>

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban a polártranszformációs generátor megírása volt a feladat. A szemléltetés a Bátfai Tanrár úr által megírt kódon keresztül történik, hiszen ez a Sun által írt program egy egyszerűbb, könnyebben megérthető változata. A működési elv egyszerű:

```
public class PolárGenerátor {  
  
    boolean nincsTárolt = true;  
    double tárolt;  
  
    public PolárGenerátor() {  
        nincsTárolt = true;  
    }  
}
```

Tehát ebben a részben annyit csinál a program, hogy boolean típusú változó segítségével megnézi, hogy van-e eltárolt normális. Ha nincs akkor a következő módon állítunk elő kettőt:

```
public double következő() {  
    if (nincsTárolt) {  
        double u1, u2, v1, v2, w;  
        do {  
            u1 = Math.random();  
            u2 = Math.random();  
            v1 = 2 * u1 - 1;  
            v2 = 2 * u2 - 1;  
            w = Math.sqrt(u1 * u1 + u2 * u2);  
            if (w > 1) continue;  
            tárolt = (v1 * v2 > 0) ? u1 : u2;  
            nincsTárolt = false;  
        } while (true);  
    }  
    return tárolt;  
}
```

```
v2 = 2 * u2 - 1;
w = v1 * v1 + v2 * v2;
} while (w>1);
double r = Math.sqrt((-2 * Math.log(w)) / w);
tárolt = r * v2;
nincsTárolt = !nincsTárolt;
return r * v1;
} else {
    nincsTárolt = !nincsTárolt;
    return tárolt;
```

Ebből a kettőből egyet eltárolunk és a másikat fogjuk a végrehajtásra felhasználni, vagyis ezzel fog a következő() függvény visszatérni. Végül pedig a mainben használjuk fel a következő függvény-t és íratjuk ki a standard outputra az eredményt.

```
public static void main(String[] args) {
    PolárGenerátor g = new PolárGenerátor ();
    for (int i = 0; i < 10; ++i) {
        System.out.println(g.következő());
    }
```

De a JDK-ban a Sunos megoldás mégiscsak különbözik a miénktől, mégpedig a synchronized public double használatában. Ez annyit tesz csak, hogy a program futását korlátozza egyetlen egy szála.

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

10.2. Homokozó

Adjuk át az első védési programot (LZW binfa) C++ nyelvről Java nyelvre, ugyanúgy működjön! Mutassunk rá, hogy gyakorlatilag a pointereket és referenciákat kell kiírtani és minden máris működik (erre utal a feladat neve, hogy Java-ban minden referencia, nincs választás, hogy mondjuk egy attribútum pointer,

referencia vagy tagként tartalmazott legyen). Miután már áttettük Java nyelvre, tegyük be egy Java Servletbe és a böngészőből GET-es kéréssel (például a böngésző címsorából) kapja meg azt a mintát, amelynek kiszámolja az LZW binfáját!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

10.3. „Gagyi”

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, írd Java példaprogramot mely egyszer végtelen ciklus, más x, t értékekkel meg nem! A példát építsd a JDK Integer.java forrására, hogy a 128-nál inkluzív objektum példányokat poolozza!

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/prog2/tree/master/prog2%20feladatok>

Tanulságok, tapasztalatok, magyarázat...

Ha az integer értékét úgy adjuk meg, hogy az -128 és 127 közé essen akkor nem lesz végtelen ciklusunk. Ez azért van így mert ebben a tartományban az Integer ugyanazt az objektumot fogja felhasználni, csupán más értékek fognak hozzárendelődni. Ezzel azt érjük el, hogy a t!=x -re minden egyes lefutás során hamis értéket fogunk kapni. Ezt például a következő képpen érhetjük el:

```
class Gagyi
{
    public static void main(String[] args)
    {
        Integer t = 127;
        Integer x = 127;

        while(x <= t && x>=t && t != x)
            System.out.println("hop");
    }
}
```

Ha viszont egy nagyon csekély változtatást alkalmazunk a kis kódunkban, máris végtelen ciklust kapunk:

```
public class Gagyi_2 {

    public static void main(String[] args) {

        Integer x = 128;
        Integer t = 128;

        while (x <= t && x >= t && t!=x) {
            System.out.println("hop");
        }
    }
}
```

```
}  
}
```

10.4. Yoda

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A Yoda egy olyan programozási stílus, ahol a kifejezések sorrendét felcseréljük, mégpedig olyan módon, hogy a konstans kifejezés a bal oldalra kerül. Ez nem változtatja meg a program működését. Azért nevezték el Yodaról ezt a módszert mert a Star Wars filmekben ő is így beszélt, tehát nem helyes nyelvtani sorrendben mondta a mondatokat. Ez a módszer nagyon hatásos akkor ha el akarjuk kerülni a nullpointeres hibákat. Ezt a módszert a null pointeres hibák ellen használjuk. Először is nézzünk egy olyan példát ahol nem használjuk a Yoda conditions és ez problémát okoz:

```
class yoda  
{  
    public static void main(String[] args)  
    {  
        String myString = null;  
        if (myString.equals("valami"))  
        {  
            System.out.println("semmi");  
        }  
    }  
}
```

Ha ezt a kis egyszerű csipetet így használnánk akkor `NullPointerException`-et kapunk és leáll. Ha viszont használjuk Yoda mester módszerét akkor a program lefut:

```
class yoda  
{  
    public static void main(String[] args)  
    {  
        String myString = null;  
        if ("valami".equals(myString))  
        {  
            System.out.println("semmi");  
        }  
    }  
}
```


10.5. 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-tanitok/javat/apbs02.html#pi_jegyei (mert ha csak lemásolod, akkor pont az a fejlesztői élmény marad ki, melyet szeretném, ha átélnél).

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/prog2/tree/master/prog2%20feladatok/PiBBP.java>

Tanulságok, tapasztalatok, magyarázat...

A BBP (Bailey-Borwein-Plouffe) algoritmus a Pi hexa jegyeit kiszámoló osztály. Az alapja a BPP formula amit 1995-ben találtak fel. Ez alapján a matematikai képlet alapján működik. Pi tizediktől a tizenötödik hatványáig számolja ki a számokat a program.

11. fejezet

Helló, Liskov!

11.1. Liskov helyettesítés sértése

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

[https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf\(93-99 f3lia\)](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf(93-99%20f%33lia)) (számos p3lda szerepel az elv megs3rt3s3re az UDPROG rep3ban, l3sd pl. source/binom/BatfaiBarki/madarak/)

Megold3s forr3sa: <https://github.com/SylwerStone/prog2/tree/master/prog2%20feladatok/Liskov>

Tanuls3gok, tapasztalatok, magy3r3zat...

A Liskov elv azt mondja ki, hogy minden oszt3lynak helyettes3thet3nek kell lennie a lesz3rmazottaival an3lk3l, hogy a program helyes m3lk3d3se megv3ltozna.

N3zz3k, hogy mi t3rt3nik, ha nem vessz3k a Liskov elvet figyelembe.

```
#include <iostream>

using namespace std;

class Negyszog{
public:
    Negyszog(int a_oldal, int b_oldal, int c_oldal, int d_oldal){
        a = a_oldal;
        b = b_oldal;
        c = c_oldal;
        d = d_oldal;
    }
    int a;
    int b;
    int c;
    int d;
    int kerulet;
    int terulet;
};
```

```
class Negyzet:public Negyszog{
public:
    Negyzet(int a_oldal, int b_oldal, int c_oldal, int d_oldal):Negyszog( ←
        a_oldal, b_oldal, c_oldal, d_oldal){
    }
};

class Teglalap:public Negyzet{
public:
    Teglalap(int a_oldal, int b_oldal, int c_oldal, int d_oldal):Negyzet( ←
        a_oldal, b_oldal, c_oldal, d_oldal){
    }
};

void setKeruletTerulet(Negyzet &negyzet){
    negyzet.kerulet = negyzet.a * 4;
    negyzet.terulet = negyzet.a * negyzet.a;
}

int main()
{
    Negyzet& negyzet = *new Negyzet(5,5,5,5);
    Teglalap& teglalap = *new Teglalap(4,5,4,5);

    setKeruletTerulet(negyzet);
    setKeruletTerulet(teglalap);

    cout << "Téglalap kerülete: " << teglalap.kerulet << ", Négyzet ←
        kerülete: " << negyzet.kerulet << endl;
}
```

Tehát elsőnek van egy Negyszog osztályunk. Ebből származtatjuk a Negyzet osztályt. Ezután viszont geometriai hibát követünk el, mert minden négyzet téglalap, de nem minden téglalap négyzet. A Negyzet osztályból származtatjuk a Teglalap osztályt. Majd definiálunk egy függvényt, mely egy négyzet kerületét és területét számolja ki. Viszont az objektum orientált nyelvekben az ős helyén használható a gyermek is. Vagyis a setKeruletTerulet függvényt meg tudjuk hívni egy Teglalap objektummal is. Ami nyilvánvalóan hibás eredményhez vezet, ha olyan táglalapot adunk át, ami nem négyzet.

Nézzük a Java-s verziót. Leginkább csak szintaktikai különbségek vannak a kettő között.

```
class LiskovSert{
    public static class Négyszög{
        public int a;
        public int b;
        public int c;
        public int d;

        public Négyszög(int a_oldal, int b_oldal, int c_oldal, int d_oldal){
            a = a_oldal;
        }
    }
}
```

```
        b = b_oldal;
        c = c_oldal;
        d = d_oldal;
    }

    public int kerület;
    public int terület;
}

public static class Négyzet extends Négyyszög{
    public Négyzet(int a_oldal, int b_oldal, int c_oldal, int d_oldal){
        super(a_oldal, b_oldal, c_oldal, d_oldal);
    }
}

public static class Téglalap extends Négyzet{
    public Téglalap(int a_oldal, int b_oldal, int c_oldal, int d_oldal){
        super(a_oldal, b_oldal, c_oldal, d_oldal);
    }
}

public static class Program{
    public void setKerületTerület(Négyzet negyzet){
        negyzet.kerület = negyzet.a * 4;
        negyzet.terület = negyzet.a * negyzet.a;
    }
}

public static void main(String[] args){
    Program program = new Program();
    Négyzet negyzet = new Négyzet(5,5,5,5);
    Téglalap teglalap = new Téglalap(4,5,4,5);

    program.setKerületTerület(negyzet);
    program.setKerületTerület(teglalap);

    System.out.println("Téglalap kerülete: " + teglalap.kerület + ", ↵
        Négyzet kerülete: "+ negyzet.kerület);
}
}
```

Annyiban eltér a C++-os forrástól, hogy ebben a setKerületTerület függvény a Program osztálynak egy tagfüggvénye.

11.2. Szülő-gyerek

Írjunk Szülő-gyerek Java és C++ osztálydefiníciót, amelyben demonstrálni tudjuk, hogy az ősön keresztül csak az ős üzenetei küldhetők! https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf (98. fólia)4

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Ennek a feladatnak a lényege az, hogy szemléltetni tudjuk azt a jelenséget, hogy a Szülő-Gyerek kapcsolatban a Szülő mindig érti a gyerek üzenetét viszont fordítva ez nincs így. Hiszen ha valamit a gyerek classban definiálunk akkor azt a szülő nem fogja érteni és elszáll a programunk. Mivel vannak olyan osztályok, metódusok amikkel a szülő rendelkezik és ezeket a gyerek örökölni tudja, viszont ez visszafelé már nem működik.

```
class Osztaly{
    public static void main(String[] args){
        Szulo sz = new Gyerek();
        sz.kiir1();
        sz.kiir2();
    }
}

class Szulo{
    public void kiir1(){
        System.out.println("Szulo");
    }
}

class Gyerek extends Szulo{
    public void kiir2(){
        System.out.println("Gyerek");
    }
}
```

Ennek szemléltetésére szerettem volna ezt a kis csipetet bemutatni. a szülő rendelkezik a kiir1 metódussal amit tud tőle örökölni a gyerek is, de ahogy láthatjuk a kódban a gyereknek van egy kiir 2 metódusa is amit viszont a szülő nem ismer és ez okozza nekünk a galibát. A kiir 2-t nem tudjuk meghívni viszont a kiir 1-et simán tudnánk, fordulna, futna a progí probléma nélkül. Javában minden objektum referencia, és a kötés dinamikus de ezzel nem küldhetjük a gyerek által hozott új üzeneteket, tehát az ősön keresztül csak az ős üzeneteit tudjuk küldeni.

```
#include <iostream>
using namespace std;

class Szulo{
public:
    void kiir1(){
        cout << "Szulo" << endl;
    }
};
```

```
class Gyerek : public Szulo{
public:
    void kiir2(){
        cout << "Gyerek" << endl;
    }
};
int main(){

    Szulo* sz = new Gyerek();
    sz->kiir1();
    sz->kiir2();
}
```

Itt pedig a c++-os verzió látható. Az eredmény ugyanaz lesz mint Javában. A példányosításnál hiába hívjuk a kiir2-t hibát fogunk kapni mint ahogy az Javában is történt.

11.3. Anti OO

A BBP algoritmussal 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/tanitok-javat/apas03.html#id561066>

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/pi-bb-bench>

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban négy különböző programnyelven(c, c++, c# és java) ugyanazt a programot fogjuk futtatni és összevetjük a futási időket. Ez a program az előző fejezetben már taglalt BBP algoritmus ami a a Pi hexadecimális alakjának a 0. pozíciótól számított 10 a hatodikon, hetediken és nyolcadikon db jegyét határozza meg.

11.4. deprecated - Hello, Android!

Élesszük fel a <https://github.com/nbatfai/SamuEntropy/tree/master/cs> projektjeit és vessünk össze néhány egymásra következőt, hogy hogyan változtak a források!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.5. Hello, Android!

Élesszük fel az SMNIST for Humans projektet! <https://gitlab.com/nbatfai/smnist/tree/master/forHumans/SMNIST>
Apró módosításokat eszközölj benne, pl. színvilág.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.6. Hello, SMNIST for Humans!

Fejleszd tovább az SMNIST for Humans projektet SMNIST for Anyone emberre szánt appá! Lásd az smnist2_kutatasi_jegyzokonyv.pdf-ben a részletesebb háttérrel!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.7. 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 főlát)!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A ciklomatikus komplexitás nem más mint egy ún. szoftvermetrika amit 1976-ban talált fel Thomas J. McCabe. Hallhattunk már róla akár McCabe-komplexitás néven is. Képes egy adott forráskód alapján a szoftver komplexitását kiszámolni és ezt konkrétan számunkra értelmezhető számokban ki is tudja fejezni. Az eredményül kapott szám a kiszámítási művelete a gráfelméletre épül. A gráf pontjai és a köztük lévő élek a forrásban lévő elágazások alapján épülnek ki és ezek alapján történik a számítás.

A komplexitás értéke: $M = E - N + 2P$ E: a gráf éleinek száma. N: a gráfban lévő csúcsok száma. P: Az összefüggő komponensek száma A ciklomatikus szám: $M = E - N + P$

A ciklomatikus komplexitás a gráf ciklomatikus száma ami a lehetséges kimeneteket köti össze a bemenetekkel. Tehát a gráf alkotói a függvényben lévő utasítások. Ha egyik utasítás után rögtön végre lehet hajtani a másikat akkor van él közöttük ami abból következik, hogy a lineárisan független útvonalakat a metrika közvetlenül számolja a forrásból.

A komplexitás számítására találtam egy nagyon jó kis oldalt amit segítségül hívtam: <http://www.lizard.ws/>

Try Lizard in Your Browser

.java

Analyse

```
public class PolárGenerátor {  
  
    boolean nincsTárolt = true;  
    double tárolt;  
  
    public PolárGenerátor() {  
        nincsTárolt = true;  
    }  
  
    public double következő() {
```

Code analyzed successfully.

File Type

.java

Token Count

239

NLOC

32

Function Name	NLOC	Complexity	Token #	Parameter #
PolárGenerátor::tor	3	1	11	
PolárGenerátor::ö	19	3	137	
PolárGenerátor::main	6	2	57	

11.1. ábra. Példa a komplexitásra

12. fejezet

Helló, Mandelbrot!

12.1. Reverse engineering UML osztálydiagram

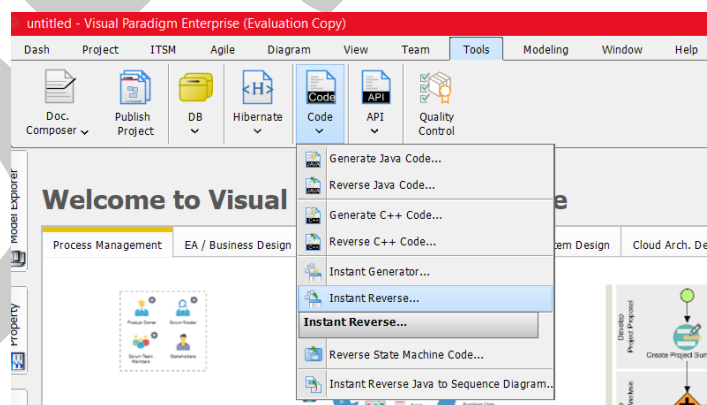
UML osztálydiagram rajzolása az első védési C++ programhoz. Az osztálydiagramot a forrásokból generáljuk (pl. Argo UML, Umbrello, Eclipse UML) Mutassunk rá a kompozíció és aggregáció kapcsolatra a forráskódban és a diagramon, lásd még: https://youtu.be/Td_nlERIEOs. <https://arato.inf.unideb.hu/batfai.norbert/UD> (28-32 fólia)

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/prog2/blob/master/prog2%20feladatok/Binfo.cpp>

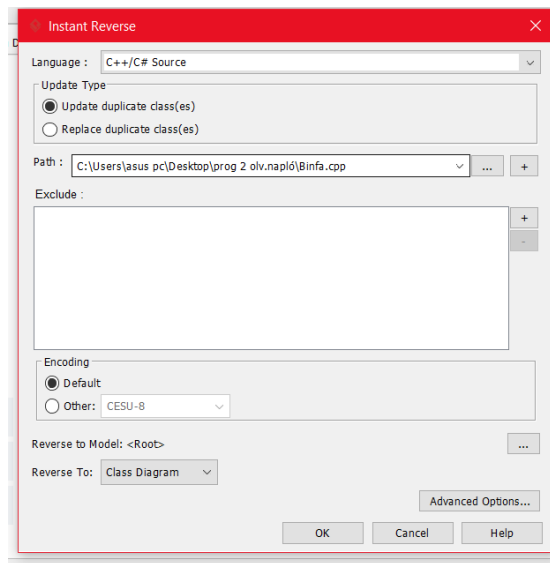
Tanulságok, tapasztalatok, magyarázat...

Itt a C++-os binfát kellett legenerálni egy uml osztálydiagrammba. Ezt a feladatot a Visual Paradigm segítségével oldottam meg. A feladat viszonylag egyszerű volt egy pár kattintással meg lehetett oldani. Először is a visual paradigm-ban kiválasztjuk a tools fület azon belül pedig lenyitjuk a code-ot. A code opció belül pedig kiválasztjuk az instant reverse-t.



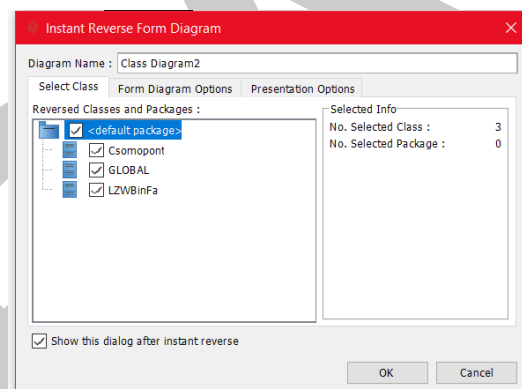
12.1. ábra. 1.

A language-nél megadjuk a c++ forrást, majd megadjuk a kódunk elérési útját és az ok-ra kattintunk, ezzel el is készült az osztálydiagram.



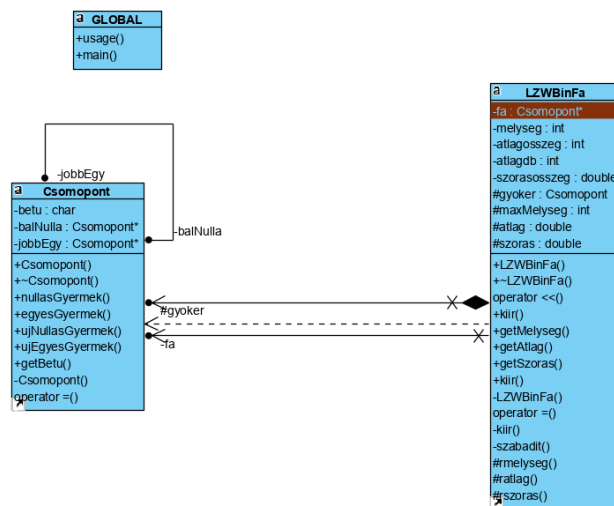
12.2. ábra. 2.

Már csak annyi a teendőnk, hogy a reversed classes and packages ablakban mindent bepipálunk így láthatóvá válik a diagrammunk.



12.3. ábra. 3.

Az importált elemek automatikusan behúzásra kerülnek megfelelő módon.(tehát minden kis vonal oda mutat ahová kell)



12.4. ábra. 4.

12.2. Forward engineering UML osztálydiagram

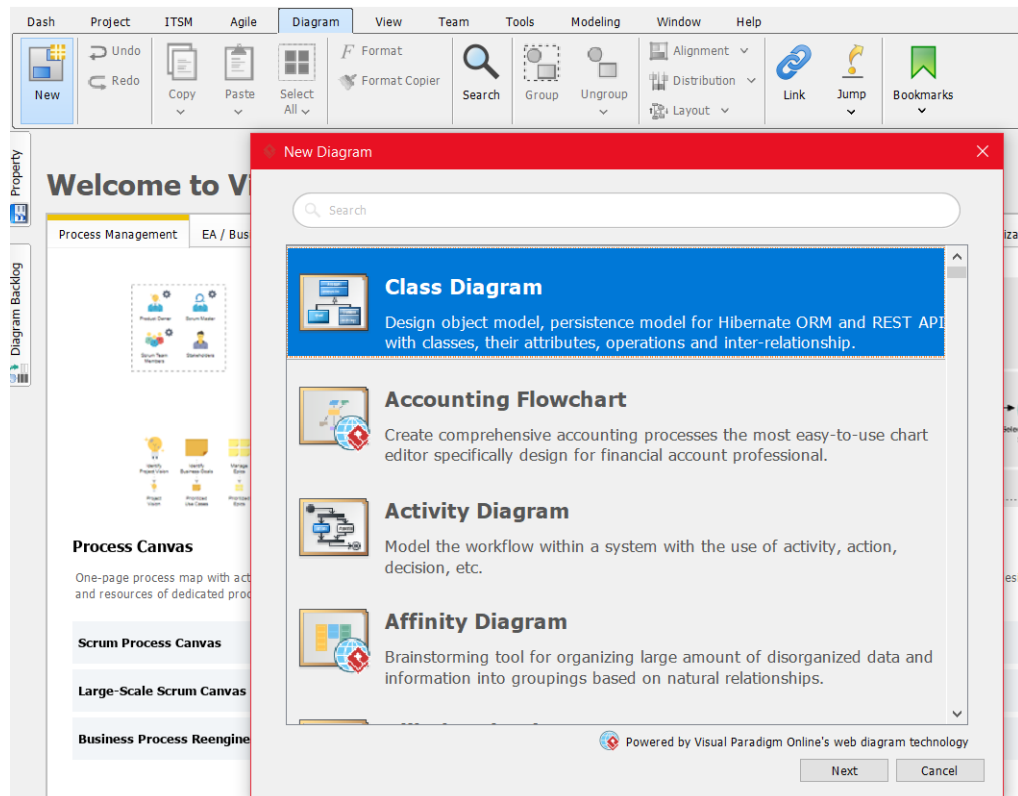
UML-ben tervezzük osztályokat és generáljuk belőle forrást!

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/prog2/tree/master/prog2%20feladatok/forward-UML>

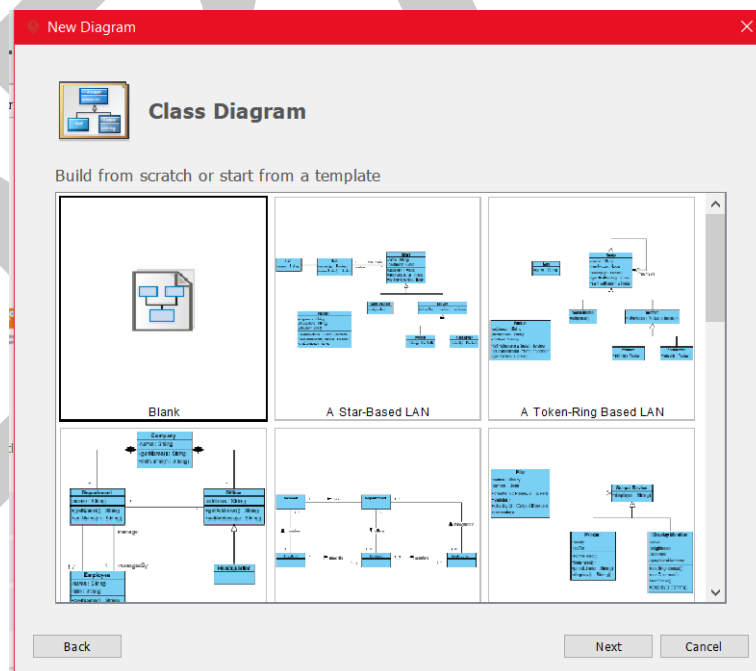
Tanulságok, tapasztalatok, magyarázat...

Ez a feladat ha úgy vesszük akkor az előző fordítottja lenne. Tehát most nekünk kell egy osztálydiagramot készíteni és ebből fogunk majd kódot generálni. Megint a Visual Paradigm-ot használtam a megoldáshoz. Ez a feladat már számomra nehezebb volt mivel még nem csináltam soha osztálydiagramot, de ettől függetlenül összeállítottam egy egyszerűt. Először is a a diagram fülön a new opciót választottam majd kikerestem a class diagramot.



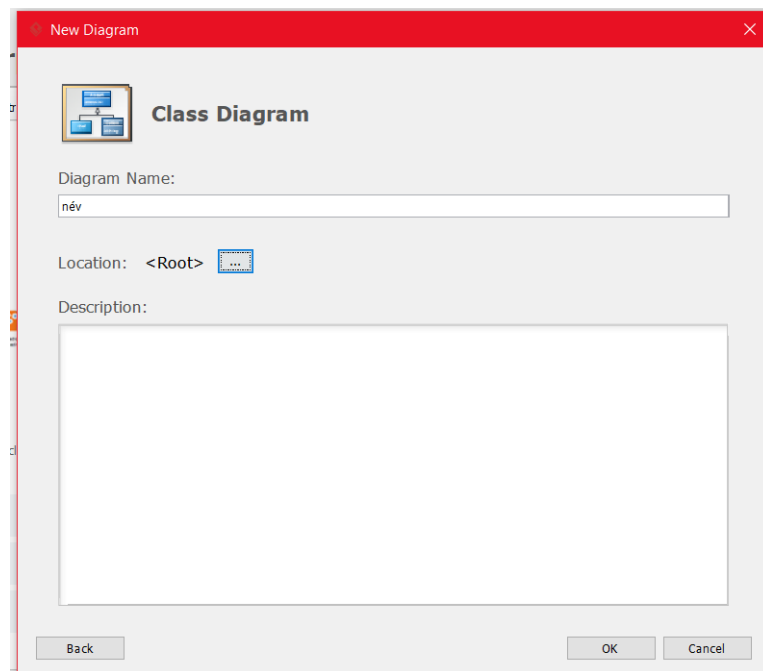
12.5. ábra. 5.

Azon belül pedig a blank opciót



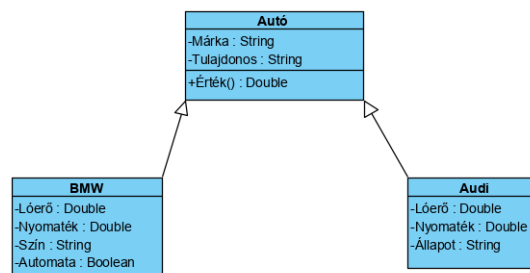
12.6. ábra. 6.

Aztán megadjuk a diagramunk nevét és okéra kattintunk.



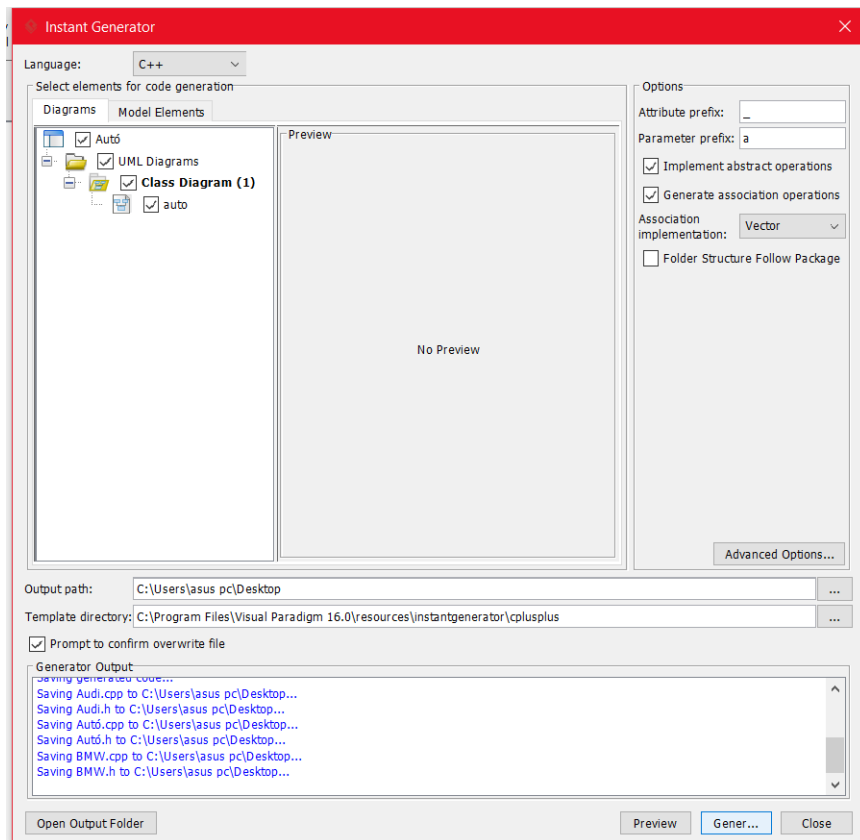
12.7. ábra. 7.

Ezek után kapunk egy üres oldalt ahol kedvünkre hozhatunk létre akármilyen osztálydiagramot. Magunktól kell, hogy létrehozzuk az osztályokat, azok metódusait, változóit, függvényeit, szóval tényleg mindent ami hozzá tartozik. Az enyém így néz ki:



12.8. ábra. 8.

Mostmár pedig már csak annyi dolgunk van, hogy ebből legeneráljuk magát a kódot amit a következőképpen csinálunk: Ugyanúgy mint az előző feladatban kiválasztjuk a tool fülön belüli code opciót de most azon belül az instant generator-re kattintunk. Itt megint bepipáljuk amiot le szeretnénk generálni, az output path opciónál megadjuk neki hova rakja a forrást, generálunk és már készen is vagyunk.



12.9. ábra. 9.

12.3. Egy esettan

A BME-s C++ tankönyv 14. fejezetét (427-444 elmélet, 445-469 az esettan) dolgozzuk fel!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.4. BPMN

Rajzoljunk le egy tevékenységet BPMN-ben! <https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog> (34-47 fólia)

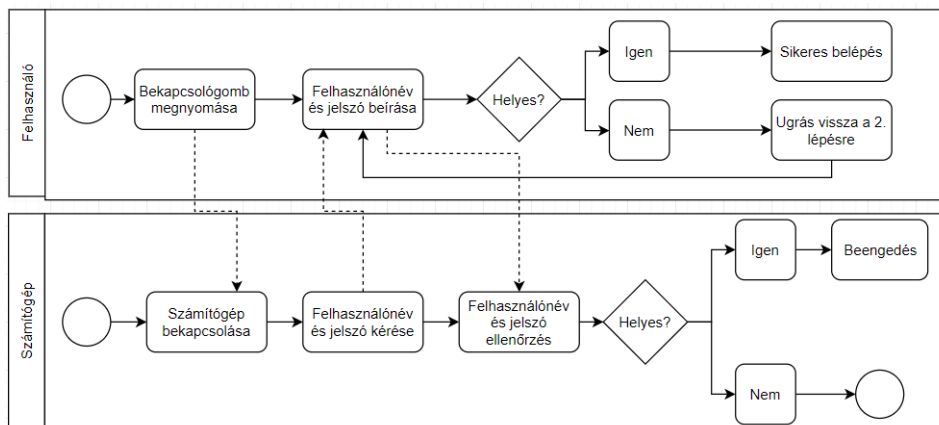
Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A BPMN (Business Process Model and Notation) legfőképpen az üzleti szférában alkalmazott folyamatleíró. Segítségével könnyen és egyszerűen tudunk grafikusán ábrázolni folyamatokat. Ebben a feladatban

ennek a segítségével kellett valami egyszerű tevékenységet ábrázolni. Én úgy döntöttem, hogy ez a számítógép bekapcsolásának és a bejelentkezésnek a folyamata lesz. A draw.io (Google) segítségével oldottam meg a feladatot. Használata elég egyszerű és szinte adja magát, hogy mit hogyan kell csinálni benne. Itt is mindent tudunk ábrázolni szinte amit szeretnénk (pl: event, activity, gateway) Az én verzióm így néz ki:



12.10. ábra. 10.

Véleményem szerint a az ábra értelmezése is meglehetősen egyszerű, csupán ránézésre meg lehet érteni milyen folyamatot ír le és mi történik abban.

12.5. BPEL Helló, Világ! - egy visszhang folyamat

Egy visszhang folyamat megvalósítása az alábbi teljes „videó tutorial” alapján: https://youtu.be/0OnlYWX2v_I

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.6. TeX UML

Valamilyen TeX-es csomag felhasználásával készíts szép diagramokat az OOCWC projektről (pl. use case és class diagramokat).

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13. fejezet

Helló, Chomsky!

13.1. Encoding

Fordítsuk le és futtassuk a Javat tanítók könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezetes betűket! <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/adatok.html>

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/prog2/blob/master/mandel.java/MandelbrotHalmazNagy%C3%A9k%C3%A9z%C3%A9s>

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban az encoding kapcsolót kell használatba vennünk annak érdekében, hogy a magyar ábécében szereplő ékezetes betűket is használhassuk a kódban illetve annak nevében. Mivel alapjáraton a fordítónk UTF-8-as karakterkódolást használ és ezt kell egy számunkra megfelelőre átállítani. Erre szerintem megfelelő lenne a Latin-2-es kódolás (kódja: iso-8859-2) Így már fordul a program gond nélkül.

13.2. OOCWC lexer

Izzítsuk be az OOCWC-t és vázoljuk a <https://github.com/nbatfai/robocaremulator/blob/master/justine/rcemu/src/lexer> és kapcsolását a programunk OO struktúrájába!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.3. l334d1c45

Írj olyan OO Java vagy C++ osztályt, amely leet cipherként működik, azaz megvalósítja ezt a betű helyettesítést: <https://simple.wikipedia.org/wiki/Leet> (Ha ez első részben nem titted meg, akkor írasd ki és magyarázd meg a használt struktúratömb memóiafoglalását!)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.4. Full screen

Készítsünk egy teljes képernyős Java programot! Tipp: https://www.tankonyvtar.hu/en/tartalom/tkt/javat-tanito-k-javat/ch03.html#labirintus_jatek

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.5. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Megoldás videó:

Megoldás forrása: <https://github.com/SylwerStone/prog2/blob/master/prog2%20feladatok/para6.cpp>

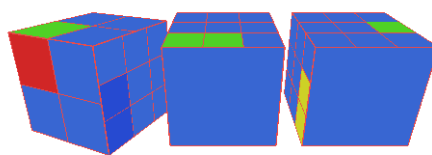
Tanulságok, tapasztalatok, magyarázat...

A program sikeres futtatásához szükséges egy pár könyvtár telepítése ha ezek még nincsenek feltelepítve persze. Ezek a libmesa a freeglut és a boost. A sikeres fordításhoz a `-lboost_system -lGL -lGLU -lglut` kapcsolót kell használatba vennünk. A feladat megoldásához Bátfa Tanár Úr `para6.cpp` nevű forrását használtam fel.

A színeken kellett változtatni illetve meg kellett oldani, hogy a program teljes képernyős módban is futtasson. A `glColor3f` eljárás segítségével tudtam változtatni a színkódokon. 3 értékre van szüksége amely egy rgb színkódot ad ki. $(x * 255)$ Ez fogja a rajzolt elemeket kiszínezni. A vonalak színét a `glBegin (GL_LINES)` meghívása után tudhatjuk, hogy itt a vonalak színezése fog történni.

A feladat másik része pedig az volt, hogy teljes képernyőre kellett rakni a programot. Itt a már létező `keyPress` eventel kellett kicsit variálni. Pontosabban ki kellett egészíteni még két új blokkal. Ezek `f` és `m` névre hallgatnak ebben az iterációban. A működése egyszerű: Ha `f` betűt nyomunk akkor a program teljesképernyős módra vált át, ha pedig ezekután megnyomjuk az `m`-et akkor visszatérünk az eredeti ablakmérethez.

```
} else if (key == 'f') {  
    glutFullScreen();  
} else if (key == 'm') {  
    glutReshapeWindow(640, 480);  
}
```



13.1. ábra. 1.

13.6. Paszigráfia Rapszódia LuaLaTeX vizualizáció

Lásd vis_prel_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, még erősebb 3D-s hatás.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.7. Perceptron osztály

Megoldás forrása: <https://github.com/SylwerStone/prog2/tree/master/prog2%20feladatok/perceptron>

Dolgozzuk be egy külön projektbe a projekt Perceptron osztályát! Lásd <https://youtu.be/XpBnR31BRJY>

A gépi tanulásban nagyon fontos algoritmus a perceptron. Léteznek úgynevezett bináris osztályozók is melyeknek feladata az, hogy eldöntsék, hogy a bemenet egy különleges(specifikus) osztály része-e. Magát a perceptront sokkal jobban nem ecsetelném mivel nagyon összetett dolog és szinte minden információ megtalálható róla pl a Wikipédián is.

Három részből áll a perceptron: van egy olyan rész melyet "retinának" nevezünk, ez az első elem, feladata azon cellák tárolása melyek fogadják az inputot. Aztán maguk a cellák. ezek összegzik a jeleket amik beérkeznek. Végül pedig vannak olyan cellák melyek már a perceptron kimeneti részében vannak, ezek a döntési cellák. A működésük módja hasonló a többi cellához.

Ez a feladat egy az előző félévi perceptronos feladat egyik iterációja. Most ugye azt is meg kell oldani, hogy bemenetként vegyünk egy képet és az lesz a többbrétegű perceptron bemenete.

Mivel többbrétegű perceptront fogunk alkalmazni ezért meg kell tennünk a megfelelő include-okat. Ezek névszerint az mlp és a png kép használata végett a png könyvtárak.

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

Maga a forrás nem túl hosszú. Először is beolvassuk a képet (get_width, get_height) és a new operátorral létrehozuk a perceptront.

```
int main(int argc, char **argv){
png::image <png::rgb_pixel> png_image(argv[1]);
int size = png_image.get_width()*png_image.get_height();
Perceptron *p = new Perceptron(3, size, 256, 1);
```

Aztán létre kell hoznunk egy double változót. Utána szépen végigmegyünk a kép magasság/szélesség pontjain for ciklusokkal. Az image fogja tárolni azt a színkomponenst amit a kódban megadtunk. A value pedig majd azt a double-t fogja tárolni amit a végén kiíratunk.

```
double* image = new double[size];
for(int i = 0; i<png_image.get_width(); ++i)
    for(int j = 0; j<png_image.get_height(); ++j)
        image[i*png_image.get_width()+j] = png_image[i][j].red;
double value = (*p) (image);
cout << value << endl;
```

A végén pedig a delete-el felszabadítjuk a számunkra szükséges helyet a memóriában.

```
delete p;
delete [] image;
```

14. fejezet

Helló, Stroustrup!

14.1. JDK osztályok

Írjunk olyan Boost C++ programot (indulj ki például a fénykardból) amely kilistázza a JDK összes osztályát (miután kicsomagoltuk az src.zip állományt, arra ráengedve)!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Ezt a feladatot a leírásban ajánlott Bátfai Tanár Úr által készített fénykard.cpp alapján készítettem el.

Azért jó kiindulási kód a fénykard mert ebben is állományok kilistázása valamint rekurzív bejárás megvalósítása történik. Ezt a feladatot itt a forrás 72. sorában található read_acts függvény végzi el, viszont nekünk ezen kell változtatnunk, mivel mi a JDK osztályait akarjuk majd kilistázni.

```
void read_acts(boost::filesystem::path path, std::map <std::string, int> & acts)
{
    if (is_regular_file(path)) {

        std::string ext(".props");
        if (!ext.compare(boost::filesystem::extension(path))) {

            std::string actproppath = path.string();
            std::size_t end = actproppath.find_last_of("/");
            std::string act = actproppath.substr(0, end);

            acts[act] = get_points(path);

            std::cout << std::setw(4) << acts[act] << "    " << act << std::endl;
        }
    }
}
```

```
    } else if (is_directory(path))  
        for (boost::filesystem::directory_entry & entry : boost::filesystem::  
            ::directory_iterator(path))  
            read_acts(entry.path(), acts);  
}
```

Ahogy itt láthatjuk a kódcsipetben ez a függvény a .props állományokat listázza ki nekünk. Ezt módosítjuk úgy, hogy a .java állományokkal tegye ezt. Ezt úgy tudjuk megvalósítani, hogy egy kicsit fargunk ebből a függvényből, vagyis az act-okat kitöröljük mivel számunkra azok lényegtelenek. Egy egyszerű push_back-el helyettesítjük, és a .props valamint acts előfordulásokat rendre .java és Classes előfordulásra cseréljük.

Így néz ki a számunkra megfelelő verzió:

```
void readClasses(boost::filesystem::path path, vector<string>& classes){  
    if (is_regular_file(path)){  
        std::string ext(".java");  
        if (!ext.compare(boost::filesystem::extension(path))){  
            classes.push_back(path.string());  
        }  
    }  
    else if (is_directory(path))  
        for (boost::filesystem::directory_entry & entry : boost::filesystem::  
            <- directory_iterator(path))  
            readClasses(entry.path(), classes);  
}
```

És ezekkel a változtatásokkal sikerült elérni a célunkat vagyis, hogy a programunk rekurzívan bejárja a kapott állományt és abból az összes .java-t kilistázza nekünk.

14.2. Másoló-mozgató szemantika

Kódcsipeteken (copy és move ctor és assign) keresztül vedd össze a C++11 másoló és a mozgató szemantikáját, a mozgató konstruktort alapozd a mozgató értékadásra!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.3. Hibásan implementált RSA törése

Készítsünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: https://arato.inf.unideb.hu/-batfai.norbert/UDPROG/deprecated/Prog2_3.pdf (71-73 fólia) által készített titkos szövegen.

Az RSA egy viszonylag régen, 1976-ban kifejlesztett titkosító algoritmus, de ettől függetlenül a mai napon is az egyik leggyakrabban használt. Az egész algoritmus nagyon összetett és bonyolult matematikai háttérrel rendelkezik, és véleményem szerint a feladatunk megoldásához most ez nem is olyan lényeges.

Működési elve viszont annyira nem nehezen érthető szerintem. Van egy titkos és egy nyilvános kulcsunk, a nyilvánossal tudunk titkosítani, és ez publikus valamint a titkos kulccsal tudjuk feloldani a titkosítást. Nekünk most azt kell valahogy megvalósítani, hogy ne működjön ez az algoritmus, tehát ne lehessen a kódolást visszafejteni. Az RSA algoritmust sajnos még nem volt lehetőségem behatóbban megismerni ezért Szilágyi Csaba barátom segítségét vettem igénybe a megoldáshoz. Szokásos módon a fontos könyvtárak importálásával kezdünk. (pl:filereader és a BigInteger)

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.HashMap;
import java.util.Map.Entry;
```

Aztán itt láthatjuk magát a main függvényt.

```
public class rsa_chiper {
    public static void main(String[] args) {
        int bitlength = 2100;

        SecureRandom random = new SecureRandom();

        BigInteger p = BigInteger.probablePrime(bitlength/2, random);
        BigInteger q = BigInteger.probablePrime(bitlength/2, random);

        BigInteger publicKey = new BigInteger("65537");
        BigInteger modulus = p.multiply(q);

        String str = "this is a perfect string".toUpperCase();
        System.out.println("Eredeti: " + str);

        byte[] out = new byte[str.length()];
        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            if (c == ' ')
                out[i] = (byte)c;
            else
                out[i] = new BigInteger(new byte[] { (byte)c }).modPow(publicKey, ←
                    modulus).byteValue();
        }
        String encoded = new String(out);
        System.out.println("Kodolt:" + encoded);

        Decode de = new Decode(encoded);
        System.out.println("Visszafejtett: " + de.getDecoded());
    }
}
```

Mainben fog történni a szöveg letitkosítása valamint itt jön létre a kulcs is. A fő osztályunk itt az rsa_chiper lesz. Itt kell majd megadni azt a szöveget is amit szeretnénk titkossá tenni, ezt fogja majd megpróbálni

visszafejtteni a programunk. És ezután látható majd a feladat megvalósításának egyik fontos lépése: a betűről betűre való titkosítás megvalósítása.

```
byte[] out = new byte[str.length()];
for (int i = 0; i < str.length(); i++) {
    char c = str.charAt(i);
    if (c == ' ')
        out[i] = (byte)c;
    else
        out[i] = new BigInteger(new byte[] { (byte)c }).modPow(publicKey, ←
            modulus).byteValue();
}
```

A titkosított szöveget úgy állítja elő a program, hogy a byte-okból emberek számára "fogyaszthatatlan" humbug szöveget csinál.

Az algoritmus a betűk gyakoriságát figyeli és aszerint helyettesíti be a karaktereket ezért kell egy lista amelyben ez az információ szerepel. Érdekes a listát a használt emberi nyelven belüli betűgyakoriság alapján beállítani.

```
private void loadFreqList() {
    BufferedReader reader;
    try {
        reader = new BufferedReader(new FileReader("freq.txt"));
        String line;
        while((line = reader.readLine()) != null) {
            String[] args = line.split("\\t");
            char c = args[0].charAt(0);
            int num = Integer.parseInt(args[1]);
            this.charRank.put(c, num);
        }
    } catch (Exception e) {
        System.out.println("Error when loading list -> " + e.getMessage());
    }
}
```

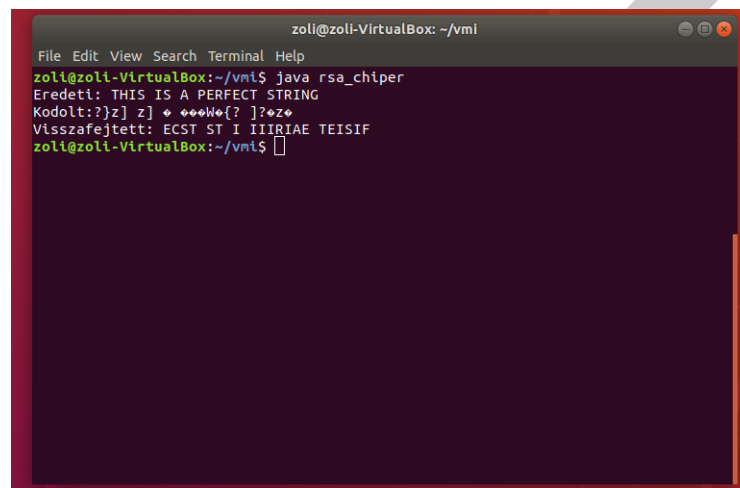
Aztán szükségünk van egy olyan függvényre ami ezt a listát beolvassa és ha előfordulást észlel akkor belerakja a listába 1-es kezdőértékkel, ha már benne van akkor pedig növelni ezt a számot egyel. Csak és kizárólag a betűket számolja, tehát mondjuk a space-t nem veszi figyelembe. Maga a vizsgálat azért fontos, hogy megtudjuk melyik betűnek van a legnagyobb értéke bagyis melyik szerepel a legtöbbször, és ez lesz a prioritásunk.

Végül a programunk meghívja a nextFreq függvényt, ez a listából behelyettesíti a karaktereket,(ún. max. kiválasztásos módszer alkalmazásával.) és ezeket ki is veszi a listából így az végül üres lesz. A visszafejtés pontossága nem mindig kielégítő de legtöbb esetben az eredeti szöveghez hasonlót kapunk vissza. A sikeresség nagyban, vagyis szinte teljesen az elkészített listától függ.

```
private char nextFreq() {
    char c = 0;
    int nowFreq = 0;
    for(Entry<Character, Integer> e : this.charRank.entrySet()) {
        if (e.getValue() > nowFreq) {
```

```
        nowFreq = e.getValue();  
        c = e.getKey();  
    }  
}  
if (this.charRank.containsKey(c))  
    this.charRank.remove(c);  
return c;  
}
```

Légvégül pedig a `getDecode` visszaadja nekünk a szöveget amit a program visszafejtett.



14.1. ábra. RSA

14.4. 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.)

A feladatot az előző csokorban szereplő perceptronos feladat alapján csináltam meg. A forrás nagy része így megegyezik.

Mivel többretegű perceptront fogunk alkalmazni ezért meg kell tennünk a megfelelő include-okat. Ezek névszerint az mlp és a png kép használata végett a png könyvtárak.

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

Maga a forrás nem túl hosszú. Először is beolvassuk a képet (`get_width`, `get_height`) és a `new` operátorral létrehozuk a perceptront.

```
int main(int argc, char **argv){  
    png::image <png::rgb_pixel> png_image(argv[1]);
```



```
int size = png_image.get_width()*png_image.get_height();
Perceptron *p = new Perceptron(3, size, 256, 1);
```

Aztán létre kell hoznunk egy double változót. Utána szépen végigmegyünk a kép magasság/szélesség pontjain for ciklusokkal. Az image fogja tárolni azt a színkomponenst amit a kódban megadtunk. A value pedig majd azt a double-t fogja tárolni amit a végén kiíratunk.

```
double* image = new double[size];
for(int i = 0; i<png_image.get_width(); ++i)
    for(int j = 0; j<png_image.get_height(); ++j)
        image[i*png_image.get_width()+j] = png_image[i][j].red;
double value = (*p) (image);
cout << value << endl;
```

Ezek után jöhetnek a változtatások az eredeti perceptronos feladathoz képest. Most azt szeretnénk, hogy egy képet generáljon nekünk a program nem pedig értékeket írjon. Ezt végrehajtandó implementálunk két új for ciklust az előzőekkel megfelelő céllal valamint double*-ot fogunk használni a sima double helyett. Ezek azért szükségesek, hogy a megfelelő végeredményt kapjuk, vagyis a képünk megfelelő adatokat kapjon a sikeres generáláshoz és a write png kiterjesztésű képet alkosson nekünk.

```
double* newPicture = (*p) (image);
for(int i=0; i<png_image.get_width(); ++i)
for(int j=0; j<png_image.get_height(); ++j)
    png_image[i][j].red = newPicture[i*png_image.get_width()+j];
png_image.write("output.png");
```

A kódunk végén pedig elvégezzük a szükséges hely felszabadítást a memóriában, amit a következő képpen teszünk:

```
delete p;
delete [] image;
```

Ahhoz, hogy minden probléma nélkül fusson már csak annyit kell tennünk, hogy az mlp.hpp header fájlban is átírjuk a double-t double*-ra, mivel azt szeretnénk, hogy visszaadja.

```
double* operator() ( double image [] )
```

14.5. Összefoglaló

Az előző 4 feladat egyikéről írd egy 1 oldalas bemutató „esszé szöveget!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15. fejezet

Helló, Gödel!

15.1. Gengszterek

Gengszterek rendezése lambdával a Robotautó Világbajnokságban <https://youtu.be/DL6iQwPx1Yw>

A gengszterek rendezéséhez ebben a feladatban az `std::sort()` nevezetű függvényt hívjuk segítségül, amely három paramétert vár: az első és második paraméterek azt fogják meghatározni, hogy a vektort (vagy éppenséggel a tömböt) melyik értéktől meddig akarjuk majd mi rendezni, a harmadik pedig az lesz, hogy mi abban a rendezés feltétele, mi alapján akarjuk ezt a rendezést megvalósítani. Ez a feladatból eredően nem lesz más mint a lambda, amivel most meg kell, hogy ismerkedjünk egy kicsit:

```
[ ] (paraméterek) -> visszatérés típusa {utasítások}
```

Maga a felépítés az előbb leírtak alapján könnyen értelmezhető. Először megadjuk azokat a változókat amiket el szeretnénk érni a szögletes zárójelek "[]" közé. (ezek függvényen kívüliek) aztán pedig rendre a paramétereket majd a visszatérést. A felépítése a következő: [] jelek közé a függvényen kívüli változókat kell megadni amiket el szeretnénk érni, aztán a paramétereket kell megadnunk, majd a visszatérést. Jelen esetben ez így fog kinézni:

```
std::sort (gangsters.begin(), gangsters.end(), [this, cop] (Gangster x, ←  
    Gangster y,) {  
    return dst (cop, x.to) < dst (cop, y.to);  
    }  
);
```

Mivel mi itt most az összes gengsztert vizsgálat alá akarjuk vetni a függvényünk által ezért kezdőértékünk a gengszterek eleje, végértékünk pedig annak a vége lesz. Ezt a `begin` és `end` függvényekkel egyszerűen megoldhatjuk. Ezután pedig ugye jön a rendezés feltétele ami a lambda. Ez pedig így néz ki jobban megtekintve:

A függvényen kívüli változóink az aktuális objektum (vagyis a `this`) és a `cop` lesz majd. A gangster `x` és `y` lesznek majd a paramétereink. Maga a vizsgálat pedig a rendőröktől való távolság alapján fog történni. Eszerint fogjuk rendezni a kis gengsztereinket. Ezt így implementáljuk a lambda-ba:

```
return dst (cop, x.to) < dst (cop, y.to);
```

Ezzel egyszerűen sorba tudjuk rendezni a gengsztereket, a "sor" legelején állnak majd a rendőrhöz legközelebb állók és szépen sorban a végére érve pedig a legtávolabbi lesz. A rendezésünk a lambda segítségével biztosan pontos eredményt fog adni nekünk, amely egy "növekvő" sorrendű rendezés lesz majd.

15.2. C++11 Custom Allocator

<https://prezi.com/jvvbytkwgsxj/high-level-programming-languages-2-c11-allocators/> a CustomAlloc-os példa, lásd C forrást az UDPROG repóban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.3. STL map érték szerinti rendezése

Például: <https://github.com/nbatfai/future/blob/master/cs/F9F2/fenykard.cpp#L180>

Megoldás forrása: <https://github.com/SylwerStone/prog2/blob/master/prog2%20feladatok/STL.cpp>

Az STL (Standard Template Library)tárolók és adatszerkezetek összessége amelyek hatékonyan, biztonságosan, kivételbiztosan és típushelyesen képesek tárolni az adatokat. Itt található például a map is ami a feladatunk része lesz.

A mapok a következőképpen épülnek fel. A mapok is tárolók lesznek, az itt tárolt elemeknek két értéke van az adat és a kulcs, és ezek alapján vannak az elemek sorba rendezve. A map egyik implementációja itt látható ebben a csipetekben:

Legelső lépésünk az lesz, hogy megalkotjuk a sort_map-et, ennek return értékei olyan vectorpárok lesznek melynek típusai eltérnek, vagyis az egyik egy string a másik pedig egy int lesz majd. Ez szerencsére nem okoz semmilyenféle fennakadást mivel itt van nekünk az std::pair függvény ami képes ezt kezelni. Az std::map referenciát adjuk majd meg paraméterként a függvénynek.

```
std::vector<std::pair<std::string, int>> sort_map ( std::map <std::string, ←  
    int> &rank )
```

A függvényünk return értéke egy általunk létrehozott "üres" vector lesz amiben majd a párok lesznek.

```
std::vector<std::pair<std::string, int>> ordered;
```

Ezután valahogy meg kell oldanunk, hogy bejárjuk az egész rank map-ünket és kikeressük a párokat belőle. Ezt egy for ciklus segítségével oldhatjuk meg. Egy if-et használunk benne ami szépen megnézi, hogy egy értéknek van-e második értéke vagyis párja. Ha talál ilyet akkor elkészíti az std::pair szerkezetet és beleteszi megfelelő helyre az első és második értékeket. A párokat amelyek itt létrehoz pedig szépen eltárolja az általunk létrehozott eddig üres vectorba.

```
for ( auto & i : rank ) {  
    if ( i.second ) {  
        std::pair<std::string, int> p {i.first, i.second};
```

```
        ordered.push_back ( p );  
    }  
}
```

Legvégül pedig a gengszteres feladatban megismert módszerrel szépen rendezzük az ordert lambda és sort segítségével.

```
std::sort (std::begin ( ordered ), std::end ( ordered ),  
[ = ] ( auto && p1, auto && p2 ) {  
    return p1.second > p2.second;  
})  
);
```

Az egész vectorunkat rendezni szeretnénk ezért megint a legesőtől az utolsóig adjuk meg a paramétereket valamint a harmadik megint a lambda lesz. Külső függvényt most nem hívunk meg csupán azt adjuk meg, hogy másolással vegyük át a változókat, ezt egy egyszerű "=" segítségével érjük el. Paramétereink auto típusúak a return érték pedig p1 és a p2 összehasonlítása adja meg majd nekünk. Ezután pedig a már rendezett vectort visszadjuk:

```
return ordered;
```

15.4. Alternatív Tabella rendezése

Mutassuk be a https://progpater.blog.hu/2011/03/11/alternativ_tabella a programban a java.lang Interface ComparableT szerepét!

Megoldás forrása: <https://github.com/SylwerStone/prog2/tree/master/tabella>

A feladatunkban azt kell megoldanunk, hogy a labdarúgó bajnokságban (jelen esetben az NB1-ben) ne aszokásos módon menjen a pontozás, hanem azt is figyelembe kell, hogy vegyük, hogy ki kivel játszott. Vagyis az a győzelem például többet ér amit egy erősebb csapat ellen szerez egy gyengébb csapat mint fordítva. Ez a rendszer már ismerős, mivel a PageRank is ez az elv alapján működik. Ezt implementáljuk majd be a kis alternatív táblánkban.

```
Interface Comparable<T>
```

A felhasználó által definiált típusokat a Java Comparable segítségével rendezzük. A java.lang-ban találjuk meg ezt az interfészt. A compareTo(Object) lesz majd az a metódus amit mi használni fogunk, illetve ez az egyetlen metódus, itt fogjuk majd definiálni a rendezésünket.

A feladathoz forrásként szereplő Wiki2Matrix.java lesz a kiindulási alap, vagyis ebben fogunk majd változtatásokat véghezvinni, hogy nekünk jó legyen. Itt vannak az adatok amelyek a bajnoksághoz tartoznak. Ezek a következőképpen vannak implementálva: piros mező=3p, sárga mező=1p, zöld mező=0p, illetve vannak üres mezők is, ezek azért vannak mert ugye egy csapat nem játszhat önmagával és az nem számít pontnak, illetve mérkőzésnek. Ha minden adat jól van megadva és fordítjuk, futatjuk a programot akkor kapunk egy ún. linkmátrixot, ezt kell majd az AlternativTabella.java-ba beírunk. Ebben a forrásban kell majd változtatásokat eszközölnünk mivel a megadott forrás nem up to date.

Miután megtörténtek a módosítások, vagyis már az aktuális bajnokság tabelláját és mérkőzéseit adtuk meg mehet a fordítás, futtatás. Itt láthatjuk majd, hogy a táblánk eltér az eredetitől, ez a kvázi "PageRank" alkalmazása miatt áll elő amit az elején írtam le.

15.5. Prolog családfa

Ágyazd be a Prolog családfa programot C++ vagy Java programba! Lásd [para_prog_guide.pdf](#)!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.6. GIMP Scheme hack

Ha az előző félévben nem dolgoztad fel a témát (például a mandalás vagy a króm szöveges dobozosat) akkor itt az alkalom!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16. fejezet

Helló, !

16.1. FUTURE tevékenység editor

Javítsunk valamit a ActivityEditor.java JavaFX programon! <https://github.com/nbatfai/future/tree/master/cs/F6>
Itt láthatjuk működésben az alapot: <https://www.twitch.tv/videos/222879467>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.2. OOCWC Boost ASIO hálózatkezelése

Mutassunk rá a scanf szerepére és használatára! <https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll>

Magát a scanf függvény adatok beolvasására használjuk. Ezeket az adatokat egy olyan bufferből olvassa be ami egy null végű string. Így néz ki maga a függvény:

```
int scanf(const char* buffer, const char* format, ...);
```

A függvényünk visszatérésének függvénye az, hogy sikerült-e annyi adatot beolvasnia amennyi a formatban meg van határozva. Tehát a buffer az azt mutatja majd meg nekünk, hogy honnan olvasunk, míg a format azt, hogy hogyan olvasunk. Mindkettő egy mutató tehát és egy null végű stringre mutatnak. Valamint lehetnek olyan esetek is, hogy hibát észlel futás közben ami annak köszönhető, hogy nem felel meg nekünk az adatbevitel.(Mármint a format előírásainak.) Ha minden rendben lefut és nem észlel hibát akkor azt adja vissza egy int formájában, hogy hány adatot olvasott be.

Ahogy ezt már korábban is tanultuk amikor a lexer-el foglalkoztunk, annak 3 része van: deklarációk, szabályok és a kiegészítő funkciók. Mivel ezt már tudjuk ezért meg tudjuk nézni hogyan használja a lexer a scanf-et.

```
{POS}{WS}{INT}{WS}{INT}{WS}{INT} {  
    std::sscanf(yytext, "<pos %d %u %u", &m_id, &from, &to);  
    m_cmd = 10001;  
}
```

```
POS "<pos"  
WS  [ \t]*
```

A lényeg az, hogy itt a POS és a WS vannak definiálva, mégpedig olyan módon, hogy az előbbi csak a pos-al kezdődő szavakat, míg a másik a [t]* karaktert jelöli.

```
{std::sscanf(yytext, "<pos %d %u %u", &m_id, &from, &to);  
    m_cmd = 10001;  
}
```

Az itt feltüntetett csipetben láthatjuk azt, hogy azokat a szövegrészeket amelyekre a lexer talált egyezést azt a yytext tartalmazza. Az értékeink a header fájlban definiált változók alapján kerülnek beolvasásra illetve beállításra. Ezek az m_id, from, to. Valamint az m_cmd amelynek értéke esetünkben 10001 lesz.

Valójában ezek után a többi rész is ezen logika alapján működik majd, csak más adatokkal illetve keresési kritériumokkal. Például találhatunk a vége felé egy bonyolultabb részt amely már azért szerintem sokkal szofisztikáltabb mint az előbb bemutatott kis kódcipet.

```
{ROUTE}{WS}{INT}{WS}{INT}({WS}{INT})* {  
    int size{0};  
    int ss{0};  
    int sn{0};  
  
    std::sscanf(yytext, "<route %d %d%n", &size, &m_id, &sn);  
    ss += sn;  
    for(int i{0}; i<size; ++i)  
    {  
        unsigned int u{0u};  
        std::sscanf(yytext+ss, "%u%n", &u, &sn);  
        route.push_back(u);  
        ss += sn;  
    }  
    m_cmd = 101;  
}
```

Talán kisebb részekre bontva könnyebben meg lehet érteni ennek a működését illetve, hogy végülis mit csinál.

```
{ROUTE}{WS}{INT}{WS}{INT}({WS}{INT})* {  
    int size{0};  
    int ss{0};  
    int sn{0};
```

Legelőször három változó deklarálása történik. Ezek lesznek a size, az ss és az sn. A size-ban fogjuk tárolni a teljes szöveg hosszát, az ss-ben pedig a már feldolgozott szövegrész mérete lesz eltárolva. Az sn pedig egy olyan tároló lesz nekünk ahol az éppen feldolgozott szöveg mérete lesz eltárolva viszont itt karakterekben mérve. Ezekre legfőképpen azért van szükségünk mert az egyik tagunk végén csillag állítjuk ami azt jelenti, hogy végtelen sok féle kombinációban állhatnak egymás után, vagy akár nincsenek is jelen.

```
std::sscanf(yytext, "<route %d %d%n", &size, &m_id, &sn);  
ss += sn;
```

Kettő db scanf függvényünk lesz egy for ciklussal egymásba ágyazva. Az elsőben a size-ba beolvassuk a szöveg méretét, majd az utána lévő int-et a m_id, sn-be.

```
for(int i{0}; i<size; ++i)
{
    unsigned int u{0u};
    std::sscanf(yytext+ss, "%u%n", &u, &sn);
    route.push_back(u);
    ss += sn;
}
```

Növeljük a feldolgozott karakterek számát az ss-ben miután megvolt az első scanf. Aztán jön az előbb említett for ciklus. Ez az egész szöveg végéig fut. Elérkeztünk a második scanf-hez, yytexthez hozzá kell, hogy adjuk az ss értéket, ezzel azt érjük el, hogy onnan tudjuk folytatni a beolvasást ahol befejeztük. Illetve lesz itt még nekünk egy unsigned int-ünk is, ezt u-ba szépen beolvassuk, sn-ben tároljuk amit kell benne, majd pedig beletesszük a route vektorba az u-t. Továbbra is el kell érniünk azt, hogy a következő beolvasásnál is onnan kezdjük el ahol itt befejeztük, ennek érdekében az ss-t sn értékével növeljük.

```
m_cmd = 101;
```

az m_cmd-t most 101-re állítjuk.

16.3. SamuCam

Mutassunk rá a webcam (pl. Androidos mobilod) kezelésére ebben a projektben: <https://github.com/nbatfai/SamuCam>

A feladatban használt kód a webkameránkat fogja használni és annak segítségével arcokat olvas majd be.

A projektet én Bátfai Tanár Úr githubjáról szedtem le gitclone segítségével. Aztán belépünk a mappába ahova letöltöttük és ott egy parancs segítségével egy xml-t kapunk, majd jöhet a qmake. Ezek után már csak egy make majd a ./fájlnév paranccsal futtatjuk is Ezek rendre így néznek ki:

```
git clone https://github.com/nbatfai/SamuCam.git
```

```
wget https://github.com/Itseez/opencv/raw/master/data/lbpcascades/ ↵
lbpcascade_frontalface.xml
```

```
~/Qt/5.12.2/gcc_64/bin/qmake SamuLife.pro
```

Ha rendelkezünk webkamerával elvileg gond nélkül érzékelnie kell azt. Sajnálatos módon én nem rendelkezem ezzel, illetve a laptopomba sincs beépített webkamera ezért a helyes futást nem tudtam ellenőrizni. Annyi információm van még a kódról, hogy ha alapértelmezetté akarjuk tenni a kameránkat akkor azt a videostream-ben tudjuk elérni a következő változtatások segítségével:

```
void SamuCam::openVideoStream()
{
    videoCapture.open(0); //itt módosítottam
    videoCapture.set(CV_CAP_PROP_FRAME_WIDTH, width);
    videoCapture.set(CV_CAP_PROP_FRAME_HEIGHT, height);
}
```



```
videoCapture.set ( CV_CAP_PROP_FPS, 10 );  
}
```

A SamuCam.h fájlból tudjuk megállapítani, hogy a Qthread osztályból származik.

```
class SamuCam : public QThread  
{  
    Q_OBJECT  
public:  
    SamuCam ( std::string videoStream, int width, int height );  
    ~SamuCam();  
    void openVideoStream();  
    void run();  
private:  
    std::string videoStream;  
    cv::VideoCapture videoCapture;  
    int width;  
    int height;  
    int fps;  
signals:  
    void faceChanged ( QImage * );  
    void webcamChanged ( QImage * );  
};
```

16.4. BrainB

Mutassuk be a Qt slot-signal mechanizmust ebben a projektben: <https://github.com/nbatfai/esport-talent-search>

A BrainB-vel foglalkoztunk korábban is a tanulmányaink során, ebben a félévben például esport kurzuson is használtuk. Maga a program azt csinálja, hogy felméri az ún. karakter elvesztést, ez legfőképpen MOBA-kban fordul elő ahol egyszerre nagyon sok minden történik és hajlamosak a játékosok nem észrevenni merre van a saját karakterük. Ez lenne végülis magának a karakter elvesztésnek a definíciója. Az általunk használt BrainB programban a 10 perces futás alatt minél több ideig és minél folyamatosabban a Samu Entropy nevű kis négyzetet kell tartanunk az egeret.

A megoldáshoz illetve a program megfelelő futásához több könyvtárra is szükségünk van, ezek a libqt4-dev, az opencv-data a libopencv-dev, valamint ugye maga a Qt is szükséges. Ebben a feladatban specifikusan a QT slot-signal mechanizmusát kell bemutatnunk majd.

A slotok nagyon sokban hasonlítanak a függvényekre. Vannak definícióik, paramétereik és meg is lehet őket hívni mint egy rendes függvényt, viszont nincs visszatérési értékük. A signaloknak pedig még ennél is egyszerűbb a felépítésük. Meghíni ugyan lehet őket, viszont csak az emit segítségével. Csak és kizárólag paraméterekkel rendelkeznek, se definíció, se visszatérési érték nincs.

A slotokat és a signalokat lehet párba is rendezni, viszont ehhez teljesülnie kell annak, hogy a paraméterek páronként ugyanolyanok, vagy csak a signalnak legyenek paraméterei és azokat a slot át tudja majd venni. Ez így néz ki a kódban:

```
BrainBWin::BrainBWin ( int w, int h, QWidget *parent ) : QMainWindow ( ←
    parent )
{
//    setWindowTitle(appName + " " + appVersion);
//    setFixedSize(QSize(w, h));
    statDir = appName + " " + appVersion + " - " + QDateTime::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 ) ) );
}
```

A slot-signal párost itt a connect segítségével hozzuk létre. Négy paraméterünk lesz majd. Az első az az objektum ami a magát a signalt küldi, aztán a signal a harmadik a signal kezelő objektumra mutató mutató és végül de nem utolsó sorban a negyedik a slot.

```
connect ( brainBThread, SIGNAL ( heroesChanged ( QImage, int, int ) ←
    ),
    this, SLOT ( updateHeroes ( QImage, int, int ) ) );
```

Vagyis ha a heroesChanged signal (ami a brainBThread objektum része) aktiválódik akkor ezt az updateHeroes slotal fogja kezelni a BrainBWin. Tehát a megjelenést és az ablakok frissítését az updateHeroes segítségével végzi majd el amit aminek értékeit a signal viszi is adja át.

```
connect ( brainBThread, SIGNAL ( endAndStats ( int ) ),
    this, SLOT ( endAndStats ( int ) ) );
```

A következő connectben viszont már az endAndStats signalt fogjuk kezelni. Ez akkor lesz aktív ha a futási idő lejár. Ekkor a BrainBWin ezt az endAndStats slotjával fogja kezelni. Itt történik majd meg végül a futási ablak bezárása és a program leállítása.

```
void BrainBThread::run()
{
    while ( time < endTime ) {
        QThread::msleep ( delay );
        if ( !paused ) {
            ++time;
            devel();
        }
        draw();
    }
    emit endAndStats ( endTime );
}
```

Ebben a kódcipetben pedig már azt láthatjuk, hogy az a második connect slot-signal-ját használja fel, és ezt az elején említett emit-tel hívja meg.

16.5. OSM térképre rajzolása

Debrecen térképre dobjunk rá cuccokat, ennek mintájára, ahol én az országba helyeztem el a DEAC hekkereket: <https://www.twitch.tv/videos/182262537> (de az OOCWC Java Swinges megjelenítőjéből: <https://github.com/emulator/tree/master/justine/rcwin> is kiindulhatsz, mondjuk az komplexebb, mert ott időfejlődés is van...)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

17. fejezet

Helló, Lauda!

17.1. Port scan

Mutassunk rá ebben a port szkennelő forrásban a kivételkezelés szerepére! <https://www.tankonyvtar.hu/-hu/tartalom/tkt/javat-tanitok-javat/ch01.html#id527287>

Ebben a feladatban a fentebb megadott forrást használtam fel.

```
public class KapuSzkennner {  
  
    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 + " figyel!");  
  
                socket.close();  
  
            } catch (Exception e) {  
  
                System.out.println(i + " nem figyel!");  
  
            }  
  
        }  
  
    }  
}
```

Ez a kis promgram egy for ciklust fog alkalmazni, amelynek segítségével 0-tól 1024-ig lévő portokon látó TCP kapukat nézi meg. Ezen belül is azt deríti ki, hogy lehet-e ezeken keresztül kapcsolatot kiépíteni. Az igazság az, hogy ezt futtatni nem érdemes/szabad csak saját gépen, vagy ismert gépen mert akár fenyegetésnek is vehetik. Ha ez a kapcsolatépítés lehetséges, vagyis a porton keresztül el tudjuk érni a szerveret akkor a figyel! üzenetet kapjuk vissza ahhoz a porthoz. Ha ez a kapcsolat viszont nem megvalósítható, akkor a try-catch segítségével visszkapunk egy kivételt a nem figyel! üzenet formájában.

17.2. AOP

Szőj bele egy átszövő vonatkozást az első védési programod Java átíratába! (Sztenderd védési feladat volt korábban.)

Az AOP egy programozási mód amely az aspektus orientált programozás rövidítése (Aspect oriented programming). OO-tól abban különbözik legfőképpen, hogy magasabb szintű az absztrakció benne. Segítségével különböző aspektusokból tudjuk megfigyelni a kódunkat, annak működését, viselkedését és ehhez még bele sem kell, hogy szerkesszünk vagy nyulkáljunk magába a kódba. Ehelyett inkább egy külön fájlba írjuk meg amit szeretnénk és ezt egy fordítóval hozzáfűzzük a fő forráshoz. Ezeknek külön logja lesz majd. A már korábban használt java-s binfa forrásunkba kell egy AspectJ szkriptet tenni. Ez azt fogja csinálni, hogy az eredeti inorder kiírás mellett preorder-ben is ki lesz íratva a kimenet, ehhez a kiír függvényen kell változtatni. A kiír függvényünket egy pointcutba helyezzük bele, ennek segítségével tudjuk majd a feladatot megcsinálni. (valamit itt lesznek majd a jointpointok is) A call függvény segítségével meghívjuk majd a kiír-t és azzal együtt az AspectJ-t is (tehát a kiír már az alapján fut le) viszont maga a kiír függvényig minden változatlanul történik majd. Ezután az after függvény segítségével megjelenítünk mindent a kiíron belül, vagyis az inorder és az AspectJ-s preordert is. És ez tökéletesen demonstrálja az AOP működését és hasznát, mivel nem nyúltunk hozzá az eredeti forráshoz, mégis más a végeredmény amit kapunk. Itt láthatjuk végül a .aj fájlunkat, ami az előbbieken elmodnott dolgokat csinálja meg, tehát a call-al meghívja a függvényeket amire szükség van, majd a kimenetet az afterben határozza meg. Aztán bele kell még írni magát a preorder kiírás függvényét is. Egy txt fogja majd tartalmazni az ehhez tartozó kimenetet.

```
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.egyGyermek(), p);
preorder(elem.nullasGyermek(), p);
--melyseg;
}
}
}
```

17.3. Android Játék

Írjunk egy egyszerű Androidos „játékot”! Építkezzünk például a 2. hét „Helló, Android!” feladatára!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.4. Junit teszt

A https://progater.blog.hu/2011/03/05/labormeres_otthon_avagy_hogyan_dolgozok_fel_egy_pedat poszt kézzel számított mélységét és szórását dolgozd be egy Junit tesztbe (sztenderd védési feladat volt korábban).

Mi is ez a Junit teszt? Vagyis pontosabban mi a Junit? Nem más mint egy keretrendszer, amit egységek tesztelésére használnak a javaban.

```
@org.junit.Test
```

A Junit tesztfutatójának úgy tudjuk megadni, hogy mely metódusokat vizsgálja meg, hogy a metódus elé egy @ jelet teszünk (ez egy annotáció, többféle is van már belőle de nekünk ez most tökéletesen megteszi) Maga a teszt úgy épül fel, hogy @-al kezdjük, aztán megadjuk a tesztelendő metódust amit meg fog majd hívni. Az eredményt pedig össze tudjuk majd hasonlítani azzal amit mi várunk el ezektől a bizonyos metódusoktól. Ezzel végülis azt tudjuk meg, hogy úgy működnek-e ahogy azt mi szeretnénk. Itt látható a tesztelési szegmens:

```
public void tesBitFeldolg() {
    for (char c : "01111001001001000111".toCharArray())
    {
        binfa.egyBitFeldolg(c);
    }
    org.junit.Assert.assertEquals(4, binfa.getMelyseg(), 0.0);
    org.junit.Assert.assertEquals(2.75, binfa.getAtlag(), 0.001);
    org.junit.Assert.assertEquals(0.957427, binfa.getSzoras(), 0.0001);
}
```

A tesztet elvégző programunk neve a testBitFeldolg lesz, ebben a megadott tömböt az egyBitFeldolg segítségével bitenként dolgozzuk fel. Ezután történik az összehasonlítás melyet a következő 3 sorban végzünk el. Magát az összehasonlítást az assertEquals függvény végzi. Ez három paraméterrel rendelkezik, ezek rendre az elvárt eredmény, a kapott érték valamint a max eltérés mértéke lesznek.

17.5. OSCI

Készíts egyszerű C++/OpenGL-es megjelenítőt, amiben egy kocsit irányítasz az úton. A kocsí állapotát minden pillanatban mentsd le. Ezeket add át egy Prolog programnak, ami egyszerű reflex ágensként adjon vezérlést a kocsinak, hasonlítsd össze a kézi és a Prolog-os vezérlést. Módosítsd úgy a programodat, hogy ne csak kézzel lehessen vezérelni a kocsit, hanem a Prolog reflex ágens vezérelje!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

18. fejezet

Helló, Calvin!

18.1. MNIST

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel, https://progpater.blog.hu/2016/11/13/-hello_samu_a_tensorflow-bol Hátterként ezt vetítsük le: <https://prezi.com/0u8ncvvoabcr/no-programming-programming/>

Az MNIST egy adatbázis, ami kézzel írott arab számjegyek ezreit tartalmazza. Kétféle állomány található benne, a tanulási és a tesztállomány. Előbbi képekből tanulja be a gépünk a számjegyeket majd utóbbival tudjuk ellenőrizni, hogy ez milyen hatékonysággal történt meg. Itt látható maga a tanulási folyamat a kódban:

```
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
```

Nem lehet csak úgy nekiugrani a feladatnak, mivel több könyvtár is szükséges ahhoz, hogy működjön. Szükségünk lesz például a python3-pip-re és a python3-dev-re, ha ezekkel nem rendelkezünk még akkor be kell szerezni őket. Ezek után pedig telepítettem a Kerast:

```
sudo pip3 install keras
```

Utána be kell, hogy töltsük az adatbázist, ez itt látható:

```
(train_X, train_Y), (test_X, test_Y) = tf.keras.datasets.mnist.load_data()

train_X = train_X.reshape(-1, 28, 28, 1)
test_X = test_X.reshape(-1, 28, 28, 1)
```


Ebben a csipetben láthatjuk a vektorok létrehozását a reshape segítségével. Az első paramétere -1 ami azt jelenti, hogy minden tagra értelmezzük, a második és harmadik paraméter pedig 28, ez azt eredményezi, hogy 28, 28 elemmel rendelkező vektort hozunk létre. A negyedik paraméter pedig az 1, ez a paraméter a színcsatornát adja meg, az 1-el a grayscale képeket állíthatjuk be. (pl 3-assal az RGB-t tudnánk beállítani)

```
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)
```

Itt a one_hot kódolás megjelenése látható a forrásban. Ennek segítségével a számokat (ugye 0-tól 9-ig) 9db 0-val és 1db 1-essel le is tudjuk írni. Ez úgy működik, hogy az egyes minden egyes számnál más helyen fog állni, így tudjuk melyik számról van szó. Talán a legnagyobb ok amiért ezt használunk kell az az, hogy a kódunkat nem tudjuk működésre bírni csak nem kategórikus adatokkal.

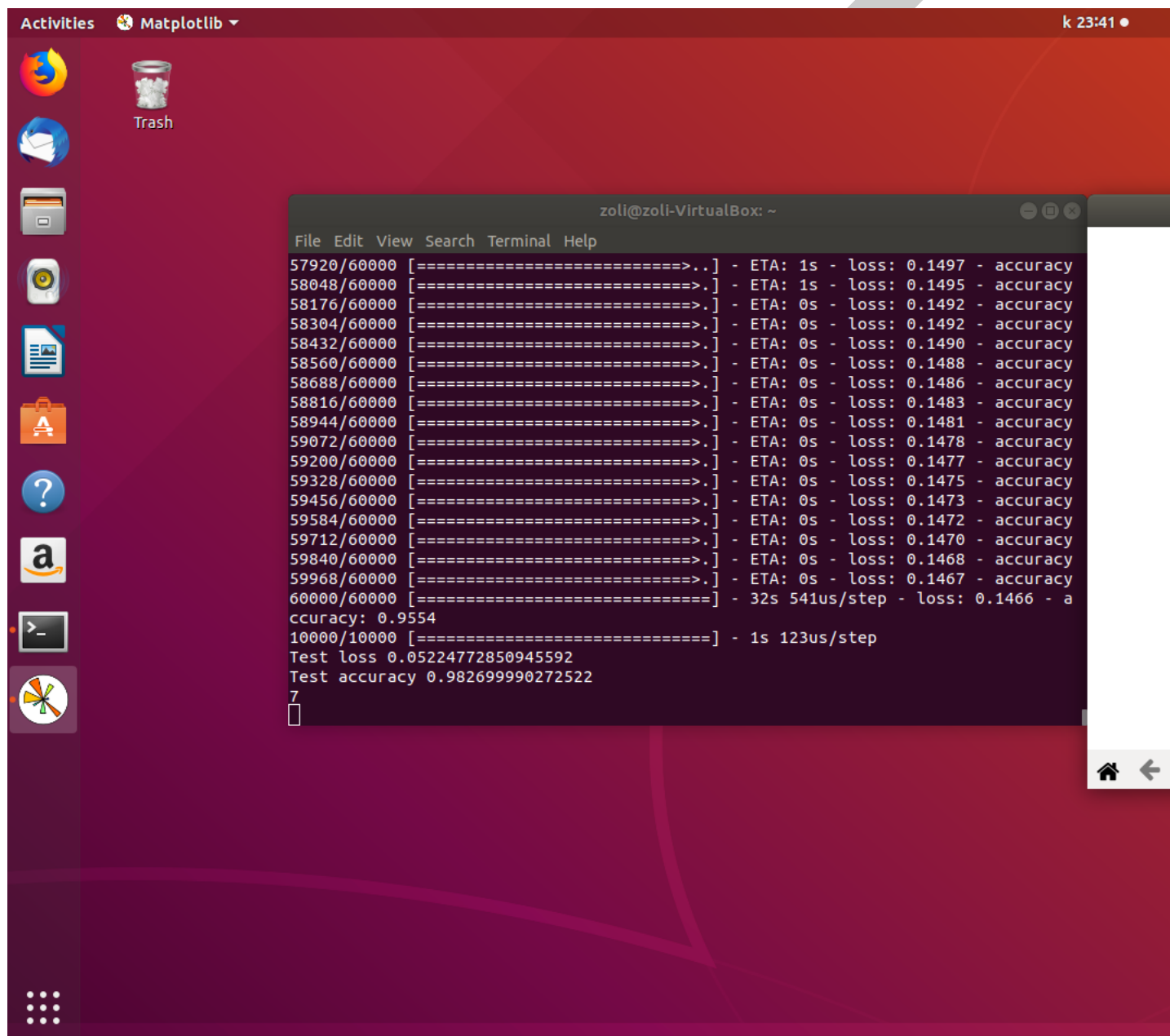
```
model = Sequential()
model.add(Conv2D(64, (3,3), input_shape=(28, 28, 1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64, (3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(64))
model.add(Dense(10))
model.add(Activation('softmax'))
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras. ←
               optimizers.Adam(), metrics=['accuracy'])
```

Az add() segítségével egymásra pakoljuk a rétegeket, ezzel létrehozva a modellünket. Három paramétere lesz. Az első a neuronok száma lesz ami a mi esetünkben most 64, a második a detectorunk lesz, ide 3,3 értéket írunk be. Végül a harmadik egy input_shape lesz, ebben a már előbb említett 28x28 grayscale-es képeink lesznek benne. Utána a Rectified Linear Unit-ot kell aktivizálnunk, ezt röviden relu-ként tudjuk megadni. Aztán a pool_size segítségével megadjuk egyszerre mennyi adatot dolgozzon fel a program. Végül elindítjuk magát a tanulási folyamatot.

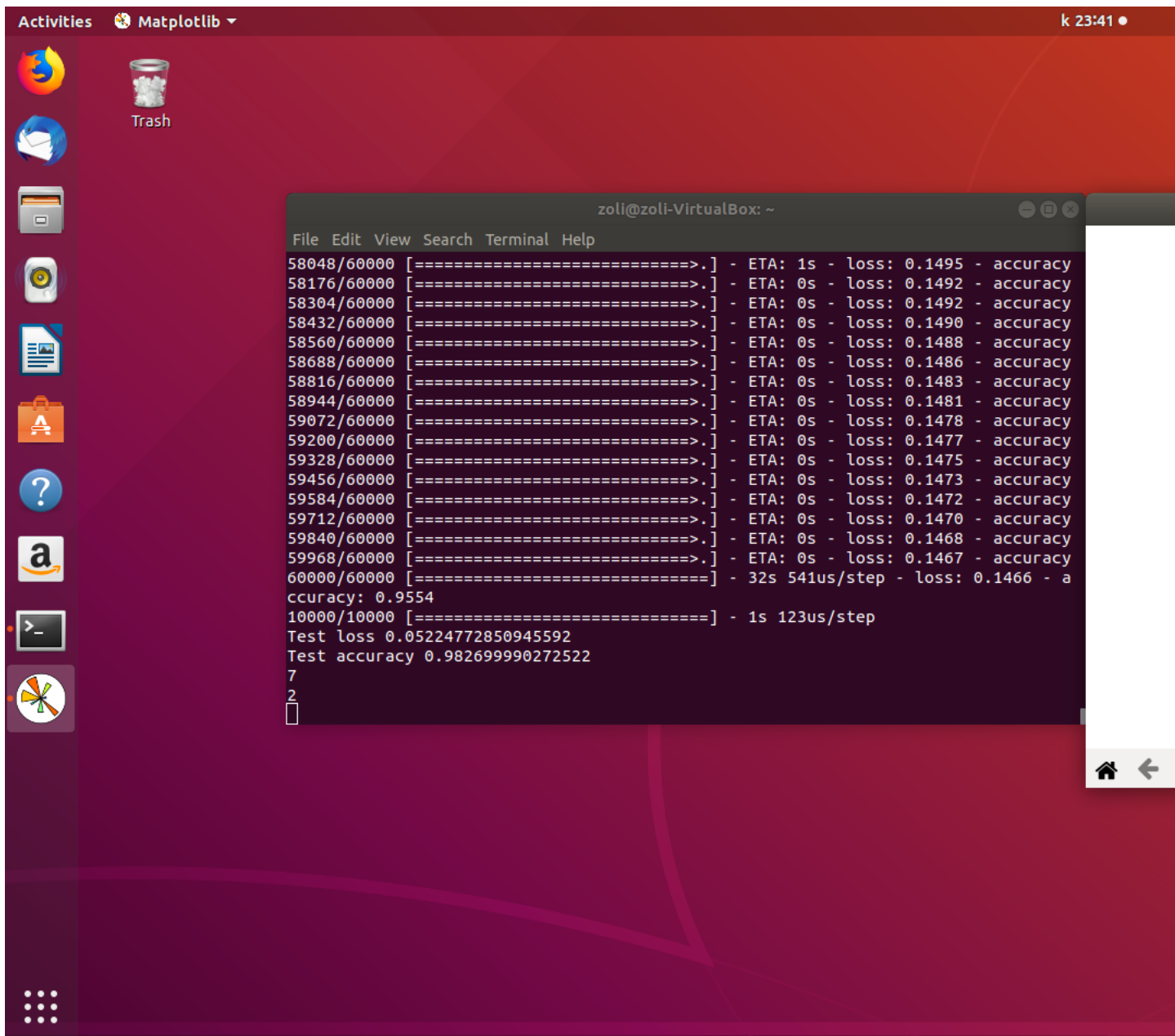
```
model.fit(train_X, train_Y_one_hot, batch_size=64, epochs=1)
test_loss, test_acc = model.evaluate(test_X, test_Y_one_hot)
print('Test loss', test_loss)
print('Test accuracy', test_acc)
predictions = model.predict(test_X)
print(np.argmax(np.round(predictions[0])))
plt.imshow(test_X[0].reshape(28, 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('szam.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()
```

A fentebb látható kódcsipetben többek között azt állítjuk be, hogy a tanulási folyamat hányszor történjen meg, egyértelműen ez minél nagyobb annál pontosabb eredményeket kapunk majd. Ezt az epochs után álló számmal állíthatjuk be. De tökéletesen megfelel az 1 is mivel így is viszonylag pontos eredményeket fogunk kapni. Ezek mellett a futtatási kiírásokat is itt adjuk meg, és ahogy azt a feladat kéri egy "kézzel" írt számot is beadunk.



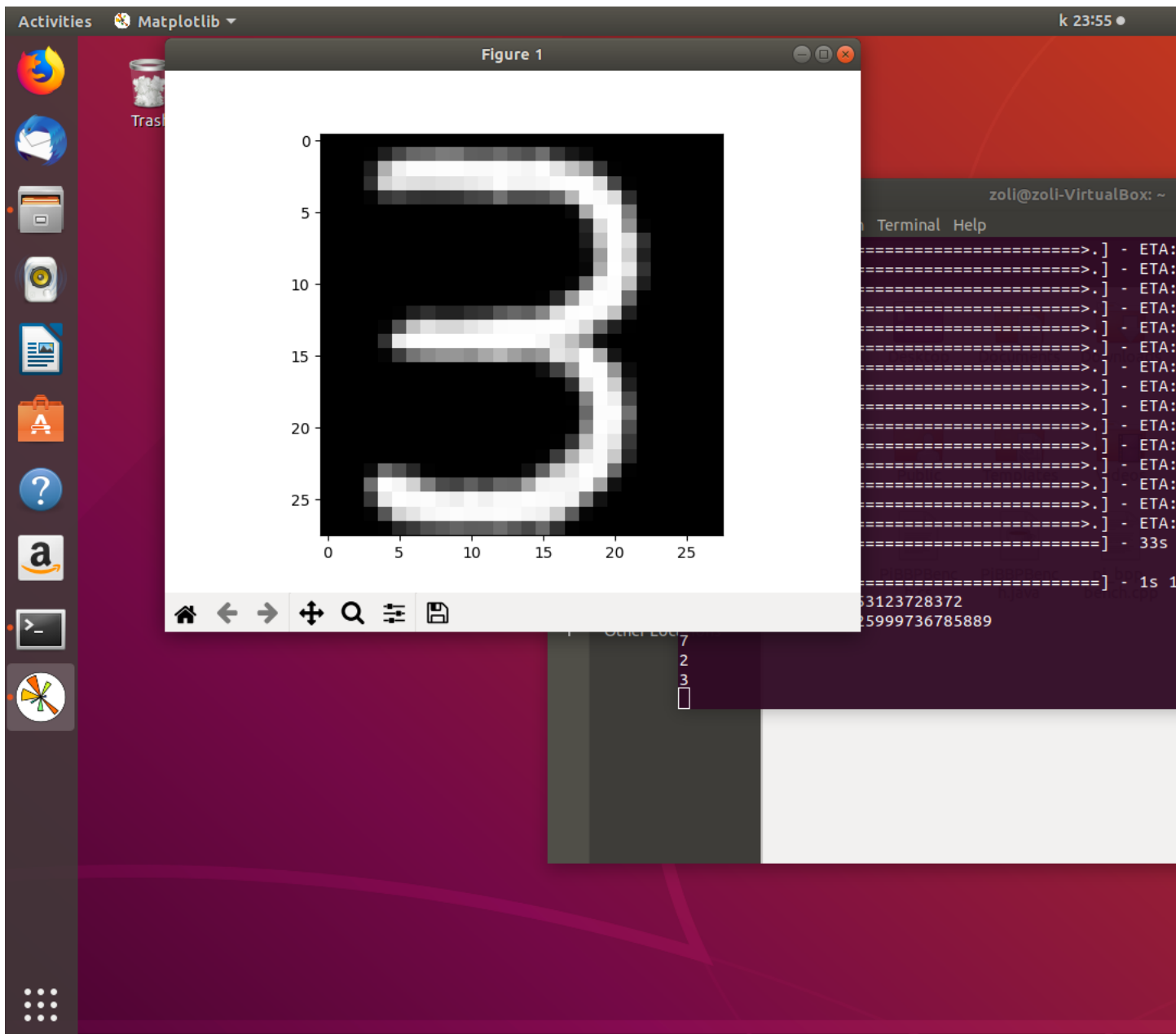
18.1. ábra. 1.



The screenshot shows a Linux desktop with a red background. On the left is a vertical dock with icons for Firefox, a mail client, a file manager, a music player, a document viewer, a shopping bag, a question mark, an Amazon logo, a terminal, and a game controller. The top bar shows 'Activities', 'Matplotlib', and the time 'k 23:41'. A terminal window titled 'zoli@zoli-VirtualBox: ~' is open, displaying the following output:

```
File Edit View Search Terminal Help
58048/60000 [=====>.] - ETA: 1s - loss: 0.1495 - accuracy
58176/60000 [=====>.] - ETA: 0s - loss: 0.1492 - accuracy
58304/60000 [=====>.] - ETA: 0s - loss: 0.1492 - accuracy
58432/60000 [=====>.] - ETA: 0s - loss: 0.1490 - accuracy
58560/60000 [=====>.] - ETA: 0s - loss: 0.1488 - accuracy
58688/60000 [=====>.] - ETA: 0s - loss: 0.1486 - accuracy
58816/60000 [=====>.] - ETA: 0s - loss: 0.1483 - accuracy
58944/60000 [=====>.] - ETA: 0s - loss: 0.1481 - accuracy
59072/60000 [=====>.] - ETA: 0s - loss: 0.1478 - accuracy
59200/60000 [=====>.] - ETA: 0s - loss: 0.1477 - accuracy
59328/60000 [=====>.] - ETA: 0s - loss: 0.1475 - accuracy
59456/60000 [=====>.] - ETA: 0s - loss: 0.1473 - accuracy
59584/60000 [=====>.] - ETA: 0s - loss: 0.1472 - accuracy
59712/60000 [=====>.] - ETA: 0s - loss: 0.1470 - accuracy
59840/60000 [=====>.] - ETA: 0s - loss: 0.1468 - accuracy
59968/60000 [=====>.] - ETA: 0s - loss: 0.1467 - accuracy
60000/60000 [=====] - 32s 541us/step - loss: 0.1466 - a
ccuracy: 0.9554
10000/10000 [=====] - 1s 123us/step
Test loss 0.05224772850945592
Test accuracy 0.982699990272522
7
2
█
```

18.2. ábra. 2.



18.3. ábra. 3.

18.2. Deep MNIST

Mint az előző, de a mély változattal. Segítő ábra, vedd össze a forráskóddal a <https://arato.inf.unideb.hu/batfai.norbert> 8. fóliáját!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.3. CIFAR-10

Az alap feladat megoldása, +saját fotót is ismerjen fel, https://progpater.blog.hu/2016/12/10/hello_samu_a_cifar-10_tf_tutorial_peldabol

Ez a feladat az előző egy átalakított változata, mivel abban számokat kellett, hogy felismerjen a programunk, itt ugyan ezt kell tennie képekkel amelyek például tárgyakat, élőlényeket ábrázolnak. Éppen ennek köszönhetően most elhagyjuk a grayscale képeket és áttérünk színesekre. A feladatok hasonlóságából eredően nem sok különbség van a források között, de természetesen vannak fontos különbségek.

```
(train_X, train_Y), (test_X, test_Y) = cifar10.load_data()
```

Egyértelmű különbség például az, hogy más adatbázist kell alkalmaznunk mivel a képek eltérőek lesznek az előzőhöz képest.

```
train_X = train_X.reshape(-1, 32, 32, 3)
test_X = test_X.reshape(-1, 32, 32, 3)
```

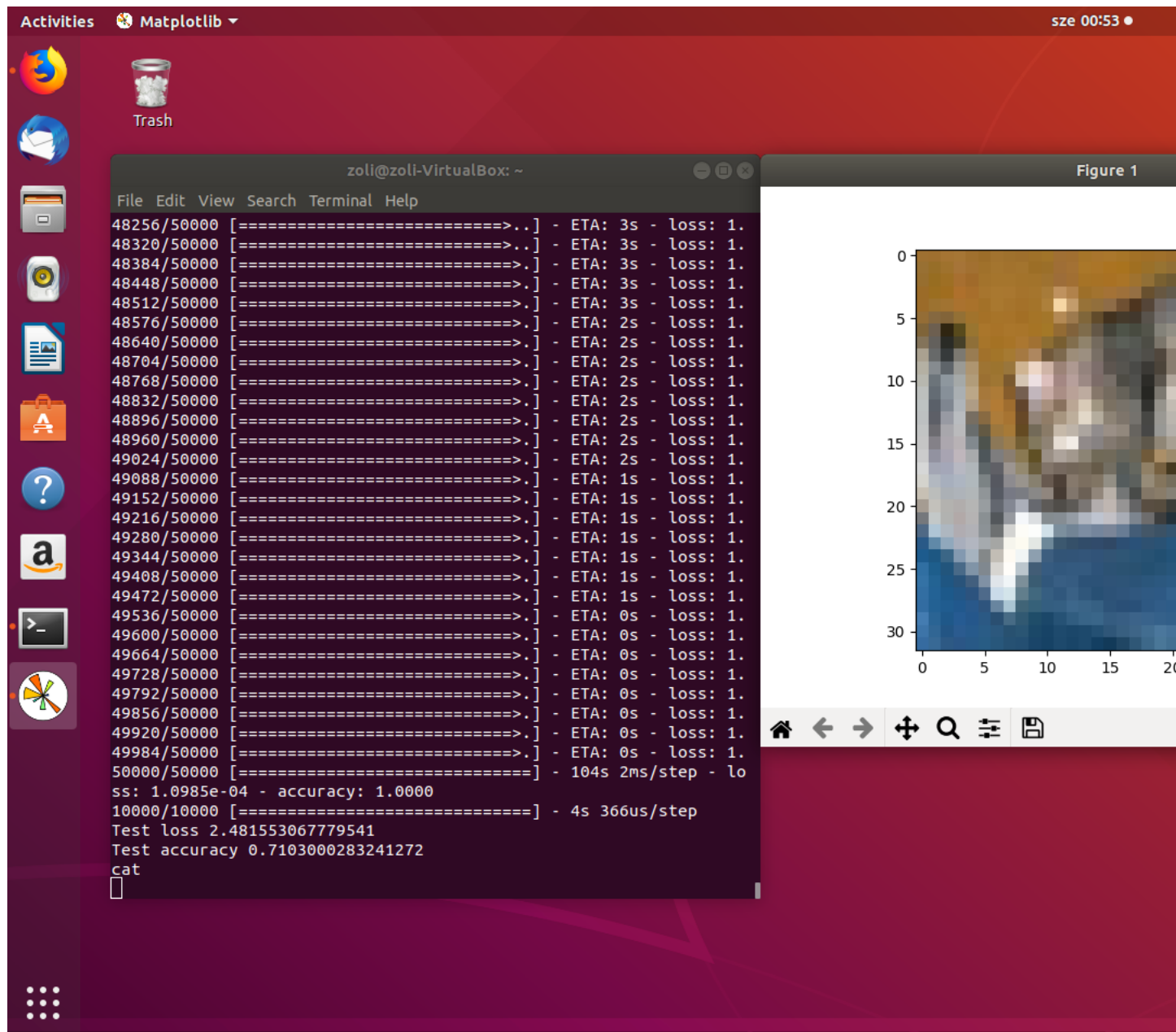
A reshape függvényünk is meg lesz variálva egy kicsit, a második, harmadik és negyedik paraméter rendre 32-re, 32-re és 3-ra módosul. Ennek oka a képek mérete, vagyis mostmar 32x32 színes képünk van.

```
model.add(Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)))
```

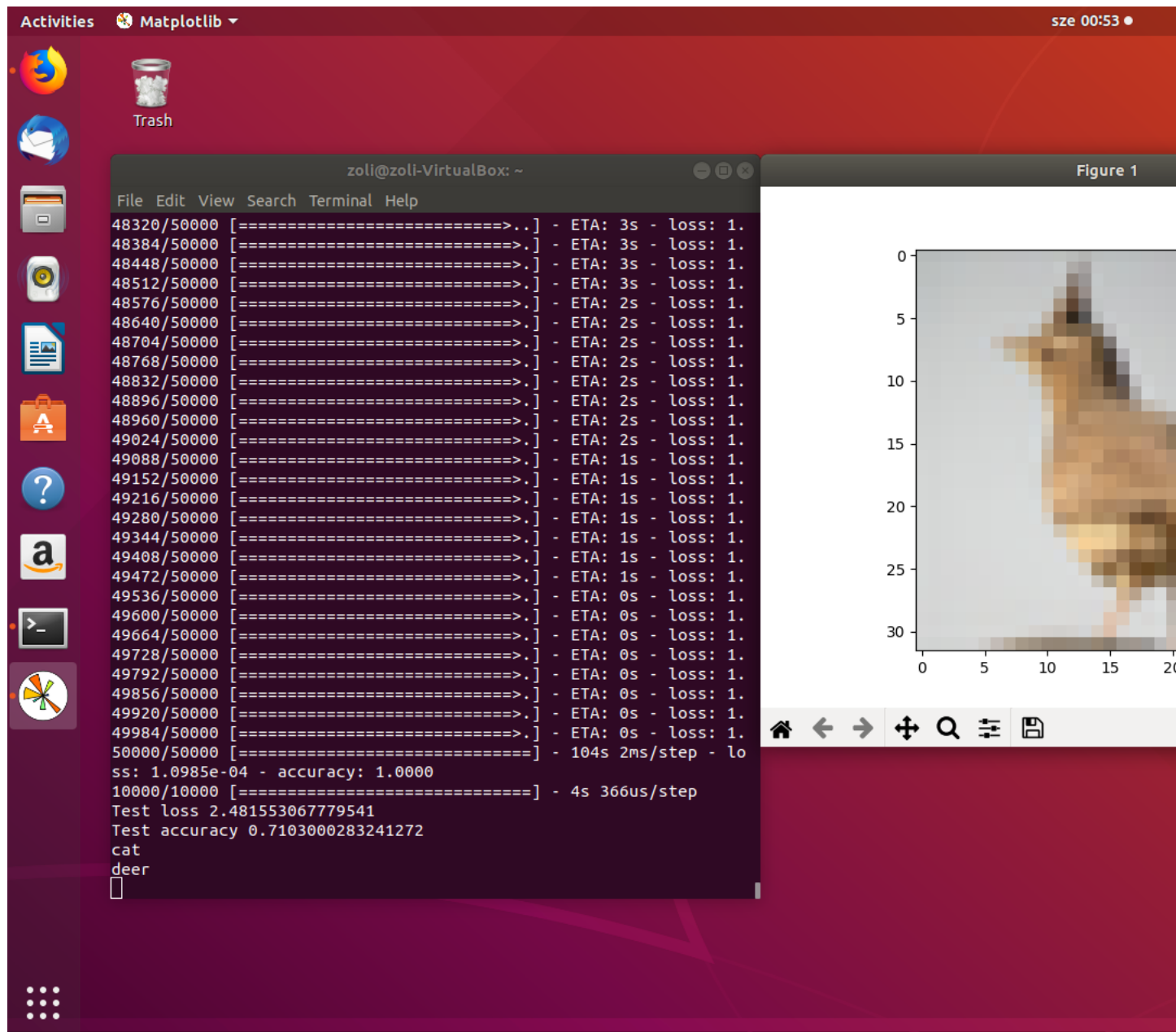
Ebből eredően változik majd az input_shape is.

```
cifar_classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', ' ←
                 frog', 'horse', 'ship', 'truck']
print(cifar_classes[np.argmax(np.round(predictions[0]))])
```

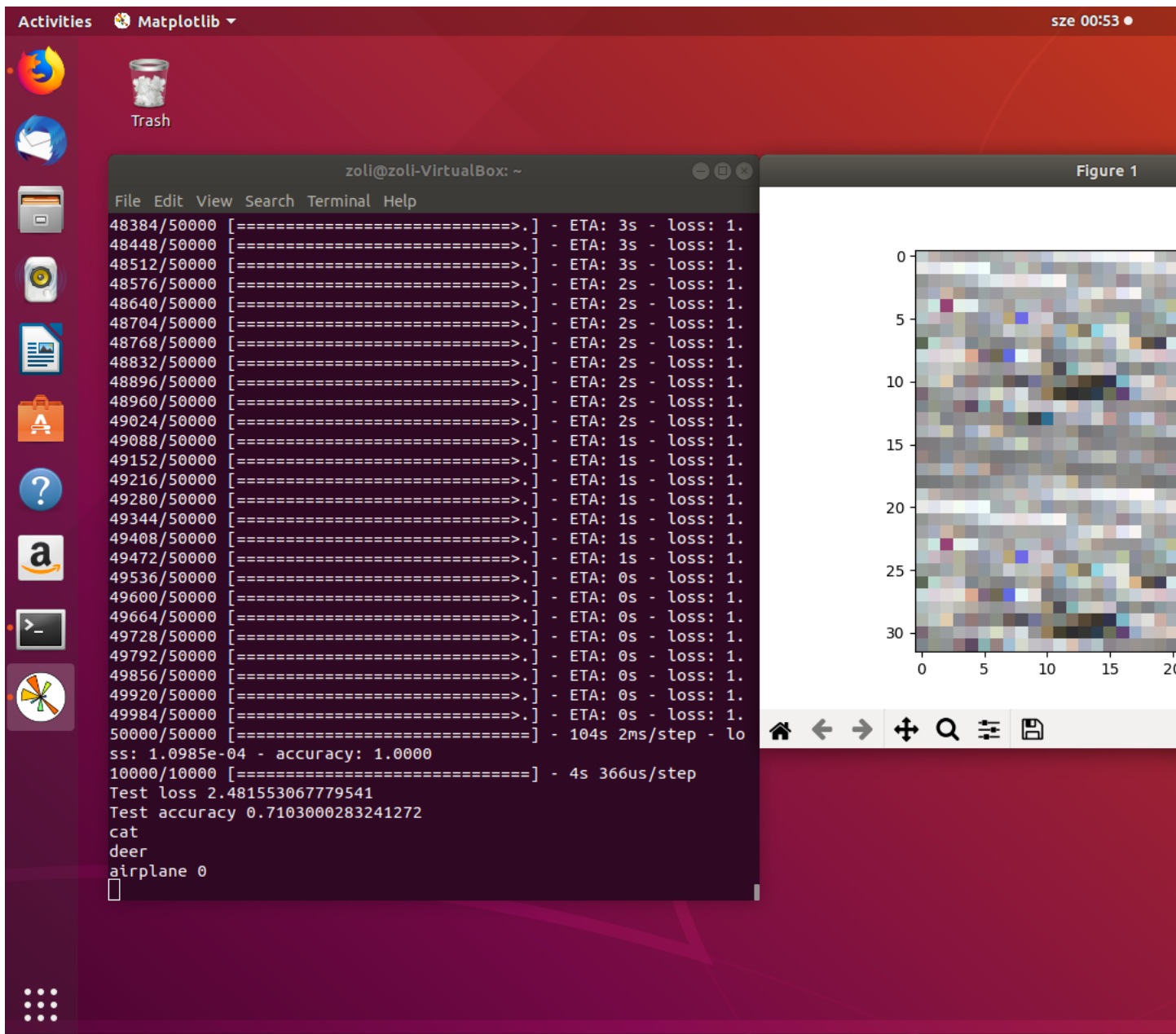
A class tömbbeli változtatások eredményezik talán a legnagyobb különbséget, mivel itt nekünk kell megadni, hogy miről található kép az adatbázisban. A feladat által kért saját képet is ezekből a kategóriákból kell majd kiválasztanunk.



18.4. ábra. 4.



18.5. ábra. 5.



18.6. ábra. 6.



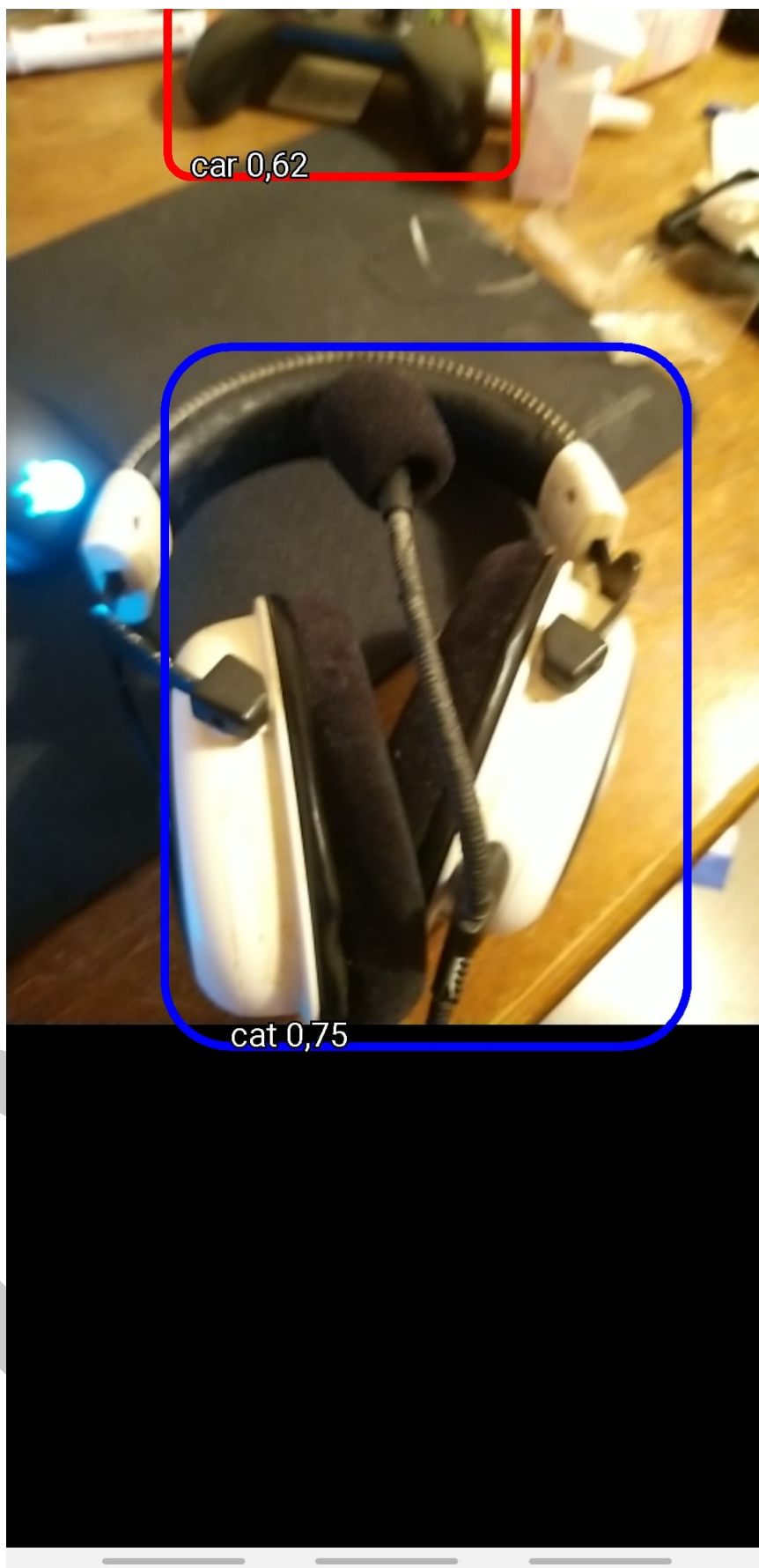
18.7. ábra. 7.

18.4. Android telefonra a TF objektum detektálója

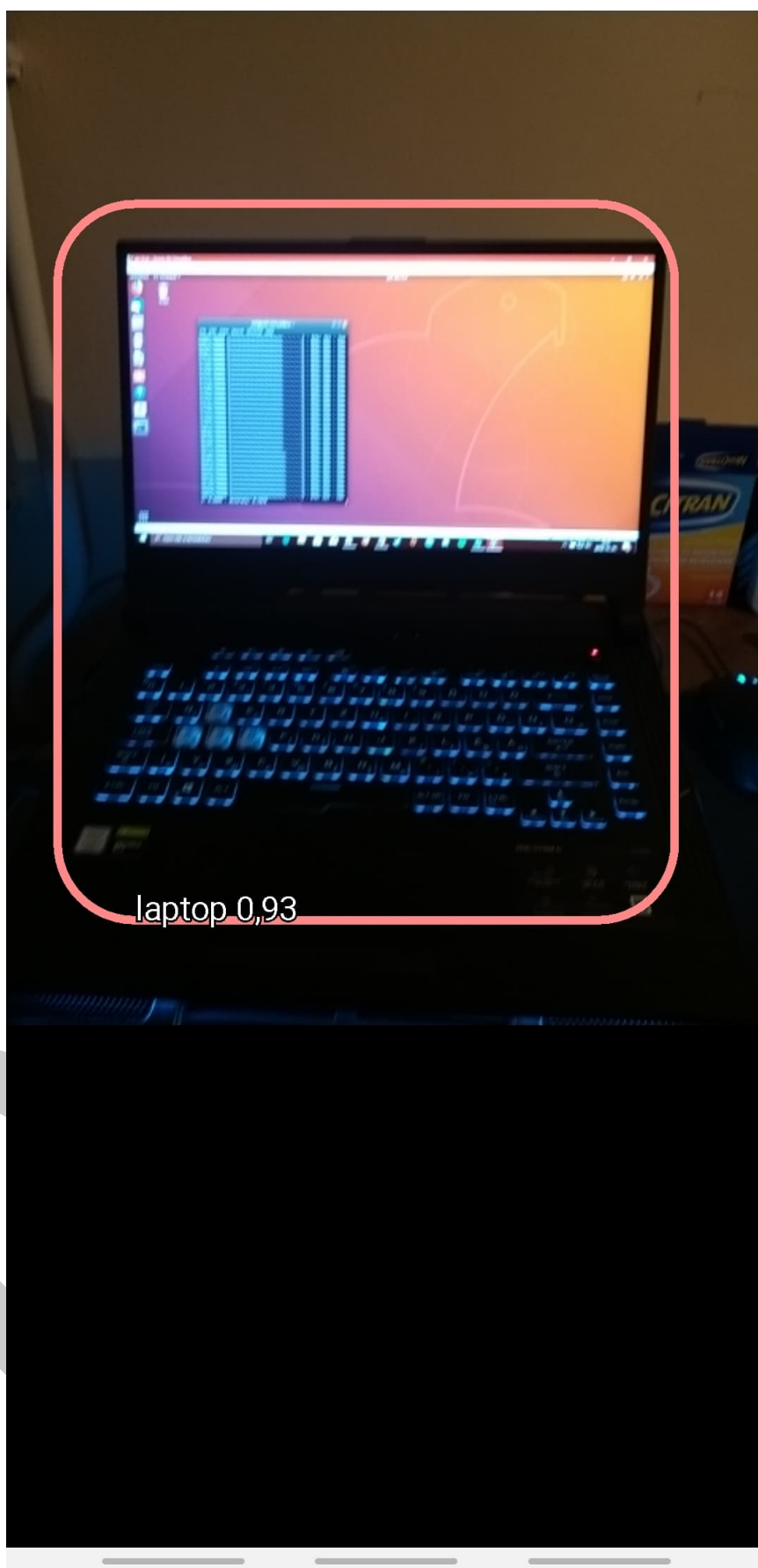
Telepítsük fel, próbáljuk ki!

Ebben a feladatban csupán annyit kellett csinálnunk, hogy a Tensorflow tárgyfelismerő appját kipróbáljuk. Erre szeretnék egy pár példát mutatni:

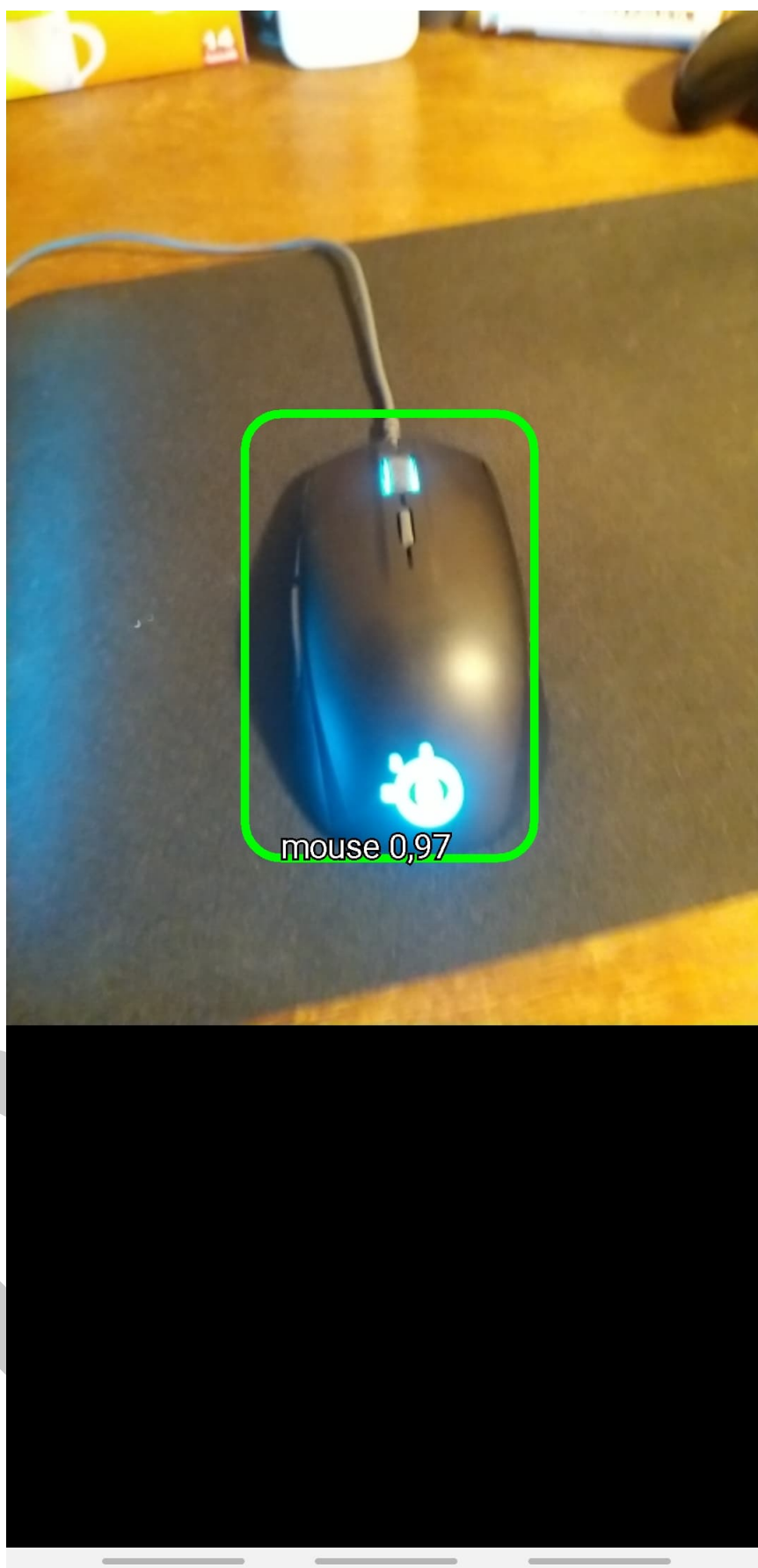
DRAFT



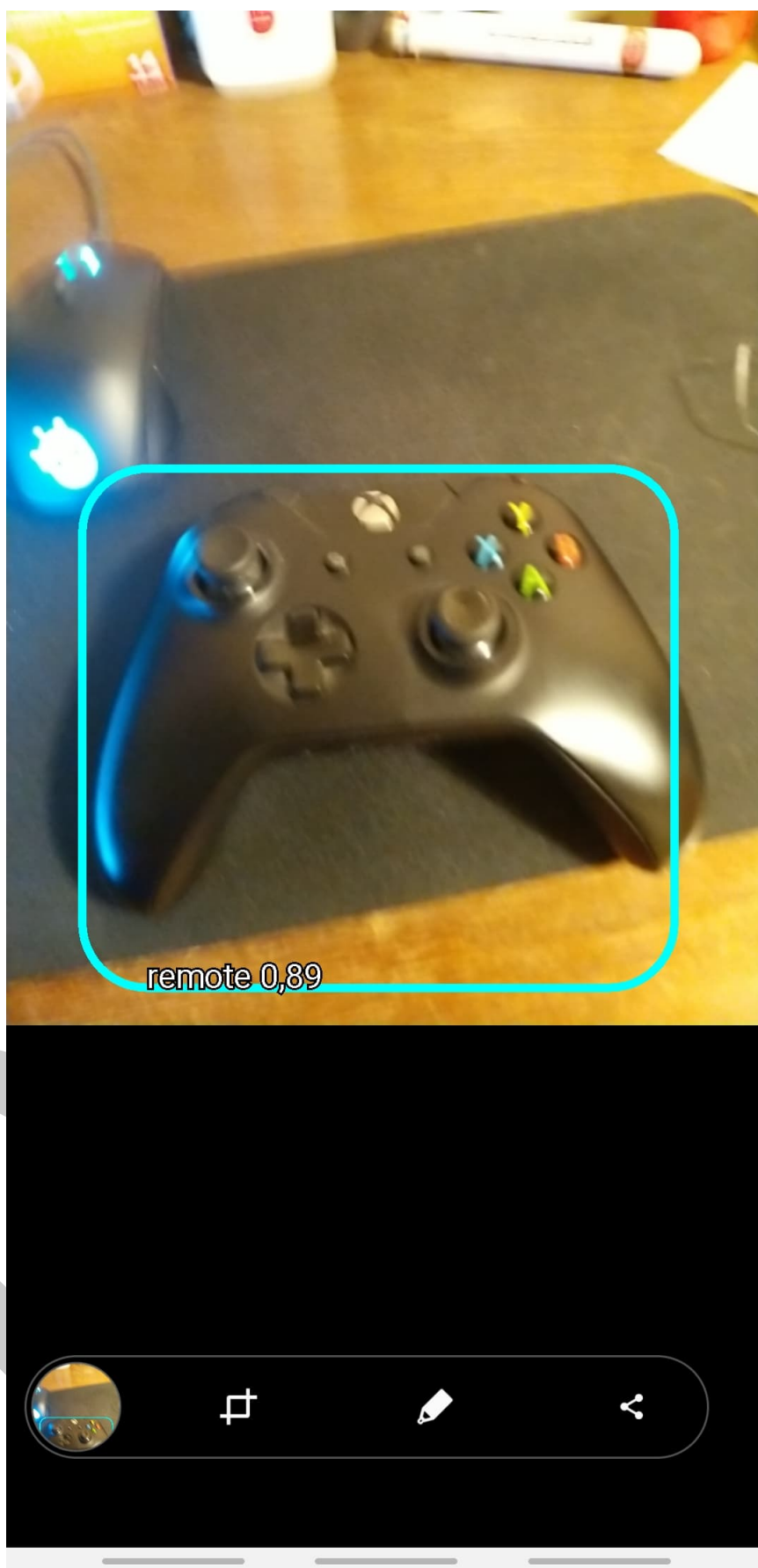
18.8. ábra. 8.



18.9. ábra. 9.



18.10. ábra. 11.



18.11. ábra. 12.

18.5. SMNIST for Machines

Készíts saját modellt, vagy használj meglévőt, lásd: <https://arxiv.org/abs/1906.12213>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.6. Minecraft MALMO-s példa

A <https://github.com/Microsoft/malmo> felhasználásával egy ágens példa, lásd pl.: <https://youtu.be/bAPSu3Rndi8>, https://bhaxor.blog.hu/2018/11/29/eddig_csaltunk_de_innentol_mi, <https://bhaxor.blog.hu/2018/10/28/minecraft>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19. fejezet

Helló, Berners Lee!

19.1. C++ és Java

Ebben a fejezetben a Java és C++ nyelv összefüggéseit és különbségeit fogjuk vizsgálni és bővebben tárgyaljuk a Java programozási nyelvet. Már a könyv elején is említik, hogy a Java a jelölésrendszerében nagyon sokmindent átvett a C++-ból. Az előzőleges tudásunkból pedig tudjuk, hogy a C++ eljárás és objektum orientált nyelv, míg a Java már szimplán az objektum orientált szemléletmódot követi. Az objektum orientált programozás célja, hogy implementálja a valós-világ egyedeit. Az objektum a valós világ egyedeire utal. Míg az objektum orientált programozás egy paradigma arra, hogy olyan programot írjunk, amely osztályokat és objektumokat használ. Az objektumoknak vannak tulajdonságai és van viselkedésük. Az objektumok tulajdonságait változókkal írjuk le általában, míg a viselkedésüket a metódusokkal jellemezzük. Az objektumok lehetnek fizikai vagy logikai dolgok. Emellett az objektum orientáltságnál fontos megemlíteni az öröklődést. Amikor az egyik objektum öröklí minden tulajdonságát és viselkedését a szülőobjektumától.

A Java-ban ezen kívül nagy figyelmet fordítottak a biztonságra és a megbízhatóságra. Ebből következik az, hogy itt már nincsenek pointerok minden referencia. A Javában interpretert használnak, míg a C++-ban compiler végzi a fordítást. A C++ fordító gépi kódra fordítja a programot. Addig a Java fordítóprogramja egy byte kódot hoz létre, amelyet a JVM futtat. Ezért a Java platform független. Míg azért a C++-nál vannak megkötések. A Java program objektumok és ezek blueprintjeinek összessége. Az osztály változókból és metódusokból épül fel. A JVM hátránya a sebesség. (lassabb mint a compiler)

Javában ha több osztályunk van akkor mindig azt fogja futtatni, amelyikben a JVM megtalálja a main-t. Javában a C vagy C++ nyelvtől eltérően a visszatérési típus megadása minden metódus számára kötelező. A C++-tól eltérően itt már egy igazi String szövegtömbbel kerülnek átadásra a paraméterek nem char.

A java-val könnyen írhatunk olyan kódot, amely html-ben futtatható ez az applet. Ilyenkor a html oldalon kívül a java kód is letöltődik a weboldalt megtekintő részére és a távoli gépen fog futni. Persze gondoltak arra, hogy ez veszélyforrást jelenthetne a vírusok miatt, de a java futtatórendszerének erős a biztonsági rendszere. Az applet programoknak fontos jellemzőjük, hogy hiányzik a main metódusuk.

A változók típusai nem különböznek a C++-ban használtaktól.

Javában a final kulcsszóval hivatkozunk rájuk, míg C++-ban const ként. Javában lehetőségünk van ezenkívül az unicode karakterek használatára is mivel már nem a 8 vagy 7 bites karaktereket használja, mint a C++ vagy a C.

A megjegyzések használata megegyezik a C++-éval, kivétel hogy itt már vannak dokumentációs megjegyzések, ami olyan mint a több soros megjegyzés csak 2 db *-al kezdjük. A dokumentációs megjegyzéseket a javadochoz tudjuk használni. Ez kiszedi a kódból a lényeges információkat és egy html oldalként jeleníti meg. Használata olyan mint a fordítása a java programoknak csak nem javacot használunk, hanem javadocot.

Ugyan úgy mint C++-ban a class szóval hívjuk meg. Minden egyes osztály tagnak egyessével adhatjuk meg a láthatóságát. Ha elhagyjuk akkor csak az adott osztályban látható. Az új objektumokat szintén ugyan úgy a new kulcsszóval vagyunk képesek létrehozni. Javában az új objektumok létrehozásukkor inicializálódnak 0 vagy null értékkel. A nyelvben már nem karakter tömb hanem ellenőrzött String osztály szerepel. A static kulcsszó nem egy elemhez tartozik hanem az osztályhoz. Ez azt jelenti, hogy a new alkalmazásakor nem foglalódik le memóriaterület számukra az objektumban. Nem kell inicializálásnál értéket adni nekik. Ezen kívül az osztály nevével is hivatkozhatunk rájuk. Az ezzel megjelölt objektumból tetszőleges számú objektum készítése után is csak egy lesz. A memória felszabadítása úgy történik, hogy egyszerűen már nem hivatkozunk az objektumra, azaz null értéket adunk neki, ezzel szemben C++-ban a delete vagy a free metódust kellett használnunk.

A metódusok a C++-hoz hasonlóan működnek és a kivételkezelés is a try-catch-el. A C++-al ellenben a Java már tartalmazza a párhuzamosítást és a grafikai megoldásokat sem kell különböző megoldásokkal megoldani pl Qt. A javának már van saját grafikai csomagja a Swing. Emellett kikerültek a Javában a nyelből a goto és a const szavak ezek bár foglaltak mégsem használtak.

Literálok:

A típusoknál a primitív típusú típusok mellett megjelentek a csomagoló osztályaik. pl.: az int nek az Integer. Ez annyiban különbözik, hogy ezek objektumhivatkozásokat tartalmaznak. Ezek bizonyos kontextusokban nyerik el valódi jelentőségüket. Ilyenek az adatszerkezetek. A java nyelvben a tömböket ugyan úgy kell megadni viszont a C++-tól eltérően ez már igazi típus és nem csak a mutatók egy másik formája.

Metódusnevek túlterhelése:

Egy osztálynál több metódus is szerepelhet ugyan azzal a névvel, ha a formális paramétereinek a száma vagy típusa eltérő. A java fordító alaphoz tudni fogja ezekből, hogy mikor melyik metódust kell meghívni. C++-ban is hasonlóképpen működik.

Bezárás, adatrejtés:

Az osztályoknál a jellemzőknek és metódusoknak két fajtájuk van. Vannak amelyeket elrejtünk más osztályok elől ez a privát vagy védett (protected). És vannak amelyeket megosztunk másokkal ezek a public információk. Az állapotleíró jellemzőket általában privátként használjuk, míg a metódusok többsége publikus. A jelöletlen tagokra nem hivatkozhat bármelyik osztály csak az azonos csomagban lévők látják. A privát tagokat csak az osztályon belül lehet látni. A protected szorosan összefügg az öröklődéssel, mivel ez a félnyilvános láthatóság kiterjesztése. A protected kategóriájú konstruktort egy más csomagba tartozó gyermek csak a super kulcsszóval képes meghívni.

Osztályhierarchia:

Az osztályok egymáshoz viszonyított összességét osztályhierarchiának nevezzük. Minden osztálynak az Object osztály vagy szülője ha nem használtunk extends-et rajta vagy pedig őse.

Öröklődés vagy inheritance:

Öröklődésnél egy osztály alosztályát hozzuk létre, amely örökli a szülő jellemzőit és metódusait. És ezeket sajátjaként használja. De ezenkívül lehetnek más tulajdonságai és metódusai is. Vagy olyan metódusai,

amely ugyan azt csinálják mint a szülő osztálynak csak másképpen. Az öröklődés egy is-a kapcsolat. Az öröklődés C++-ban a :-al történik, míg javában az extends kulcsszót használjuk. Javában a super kulcsszóval használhatjuk a szülő konstruktorát, de alapvetően a gyerek nem öröklí a konstruktorokat. C++-ban engedélyezett a többszörös öröklődés, de javában nem.

Absztrakció:

Az absztrakció az amikor egy osztályból létrehozunk egy vázlatot nem deklarálunk semmi, csak definiálunk és ebből majd leszármaztatunk osztályokat. Absztrakciónál megfigyeljük, hogy mik a közös tulajdonságai az objektumainknak amikkel dolgozunk és lényegében egy közös interfészt nyújt, amely minden származtatott objektumra egyaránt érvényes.

Polimorfizmus:

A polimorfizmus lehetővé teszi, hogy mivel a gyermek minden olyan tulajdonsággal rendelkezik mint a szülője, ezért minden olyan esetben ahol a szülő használható, ott legyen használható a gyerek is. Felülbíró polimorfizmus mikor a gyermek örököl egy metódust a szülőtől viszont a gyermeknél az a metódus másképpen működik ezért lehetőség van overrideolni a metódusokat. Tehát az a lehetőséget, hogy egy változó nem csak a deklarált típusú, hanem a származtatott objektumra is hivatkozhat polimorfizmusnak nevezzük. Osztálydefiníciókat nem lehet felüldefinálni csak elfedni. Mindig futási időben dől el, hogy a metódus mely implementációját kell meghívni ez a dinamikus kötés. C++-ban a dinamikus kötést a virtual kulcsszóval kellett jeleznünk java-ban minden virtuális kötéssel van ellátva.

```
class Alakzat{
void kiir(){System.out.println("Alakzat vagyok");}
}
class Haromszog extends Alakzat{
    void kiir(){System.out.println("Háromszög vagyok");}
}
class Teglalap extends Alakzat{
    void kiir(){System.out.println("Teglalap vagyok");}
}
class Pelda{
    public static void main(String[] args) {

        Alakzat a=new Alakzat();
        a.kiir();
        a=new Haromszog();
        a.kiir();
        a=new Teglalap();
        a.kiir();

    }

}
```

Interface:

Az interface már a C-óta létezik protokoll néven, innen vette át a java csak módosítva. Az interface egy új referencia típus. Az interface-ben a metódusok megvalósítás nélkül szerepelnek azaz csak deklarálva szerepelnek. Az interface egy felületet definiál. Az interfacek használata egy új absztrakciós szintet jelent, így

el lehetet vonatkoztatni a konkrét implementációtól. Ez megkönnyíti a tervezést és segíti a módosíthatóságot. Az interface használata az implementációján keresztül történik. Egy osztály implementál egy interfacet ha az osztály minden egyes interface által specifikált metódushoz implementációt ad. Ahol interface szerepel típusként ott az interface-t implementáló osztály vagy leszármazottjának példánya felhasználható. Interfaceknél is létezik öröklődés itt kiterjesztésnek nevezzük. Az osztályokéval ellentétben itt lehetséges a többszörös öröklődés. Egy osztály tetszőleges számú interfacet örökölhet, ez kiváltja a C++-os többszörös öröklődést.

A kulcsszavak: Azok a szavak amelyből a nyelv építkezik. Tehát ezek lefoglalt karakterek, amit nem adhatunk meg azonosítónak. Az azonosítók például változónevek azok a szavak amelyek betűvel kezdődnek majd folytatódhatnak számmal vagy betűvel. Emellett vannak a literálok, amelyek nevesített konstansok értékeként vagy kezdőérték adáskor használatosak. Emellett lefoglalt karakterekhez tartoznak az operátorok, amelyek lehetnek egy vagy több operandúsúak. Használatukkor valamilyen művelet (matematikai, logika) hajtódik végre.

Utasítások: kétféle létezik a deklarációs és a kifejezéses. Deklarációs változók létrehozása, metódusok definiálása például. Kifejezéses a metódusok meghívása vagy ilyen például a prefix és a postfix operátorok használata. De az értékadás és példányosítás is. A deklarációs utasítás a változó létrehozását és kezdőérték adását foglalja magába. Ezeknél jön képbe a blokkok használata és a hatáskör egy változónak az adott blokkon belül van hatásköre. Persze van kivétel is.

Konstansok: Javában a final kulcsszóval tudjuk létrehozni őket C++-ban ugyan ezt a const-al érjük el. A konstansoknak mindig nevet adunk és kezdőértéket. A típusuk lehet bármilyen. A konstansok értéke nem változik a program futása során.

Karakterkészlet: a Java a C++ 8 bites karakterkészletével szemben már 16 bites karakterkészletet használ, amely már a magyar szavakat is magába foglalja.

Metódusok: Az osztályok metódusai ugyebár az objektumok viselkedését írják le nem úgy mint a változók, amelyek a tulajdonságukat az objektumoknak.

Konstruktorok: Minden osztálynak van egy konstruktora mind Java-ban mind C++-ban. De emellett írhatunk újabb konstruktorokat. De lesz mindig egy alapértelmezett, amely nem vár paramétert. Ugyebár itt is él a metódus túlterhelés lehetősége. A konstruktor neve megegyezik az osztály nevével. A konstruktorok minden példányosításkor meghívódnak. Öröklődéskor gyakori, hogy a leszármaztatott osztály meghívja a szülője konstruktorát.

Destruktorok: Java-ban garbage collector C++-ban magunknak kell definiálni. Lényegében itt szabadítjuk fel az objektum által lefoglalt területet a memóriában, azaz töröljük az objektumot. A destruktork neve szintén megegyezik az osztály nevével csak itt egy ~ a neve előtt jelzi, hogy ő a destruktork. Alapból létrehozódik egy a compiler által akkor kell sajátot írni ha van dinamikusan lefoglalt memóriánk az osztályban.

Indexelők: Szögletes zárójelekbe írjuk főként tömböknél használjuk őket.

A beágyazott osztályok lényegében osztályok az osztályban ezzel jelezzük a szoros kapcsolatot például, hogy egyik nem létezhet a másik nélkül, mert önmagában nem értelmezhető.

Gyűjtemények: olyan adatszerkezetek amelyek több objektumot tudnak tárolni, akár különböző típusúakat is. C++-ban tömböt létrehozhatunk mutatók segítségével is maga a tömb egy olyan mutató amely más mutatókra mutat, amelyek a tömbben tárolt elemek címére mutatnak. Java-ban nincsenek mutatók tehát ilyet nem tehetünk meg.

Lambda kifejezések: egysoros utasítások amelyek lerövidítik és egyszerűbbé teszik a kódot általában a visszatérési értékük határozza meg a típusukat. Szerepel mind a C++-ban a C++11 től és a Java-ban is.

Lényegében egyszeri felhasználású kódok, ezért is nem nevezzük el őket azaz névtelenek. Viszont más a definiálásuk. Nézzük is meg:

```
C++-ban a lambda kifejezés:  
[ captures ] <tparams>(optional) (c++20) ( params ) specifiers ←  
exception attr -> ret requires(optional) (c++20) { body }  
Captures lehet ==jellel érték szerint másolni vagy ←  
referenciaként &-el lehetséges. () a paraméterek kerülnek ←  
ide amiket átadunk a lambdának. Majd végül a -> operátor ←  
után jöhet a body {}- hogy mit csináljon a lambda utasítás.  
Javában a lambda kifejezés:  
lambda operator -> body  
pl: (p) -> System.out.println("One parameter: " + p);
```

Streamok: Javában mint minden ez is objektumként van jelen. Tehát osztály felel a streamokért. C++-ban ez byte sorozatokként jelenik meg, tehát adatfolyamatokban (streamokban). A C-ben 3 előredefiniált és megnyitott állomány leírója. Az első az stdin ami a standard input C++-ban a cin. A második az stdout a standard output vagy C++-ban a cout. És az stderr a szabványos hibakimenet. Ezek magas szintű állomány-leírók az stdin csak olvasható a másik kettő csak írható. C++-ban leírók helyett már objektumok vannak. Az istream típusú objektumok csak olvasható bemeneti adatfolyamatokat, míg az ostream csak írható kimeneti adatfolyamatokat takarnak. Ezek az adatfolyamatok a következő operátorokat használják:

```
beolvasás >> és kiírás << operátorok
```

A használatához be kell építenünk a az istream állományt. C++-ban az állománykezelések is adatfolyamatokat használnak. A kétirányú adatfolyamat az fstream valósítja meg. Az állományok megnyitását a konstruktorok végzik lezárásukat a destruktorok. Az adatfolyamat mögött létezik egy buffer, amely kezeli a beírt adatokat. Az adatfolyam buffer itt egy olyan objektum, amely rdbuf tagfüggvényének paraméter nélküli változatával lekérdezhető az adatfolyamba beírt karakterek.

Serializáció: Az objektumokat byte sorozattá alakítja. A használatához egy interface szükséges a Serializable.

Kivételkezelés: Lényegében ez a hibakezelés, ha futási időben valamilyen probléma történik, amire számítnunk, hogy bekövetkezhessen. Például: rossz file név, kevés bemeneti argumentum stb. Akkor ezeket képesek vagyunk kezelni és a futást a hibakezelő ágon folytatni. Ez C++-ban és Javában is a try catch segítségével történik. De persze nem csak hiba felmerülésekor dobhatunk exceptiont azaz kivételt. Javában természetesen ez is objektum lesz mivel osztályokon alapszik az egész. És persze kiegészül a try-catch szerkezet egy finally-val.

Annotációk: @-al jelöljük őket a kódban. Ez lényegében a fordítónak nyújt segítséget. Nagyon gyakran fordul elő Javában az @Override. Lényegében ez azt mondja meg, hogy felüldefiniáljuk a függvényét a szülő osztálynak. Tehát a fordítónak ellenőriznie kell, hogy ez megtörténik-e.

Multiparadigmás nyelvek: Ahogy a neve is sugallja ezek olyan nyelvek, amelyek több programozási paradigmát támogat. Ezzel nyújt nagyobb szabadságot a programozónak. Tehát több eszközt is nyújt a programozónak egy probléma megoldására. És a programozó feladata a számára leghasznosabb kiválasztása.

19.2. Python

A python 1990-ben alkotta meg Guido van Rossum. Ez egy magas szintű, dinamikus, objektum orientált programozási nyelv. Pythonban a fejlesztés sokkal könnyebb mint Javában vagy C/C++-ban. Tökéletes választás a prototípusok elkészítéséhez vagy algoritmusok teszteléséhez. Rengetek modult tartalmaz. Ezek lefedik a hálózatkezelés, felhasználói felület kialakítása, fájlkezelés, rendszerhívások területet és ezen kívül még sok mást is. A pythonnak nincs szüksége fordításra vagy linkelésre. A kódok pythonba sokkal rövidebbek mint java-ban vagy c++-ban. A magasszintű adattípusok segítségével összetett kifejezéseket valósíthatunk meg röviden. A csoportosítás egyszerű tagolással működik nem kell zárójelezni. Nem kell a változókat vagy argumentumokat definiálni, ezen kívül a ; is elhagyható az utasítások végéről. Ha több sorban akarunk utasítást használni a \-t kell használni. A python kis és nagybetű érzékeny. Az értelmezője mindent tokenekre bont. Tokenfajták: azonosítók, kulcsszók, operátorok, literálok, delimiterek. A megjegyzéseket a #-el írhatjuk, ez a sor végéig tart. A pythonba minden adatot objektum reprezentál. Itt nem muszáj változó típust adni, ez automatikusan történik érték alapján.

Adatszerkezetek: ennesek sima zárójellel hivatkozunk rájuk az értékek közé vesszőt teszünk, listák kocka zárójellel hivatkozunk rá, szótárak kapcsos zárójellel hivatkozunk rájuk kulcs-érték párokat használ. Pythonban a null értékre None-val hivatkozunk. Szekvenciákat a + jellel fűzhetünk össze. Az indexek 0-nál kezdődnek, és ezekre a :-al hivatkozhatunk.

Változók: objektumokra mutató referenciák, nincs típusuk. Ugyan úgy mint a javánál ha egy objektumra nem hivatkozik már semmi, akkor a garbage collector felszabadítja. Egy változó hozzárendelést a del kulcsszóval törölhetünk. A függvényben felvett változók alapértelmezetten lokálisak lesznek, ha globálisat akarunk létrehozni a függvény elején kell és a global kulcsszót kell elé írunk.

Lista műveletek: count megadja az előfordulások számát, append hozzáfűz a lista végére, extend egy másik listát fűz a lista végére, insert beszúr egy elemet a megadott helyre. Remove eltávolítja az első előfordulást a listából, pop az i edik elemet távolítja el. Revers megfordítja az elemek sorrendjét. Sort: sorba rendez.

Kiíratás konzolra maradt a print. Az if hasonló mint a többi nyelven kivéve, hogy itt else if helyett elif van. A függvények is hasonlóak kivéve a fornál a 3 as tagolásból kettő lett csak meg kell adni min lépkedjen végig a ciklus. Támogatja a goto utasítást. Függvényeket a def kulcsszóval hozhatunk létre. Függvényeknek egy visszatérési értéke van, de akár ennesekkel is visszatérhet.

Ahogy már említettem a python támogatja az objektum orientált programozást. Osztályokat a class szóval hozhatunk létre. Az osztályoknak lehetnek attribútumai és függvényei. Ezen kívül van inheritance is azaz öröklődés. Az osztályoknak lehet egy speciális konstruktor metódusuk az __init__. A speciális tulajdonságú változókat és függvényeket jelezzük még úgy, hogy két alulvonás közé írjuk őket. Az init függvény első paramétere a self azaz az objektum maga.

Modulok: Sokat segítenek a mobilapplikáció fejlesztésben. PL.: messaging modul az sms üzenetek kezelését, sysinfo a telefon adatainak lekérdezése imei, aksi töltöttségi szintje.

A kivételkezelés szintén támogatott és hasonló a C++-hoz try-catch szerkezet.

IV. rész

Irodalomjegyzék

19.3. Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

19.4. C

[KERNIGHANRITCHIE] Kernighan, Brian W. & Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

19.5. C++

[BMECPP] Benedek, Zoltán & Levendovszky, Tihamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

19.6. 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 UDPROG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEACHackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésükért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPROG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségben született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.