

Univerzális programozás

Így neveld a programozód!

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, Bátfai, Margaréta, Ács Pálffy, Viktor	2020. december 8.	

REVISION HISTORY

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

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	4
2. Helló, Turing!	6
2.1. Végtelen ciklus	6
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	7
2.4. Labdapattogás	8
2.5. Szóhossz és a Linus Torvalds féle BogomIPS	9
2.6. Helló, Google!	9
2.7. A Monty Hall probléma	9
2.8. 100 éves a Brun tétel	10
2.9. Vörös Pipacs Pokol/csigafolytonos mozgási parancsokkal	10
3. Helló, Chomsky!	11
3.1. Decimálisból unárisba átváltó Turing gép	11
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	12
3.3. Hivatkozási nyelv	12
3.4. Saját lexikális elemző	12
3.5. Leetspeak	13

3.6. A források olvasása	13
3.7. Logikus	15
3.8. Deklaráció	15
3.9. Vörös Pipacs Pokol/csigá diszkrét mozgási parancsokkal	17
4. Helló, Caesar!	18
4.1. double** háromszögmátrix	18
4.2. C EXOR titkosító	19
4.3. Java EXOR titkosító	20
4.4. C EXOR törő	21
4.5. Neurális OR, AND és EXOR kapu	23
4.6. Hiba-visszaterjesztéses perceptron	25
4.7. Vörös Pipacs Pokol/írd ki, mit lát Steve	25
5. Helló, Mandelbrot!	26
5.1. A Mandelbrot halmaz	26
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	27
5.3. Biomorfok	28
5.4. A Mandelbrot halmaz CUDA megvalósítása	29
5.5. Mandelbrot nagyító és utazó C++ nyelven	29
5.6. Mandelbrot nagyító és utazó Java nyelven	30
5.7. Vörös Pipacs Pokol/fel a láváig és vissza	31
6. Helló, Welch!	32
6.1. Első osztályom	32
6.2. LZW	32
6.3. Fabejálás	33
6.4. Tag a gyökér	34
6.5. Mutató a gyökér	34
6.6. Mozgató szemantika	35
6.7. Vörös Pipacs Pokol/5x5x5 ObservationFromGrid	36
7. Helló, Conway!	37
7.1. Hangyaszimulációk	37
7.2. Java életjáték	39
7.3. Qt C++ életjáték	40
7.4. BrainB Benchmark	41
7.5. Vörös Pipacs Pokol/19 RF	41

8. Helló, Schwarzenegger!	42
8.1. Szoftmax Py MNIST	42
8.2. Mély MNIST	42
8.3. Minecraft-MALMÖ	43
8.4. Vörös Pipacs Pokol/javíts a 19 RF-en	43
9. Helló, Chaitin!	44
9.1. Iteratív és rekurzív faktoriális Lisp-ben	44
9.2. Gimp Scheme Script-fu: króm effekt	45
9.3. Gimp Scheme Script-fu: név mandala	46
9.4. Vörös Pipacs Pokol/javíts tovább a javított 19 RF-edet	47
10. Helló, Gutenberg!	48
10.1. Juhász István - Magas szintű programozási nyelvek 1 olvasónaplója	48
10.2. Kernighan és Richie olvasónaplója	49
10.3. Benedek Zoltán, Levendovszky Tihamér - Szoftverfejlesztés C++ nyelven olvasónaplója	50
10.4. Python nyelvi bevezetés	51
III. Második felvonás	52
11. Helló, Berners-Lee!	54
11.1. Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II és Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven olvasónapló	54
11.2. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba	59
12. Helló, Arroway!	61
12.1. OO szemlélet	61
12.2. Gagy	63
12.3. Yoda	64
12.4. Kódolás from scratch	66
13. Helló, Liskov!	69
13.1. Liskov helyettesítés sértése	69
13.2. Szülő-gyerek	71
13.3. Anti OO	73
13.4. Ciklomatikus komplexitás	76

14. Helló, Mandelbrot!	79
14.1. Reverse engineering UML osztálydiagram	79
14.2. Forward engineering UML osztálydiagram	80
14.3. Egy esettan	82
14.4. BPMN	88
15. Helló, Chomsky!	90
15.1. Encoding	90
15.2. Leetspeak	91
15.3. Full screen	93
15.4. Paszigráfia Rapszódia OpenGL full screen vizualizáció	95
16. Helló, Stroustrup!	98
16.1. JDK osztályok	98
16.2. Másoló-mozgató szemantika + Összefoglaló	100
16.3. Hibásan implementált RSA törése	105
16.4. Változó argumentumszámú ctor	108
17. Helló, Gödel!	111
17.1. C++11 Custom Allocator	111
17.2. STL map érték szerinti rendezése	113
17.3. Alternatív Tabella rendezése	116
17.4. GIMP Scheme hack	119
18. Helló, !	124
18.1. FUTURE tevékenység editor	124
18.2. OOCWC Boost ASIO hálózatzkezelése	129
18.3. SamuCam	130
18.4. BrainB	133
19. Helló, Lauda!	137
19.1. Port scan	137
19.2. AOP	139
19.3. Junit teszt	142

20. Helló, Calvin!	145
20.1. MNIST	145
20.2. Deep MNIST	148
20.3. SMNIST for Machines	151
 IV. Irodalomjegyzék	 155
20.4. Általános	156
20.5. C	156
20.6. C++	156
20.7. Lisp	156

Táblázatok jegyzéke

13.1. Összehasonlítás	76
---------------------------------	----

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 (hogy lássuk mást is) példával.

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

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

1.2. Milyen doksikat olvassak el?

- Kezd ezzel: <http://esr.fsf.hu/hacker-howto.html>!
- Olvasgasd aztán a kézikönyv lapjait, kezd a **man man** parancs kiadásával. A C programozásban a 3-as szintű lapokat fogod nézegetni, például az első feladat kapcsán ezt a **man 3 sleep** lapot
- C kapcsán a [KERNIGHANRITCHIE] könyv adott részei.
- C++ kapcsán a [BMECPP] könyv adott részei.
- Az igazi kockák persze csemegéznek a C nyelvi szabvány ISO/IEC 9899:2017 kódcsipeteiből is.
- Amiből viszont a legeslegjobban lehet tanulni, az a [The GNU C Reference Manual](https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf), mert gcc specifikus és programozókra van hangolva: szinte csak 1-2 lényegi mondat és apró, lényegi kódcsipetek! Aki pdf-ben jobban szereti olvasni: <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf>
- Az R kódok olvasása kis általános tapasztalat után automatikusan, erőfeszítés nélkül menni fog. A Python nincs ennyire a spektrum magától értetődő végén, ezért ahhoz olvasd el a [BMECPP] könyv - 20 oldalas gyorstalpaló részét.

1.3. Milyen filmeket nézzek meg?

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

- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.

DRAFT

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyTuring/ciklus.c

Egy magot 100%-on dolgoztatni egész egyszerű feladat, hiszen ha egy szimpla while ciklust megírunk, az alapvetően így működik. Egy magot 0%-on dolgoztatni sem egy egetrengető kihívás, viszont itt már kell minimálisan gondolkodni. De hamar rájövünk, hogy a sleep(x) parancs kiadásával x másodpercig nem használja a processzort a program. Viszont ha nincs parancs a cikluson belül, a processzor ugyanúgy 100 %-on dolgozik, ami abból adódik, hogy az operációs rendszer elvégzendő feladatnak tekinti és neki adja az összes processzoridőt. Viszont az összes magot 100 %-on dolgoztatni nehezebb feladatnak bizonyult. Az egyik ismerősöm tanácsára az OpenMP-t kezdtem el nézegetni a feladat megoldásához. Pár fórumbejegyzés és egy kis utánajárás után pedig sikerült a feladatot megoldanom.

A programot roppant egyszerű használni. Ha egy magot szeretnénk 100%-ban dolgoztatni, akkor semmit nem kell módosítani, szimplán csak le kell fordítani és futtatni.

Ha egy magot szeretnénk 100%-ban dolgoztatni, akkor vegyük ki a // -t a

```
//sleep(1)
```

függvényhívásból.

Ha pedig az összes magot szeretnénk 100%-ban dolgoztatni, akkor ugyanúgy a // -t kell kitörölni a következő helyről:

```
#pragma omp parallel while
```

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:

Nem tudunk olyan programot írni, ami minden más programról eldönti, hogy van e benne végtelen ciklus. Mivel, ha tudnánk, akkor már valószínűleg lett volna olyan ember, aki ezt a programot megírja.

De tegyük fel, hogy megírjuk ezt a programot, aminek a neve legyen eldöntő. Annak a programnak a neve, amelyről el kell dönteni, hogy van e benne végtelen ciklus, legyen eldöntendő. Nyilván az eldöntő bemeneti argumentuma lesz az eldöntendő. Ahhoz, hogy eldöntő megállapítsa, hogy van e eldöntendőben végtelen ciklus, futtatnia kell az eldöntendő kérdéses részleteit. Ekkor ha az eldöntendő programban nincs végtelen ciklus, eldöntő hamissal tér vissza, ami azt jelenti, hogy nincs eldöntendőben végtelen ciklus.

Azonban ha az eldöntendő programban tényleg van egy végtelen ciklus, és azt eldöntő futtatja, hogy megbizonyosodjon róla, akkor eldöntő maga is egy végtelen ciklussá válik. Éppen ezért eldöntő sose fog igazgal visszatérni, mert minden ilyen esetben ő is le fog fagyni.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyenTuring/felcserelo.c

Annak ellenére, hogy nem használhatunk logikai utasításokat/kifejezéseket, ez egy egyszerű feladatnak bizonyul egy kezdő programozó számára is. Itt most négy módját is bemutatom.

A program bekér két számot és eredményül a kettő értékének a felcseréltjét adja vissza.

```
int main()
{
    int a;
    int b;
    int c;
    printf("Kerem a felcserelni kivant szamokat! \n");
    scanf("%d", a);
    scanf("%d", b);
    printf("A ket szam: \n");
    printf("A = " "%d", a);
    printf("B = " "%d", b);

    //Első változat:
    /*
        c = a;
        a = b;
        b = c;
    */
    //Második változat:
    /*
        a = a-b;

```

```
        b = a+b;
        a = b-a;
    */
    //Harmadik változat:
    /*
        a = a*b;
        b = a/b;
        a = a/b;
    */
    //Negyedik változat:
    /*
        a=a^b;
        b=a^b;
        a=a^b;
    */
    printf("A ket szam felcserelve: \n");
    printf("A = " "%d", a);
    printf("B = " "%d", b);

    return 0;
}
```

2.4. Labdapattogás

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

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

Megoldás forrása [Bátfai Norbert](#) tulajdona.

Ez egy egyszerű program, ami a grafikus megjelenítést imitálja. Egy labdának álcázott o betűt mozgat egy while ciklus segítségével. Ez a program egy nagyon jó kezdet a grafikus felületű programokkal való megismerkedéshez.

A program két fő részből áll. Az egyik egy függvény, ami a labdát rajzolja ki a konzolra, A másik pedig maga a main.

A main-ben először létrehozunk egy maxX és egy maxY változót, amiket át is adunk a tx és a ty tömbök méretének.

Ezután két for ciklus végigmegy a két tömbön, a második, és az utolsó elemek értéke -1 lesz, a többi elem pedig 1

végül pedig egy while ciklus és a függvény segítségével kiírja a konzolra a labdát.

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: [Saját videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/bogo.cpp

A BogoMips a processzorunk sebességének meghatározásához használatos mértékegység. Azt mondja meg, hogy a számítógép processzora mekkora szóhosszal dolgozik.

Ezt a $XOR \wedge$ művelet segítségével számolja ki a program, ami a kizáró vagy művelete. Az int értékének 1-et adunk, és addig shifteljük balra, ameddig lehet, vagyis amíg az int értéke egyenlő nem lesz nullával.

Közben egy másik változóval számoljuk, hogy hányszor shiftelt balra az int, ezzel meghatározva a szóhosszt. Az én esetemben az eredmény 32 lett, ami a virtuális gép miatt van. Valójában 64 bit-es a processzorom ami 8 bajtnak felel meg.

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/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/pagerank.c

Én a Bevezetés a Programozásba nevű tárgyon már átnézett [PageRank](#) programot vettem alapnak, és azt alakítottam át C++-ból C-re.

2.7. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Megoldás videó: https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall_paradoxon_kapcsan

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

A megoldás Bátfai Norbert tulajdona.

A Monty Hall problémát még középiskolában ismertem meg, sok különböző változata van. Az általam ismert történetben Monty Hall egy műsorvezető volt, akinél a nyertes játékosok választhattak három darab ajtó közül. A háromból két ajtó mögött 1-1 darab kecske, míg a harmadik ajtó mögött egy sportautó volt. A játékos választott egy ajtót, majd Monty Hall, aki tudta, hogy melyik ajtó mögött van az autó, kinyitott egy másik ajtót, ami mögött egy kecske lapult. Ezek után a játékosnak lehetősége volt változtatni a döntésén, vagy maradhatott az eredetileg kiválasztott ajtónál. A kérdés az, hogy mely esetben van több esélye megnyerni az autót? A legtöbb ember azt mondaná, hogy 50-50% esélye van megnyerni az autót, hiszen vagy

az egyik ajtó mögött van az autó, vagy a másik mögött. Ekkor persze hiába magyarázzuk, hogy $1/3$ esélye van megnyerni az autót, ha nem vált, és $2/3$ ha vált, a legtöbb embert elég nehéz meggyőzni erről. Ekkor kell kicsit átalakítani a kérdést. Ha van 1 millió ajtó, ebből kiválaszt a játékos 1-et, majd kinyitnak 999,998 ajtót, amik mögött kecske van, akkor melyik esetben van több esélye a játékosnak megnyerni az autót? Ilyenkor már a legtöbb ember egyértelműnek tartja, hogy vált, de van olyan ismerősöm, aki még ekkor is azt mondta, hogy 50-50% esélye van megnyerni az autót, ha vált ha nem. Ez a program ennek a játéknak a nyerési eseteit szimulálja. Tízmillió esetből hányszor nyer az, aki mindig vált, és az aki egyáltalán nem vált.

2.8. 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 megoldás Bátfai Norbert tulajdona.

Mint tudjuk, léteznek a prímszámok. Ezek olyan számok, amik csak 1-el és önmagukkal oszthatóak. Valamint léteznek az ikerprímek. Ezek pedig olyan prímszámpárok, amiknek a különbsége pontosan 2. Ha minden ikerprím reciprokának az összegének vesszük a sorozatát, akkor ez a sorozat egy számhoz konvergál. Ez a szám a Brun-konstans. Nem tudjuk azt, hogy az ikerprímek száma véges vagy végtelen e, de ez nem okoz gondot, hiszen elvileg ha végtelen se lépi túl az összegük a Brun-konstanst. Na most be kell vallanom, hogy számtalan olyan ember létezik a földön, aki nálam jobban ért a matematikához. Viszont nekem erről egy elég érdekes dolog jutott eszembe, ami nem más, mint Zeno paradoxona. E szerint x utat teszünk meg, hogy elérjük a célunkat. Ezek alapján megteszünk $1/2x$ utat + $1/4x$ utat + $1/8x$ utat + $1/16x$ utat... Ha ezekből képzünk egy sorozatot, az a sorozat 1-hez fog konvergálni, Éppen ezért soha nem érünk el oda, ahova megyünk. Maga a tétel matematikailag helyes, azonban a való életben tudjuk, hogy ez nem így működik.

2.9. Vörös Pipacs Pokol/csiga folytonos mozgási parancsokkal

Megoldás videó: <https://youtu.be/uA6RHzXH840>

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ebben a feladatban a karaktert folytonos mozgásra bírtuk, különböző módokon. Ez egy nagyon kezdetleges lépés volt, de mint már azt tudjuk, kis lépés egy karakternek, óriási a hackernek :).

együtt a levont egyeseket a tába helyezi. Ezt minden számjeggyel megismétli.

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:

Az $a^n b^n c^n$ nyelv nem környezetfüggetlen. Na de először értsük meg, hogy mit is jelent ez a nyelv.

Az $a^n b^n c^n$ annyit jelent, hogy n darab a , majd n darab b , majd végül n darab c áll egymás után. Ezek a terminális szimbólumok. A szabály alapján a környezetfüggő nyelveknél bal oldalon csak egy önmagában álló nem-terminális szimbólum állhat. Azonban nem létezik olyan képzési szabály ami alapján ez a szabály teljesíthető, ezért ez a nyelv nem környezetfüggetlen.

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: [Saját videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyiChomsky/nyelvek.c

A BNF (Backus-Naur-forma) használatával környezetfüggetlen nyelveket lehet leírni. Nagyon sok programozási nyelv szintaxisa is BNF-ben vannak leírva.

A programozási nyelveknek is van nyelvtana, illetve nyelvtani szabályaik. Az egyik ilyen szabály C89-ben az, hogy a for ciklus fejrésében nem lehetett változót deklarálni, éppen ezért ha a következőképpen szeretnénk lefordítani a fenti programot:

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

Akkor a következő hibaüzenetet kapjuk:

```
nyelv.c:3:2: error: 'for' loop initial declarations are only allowed in C99 or C11 mode
```

Ez pontosan leírja nekünk, hogy a for ciklusban deklarálni csak c99 vagy c11 módban lehet.

Éppen ezért ha `-std=c89`-et kihagyjuk és nem írunk oda semmit, vagy `-std=c99`-et írunk, akkor a program gond nélkül lefordul.

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/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyChomsky/lex.l

A megoldás forrása [Bátfai Norbert](#) tulajdona.

A lexer-rel szövegelemző programokat lehet generálni az általunk megadott szabályok alapján. A program különböző részeit % jelekkel kell elválasztani egymástól. Itt a numbers változóban fogjuk számolni a valós számok darabszámát. Majd megmondjuk, hogy a digit egy 0 és 9 között lévő számot jelöl. Ezek után jön az a kódrészlet, ami megmondja a lexernek, hogy a valós számokat számolja meg. Végül pedig kiírjuk a valós számok darabszámát. A futtatáshoz először is telepítenünk kell a lex-et majd a forrásban található programot kell megírni.

Majd azt a következőképp kell lefordítanunk:

```
lex -o lex.c lex.l
```

```
gcc -o lex lex.c -lfl
```

Ezzel megkapjuk a https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyChomsky/lex.c oldalon található programot, ami a feladat megoldása.

3.5. Leetspeak

Lexelj össze egy l33t ciphert!

Megoldás videó: https://youtu.be/06C_PqDpD_k

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyChomsky/lex.l

Megoldás forrása: A megoldás forrása [Bátfai Norbert](#) tulajdona.

A leet (1337, saját formában írva) egy, az internettel együtt elterjedt szleng nyelv, amiben a betűket különböző számokként, és egyéb ASCII karakterekként, a számokat pedig különböző betűkként ábrázoljuk.

Itt is érvényes az a szabály, hogy a program egyes részeit % jelekkel kell elválasztani egymástól. Itt a legelső részben a cipher struktúrában meg vannak adva a karakterek leet formái

A második részben történik az igazi munka, először a szöveget kisbetűssé alakítja a program, majd pedig végigmegy a szövegen és minden karaktert a neki megfelelő leet formájú karakterre alakítja át.

A harmadik és egyben utolsó részben található a main() amiben a lex meghívása történik.

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 jelkezelő 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 megyránézésre, elkapja valamelyiket esetleg a splint vagy a frama?

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

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:

1: Ha kapunk egy INTERACT szignált, akkor a jelkezelő függvénnyel eldöntjük, hogy mihez kezdünk azzal a szignállal, mit reagáljon rá a program.

2: Egy for ciklus ami nullától négyig megy és a ciklus törzsében lévő művelet elvégzése előtt nő az értéke eggyel.

3: Szintén egy for ciklus, ami szintén nullától négyig megy, viszont itt már a ciklus törzsében lévő műveletek elvégzése után növekszik az értéke.

4: Egy for ciklus, ami berakja a tomb[i]-edik helyére az i értékénél eggyel nagyobb értéket, és közben i értékét is növeli.

5: Egy for ciklus, ami addig megy, amíg i kisebb mint n, illetve amíg a d és s pointerek értékei megegyeznek.

6: Kiírunk két, az f nevű függvény által generált számot. Az egyik szám az a majd a eggyel megnövelt értékének a feldolgozásából jön létre, míg a másik szám a+1 és a feldolgozásából. Fontos a sorrend.

7: Szintén két számot írunk ki, az egyik szám az f nevű függvény által feldolgozott a nevű számból előállt érték, a másik pedig a értéke.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

Első értelmezése: Minden számra igaz, hogy létezik tőle nagyobb y prímszám.

Második értelmezése: Minden számra igaz, hogy létezik egy olyan tőle nagyobb y prímszám, hogy $y+2$ is prímszám.

Harmadik értelmezése: Létezik olyan szám, amitől minden prímszám kisebb.

Negyedik értelmezése: Létezik olyan szám, amitől egyik kisebb szám se prímszám.

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; //létrehoz egy egész típusú változót`
- `int *b = &a; //egy pointer, ami a memóriacímére hivatkozik`
- `int &r = a; //egy referencia a-ra`
- `int c[5]; //5 elemű tömb aminek c a neve`
- `int (&tr)[5] = c; //egy tr nevű referencia c-re`
- `int *d[5]; //egy 5 elemű pointerekből álló tömb`
- `int *h (); //Egy egészre mutató mutatót visszaadó függvény`
- `int *(*l) (); //Egy egészre mutató mutatóra mutató mutatót visszaadó ↔
függvény`
- `int (*v (int c)) (int a, int b) //Függvénytmutató, ami egy egészet ↔
visszaadó függvényre mutató mutatóval visszatérő függvény`
- `int ((*z) (int)) (int, int); //Függvénytmutató, ami egy egészet visszaadó ↔
függvényre mutató mutatót visszaadó függvényre mutat`

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyChomsky/deklaracio.cpp

Egész:

```
int a;
```

Egészre mutató mutató:

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

Egész referenciája:

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

Itt fontos megjegyezni, hogy c-ben nincs referencia, ezért ezt a kódcsipetet érdemes g++-al fordítani gcc helyett.

Egészek tömbje:

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

Egészek tömbjének referenciája (nem az első elemé):

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

Egészre mutató mutatók tömbje:

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

Egészre mutató mutatót visszaadó függvény:

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

Egészre mutató mutatót visszaadó függvényre mutató mutató:

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

Egészre visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény:

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

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

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

3.9. Vörös Pipacs Pokol/csiga diszkrét mozgási parancsokkal

Megoldás videó: <https://youtu.be/Fc33ByQ6mh8>

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ebben a feladatban ugyanazt csináltuk mint az előző hetiben, viszont itt diszkrét mozgási parancsokat kell megadnunk a karakternek.

4. fejezet

Helló, Caesar!

4.1. double** háromszögmátrix

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

Megoldás videó: [Saját videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_Igy_Caesar/tm.c

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Először is egy alapfogalom: Az alsó háromszög mátrixnak ugyanannyi sora van, mint oszlopa. Ezen kívül még egy nagyon fontos tényezője az is, hogy a főátlója felett csak 0 szerepel.

Az ilyen mátrixokat, ha tömbökben tároljuk, akkor nincs értelme a nullákat is tárolni a többi, számunkra fontos elemmel együtt, ezért ezeket nem is tároljuk. Amikor egy ilyen tömböt vissza szeretnénk alakítani az eredeti alakjára, akkor sorfolytonosan írjuk fel az elemeit. Ez annyit jelent, hogy a mátrix első sorába az első elemet írjuk fel, a második sorába a második és harmadik elemet és így tovább minden sorban eggyel több elemet írunk fel mint az előző sorban.

Ebben a programban egy ilyen alsó háromszög mátrixot hozunk létre egy

```
double **
```

segítségével. Ez egy pointerre mutató pointer, ami tökéletes a többdimenziós tömbök használatához.

Ezek után a következő kis programrészlet:

```
if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
{
    return -1;
}
printf("%p\n", tm);

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

```

    {
        return -1;
    }
}

```

Ellenőrzi, hogy történt-e valamilyen memóriahiba, (pl. nincs tele a memória?) és ha történt, akkor -1-el tér vissza.

Ellenkező esetben a program a

```
tm[i][j] = i * (i + 1) / 2 + j;
```

képletet használva feltölti a tömböt. Ezután két egymásba ágyazott for ciklus segítségével kiírja azt. Ezek után módosítunk a tömb egyes elemein, majd megint kiírjuk őket.

Legvégül, pedig a

```

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

```

függvény használatával felszabadítjuk a tömbnek lefoglalt helyet.

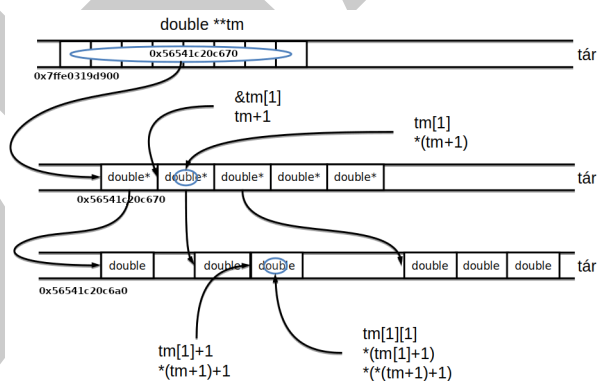
A program futtatásnál a következő memóriacímeket írta ki:

```

./tm
0x7ffe0319d900
0x56541c20c670
0x56541c20c6a0

```

Aminek a jelentése:



4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó: [Saját videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyiCaesar/exor.c

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Ez a fajta titkosítás a kizáró vagy műveleten alapul. A megadott kulcs, és a forrásfájl karaktereit kizáró vaggal titkosítva egy szöveget úgy tudunk titkosítani, hogy egy olvashatatlan karakterhalmazt kapunk végeredményül. Viszont az a személy, aki ismeri a kulcsot ugyanolyan egyszerűen vissza is tudja alakítani a szöveget az eredeti alakjára úgy, hogy még egyszer lefuttatja a programot, de a titkosított forrást adja meg titkosítandóként, ezzel visszakapva az eredeti szöveget. Így más nem tudja elolvasni a titkainkat, csak az, aki ismeri hozzá a kulcsot.

Először is a

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

használatával megadjuk a maximális kulcs és buffer méretet. a main osztály első argumentuma a kulcs lesz, míg a második az maga a szöveg, amit titkosítani szeretnénk.

A következő ciklusok használatával:

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

program végigmegy a bemeneti adatok (titkosítandó fájl) karakterein, mindegyiket titkosítja a kulcs használatával és kiírja a végeredményt.

A program használata: `./exor kulcs <titkosítandó fájl> titkosított fájl`

Erre egy példa: `./exor 12345678 <lista> titkoslista`

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó: [Saját videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyiCaesar/exor.java

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Itt az előző feladatban megírt EXOR titkosítót írjuk át java nyelvre. Ehhez importálnunk kell az input/output streamet. Ez ahhoz kell, hogy olvasni tudjuk a bemeneti fájlt, illetve, hogy írni tudjuk a kimeneti fájlt.

A main-ben megpróbáljuk a try-al beolvasni az args tömbbe azt a fájlt, amit titkosítani szeretnénk és ha ez nem sikerült, akkor "elkapjuk" a hibát a catch szerkezettel, és kiírjuk, hogy mi a hiba:


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

        try {

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

        } catch(java.io.IOException e) {

            e.printStackTrace();

        }

    }
}
```

Ha viszont sikerült beolvasni a fájlt, akkor az ExorTitkosító nevű függvényt meghívva előállítjuk a titkosított szöveget. A System.in illetve System.out a bemenő és a kimenő fájlra utalnak.

Először a függvény átadja a program a kulcs nevű tömbjének a bemenő szöveget, és létrehoz egy buffer nevű tömböt is 256-os mérettel. Erre az EXOR művelethez lesz szükség.

Végül a program egy while-ba épített for ciklus segítségével végigzongorázza a szöveget, minden egyes karakternek meghatározza a titkosított verzióját, és kiírja azt a kimeneti fájlba.

4.4. C EXOR törő

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

Megoldás videó: [Saját videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyiCaesar/tores.c

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Elfelejtetted egy EXOR-ral titkosított fájlod kulcsát? Szeretnél kutakodni mások fájljai között, de azok titkosítva vannak?

Ne is kénldj tovább. Az EXOR törőt neked találták ki! A program sikerességéhez mindössze annyit kell tudnod, hogy hány karakterből áll a kulcs, és máris használhatod ezt a fantasztikus programot!

A működése nagyon egyszerű. Mivel nem ismerjük a kulcsot, ezért a program az összes lehetséges kombinációt végigpróbálja. A következőkben bemutatott példában a kulcs 8 darab karakterből áll.

Legelőször a program a következő while ciklus

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

használatával beolvassa a feltörni kívánt fájlt, majd a maradék helyet a bufferben, egy for ciklust használva,

```
for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
titkos[p - titkos + i] = '\\0';
```

feltölti 0 értékekkel.

Ezek után egy csomó (ami jelen esetben 8) for ciklussal

```
#pragma omp parallel for private(kulcs)
for (int ii = '0'; ii <= '9'; ++ii)
for (int ji = '0'; ji <= '9'; ++ji)
for (int ki = '0'; ki <= '9'; ++ki)
for (int li = '0'; li <= '9'; ++li)
for (int mi = '0'; mi <= '9'; ++mi)
for (int ni = '0'; ni <= '9'; ++ni)
for (int oi = '0'; oi <= '9'; ++oi)
for (int pi = '0'; pi <= '9'; ++pi)
{
    kulcs[0] = ii;
    kulcs[1] = ji;
    kulcs[2] = ki;
    kulcs[3] = li;
    kulcs[4] = mi;
    kulcs[5] = ni;
    kulcs[6] = oi;
    kulcs[7] = pi;
    exor_tores (kulcs, KULCS_MERET, titkos, ←
                p - titkos);
}
```

megpróbálja a program előállítani az eredeti szöveget. Azonban több kombináció is ad eredményt, ezért nekünk kell kitalálni, hogy a kapott eredmények közül melyik a helyes. Ezek a for ciklusok az összes magot dolgoztatni fogják, ezzel jelentősen lecsökkentve a töréshez szükséges időt.

Ha a kulcs nem 8 karakterből áll, akkor sem kell aggódnunk! Csupán néhány (pontosan 3) szekcióban kell módosítani a program kódját. Ezek a következők:

Először is a program fejében a

```
[#define KULCS_MERET 8
```

sorban a 8-at át kell írni arra a számra, amennyi karakterből áll a kulcs.

Majd a 70. és 71. sorokban lévő

```
[ 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);
```

utasításokban annyi **%c** és **kulcs[n]** legyen, amennyi karakterből áll a kulcs.

Végül pedig az előzőekben már látott for ciklus halmon kell módosítanunk úgy, hogy pontosan annyi **for** ciklus, és pontosan annyi **kulcs[n] = xi**; legyen a programban, amennyi karakterből áll a kulcs.

Most ezeknek az ismeretében, a programot a következőképpen kell fordítani: **gcc tores.c -fopenmp -o tores -std=c99**

És futtatni: **./tores <titkosfájl**

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

A megoldás forrása Bátfai Norbert tulajdona.

Egy neurális háló neuronok sokaságából áll. A neuronok ingerlékeny sejtek, amelyek képesek az ingerek felvételére és azok továbbítására is. Az emberi agyban ez a hálózat felel az információ feldolgozásáért. Egy ilyen hálózatot kiválóan lehet programozni viszonylag egyszerűen.

R-ben fogjuk ezt a feladatot megoldani, így biztosítanunk kell a megfelelő környezetet.

```
sudo apt install r-base
```

Ha ez megvan, telepíteni kell még R-en belül egy csomagot, hogy tudjunk neurális hálókat modellezni.

```
install.packages("neuralnet")
```

Amennyiben ezeket sikeresen telepítettük, neki is láthatunk a háló betanításának. Elsőként az OR műveletet tanítjuk meg neki:

```
library(neuralnet)
x1 <- c(0,1,0,1)
x2 <- c(0,0,1,1)
OR <- c(0,1,1,1)

or.data <- data.frame(x1, x2, OR)

nn.or <- neuralnet(OR~x1+x2, or.data, hidden=0, linear.output=FALSE,
stepmax = 1e+07, threshold = 0.000001)

plot(nn.or)

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

A kód elején változókat definiálunk, aztán ezeket az `or.data`-ban tároljuk.

Az `nn.or` értéke a `neuralnet` visszatérési értéke lesz.

```
nn.or <- neuralnet(OR~x1+x2, or.data, hidden=0, linear.output=FALSE,
stepmax = 1e+07, threshold = 0.000001))
```

Ez a függvény elég sok paramétert kap; nézzük meg ezeket egyesével:

`OR~x1+x2` - a formula, amit a program meg kell, hogy tanuljon

`or.data` - a minta, ami alapján tanul

`hidden=0` - a rejtett neuronok száma

`linear.output=FALSE` - megadja, hogy a függvény a kimeneti neuronokra ne fusson le

`stepmax=1e+07` - a maximális lépésszám

`threshold=0.000001` - a folyamat leállításának kritériuma

Ezek után kirajzoltatjuk a hálót a

```
plot(nn.or)
```

parancs használatával.

Ugyanezzel a módszerrel tanítjuk fel a hálónak az AND műveletet is. Ebben az esetben az AND értékeit változtatjuk meg, hogy passzoljon a logikai formulánkoz.

```
x1    <- c(0,1,0,1)
x2    <- c(0,0,1,1)
OR     <- c(0,1,1,1)
AND    <- c(0,0,0,1)
```

```
orand.data <- data.frame(x1, x2, OR, AND)
```

```
nn.orand <- neuralnet(OR+AND~x1+x2, orand.data, hidden=0, linear.output= <-
  FALSE, stepmax = 1e+07, threshold = 0.000001)
```

```
plot(nn.orand)
```

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

Legvégül pedig az EXOR műveletet tanítatjuk meg vele.

```
x1    <- c(0,1,0,1)
x2    <- c(0,0,1,1)
EXOR   <- c(0,1,1,0)
```

```
exor.data <- data.frame(x1, x2, EXOR)
```

```
nn.exor <- neuralnet(EXOR~x1+x2, exor.data, hidden=0, linear.output=FALSE, <-
  stepmax = 1e+07, threshold = 0.000001)
```

```
plot(nn.exor)
```

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

Ez azonban ránézésre is helytelen eredményt fog produkálni, így módosítanunk kell a 'hidden' paramétert, hogy helyes értéket adjon.

```
x1    <- c(0,1,0,1)
x2    <- c(0,0,1,1)
EXOR   <- c(0,1,1,0)
```

```
exor.data <- data.frame(x1, x2, EXOR)

nn.exor <- neuralnet(EXOR~x1+x2, exor.data, hidden=c(6, 4, 6), linear. <-
  output=FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

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

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása: <https://github.com/nbatfai/nahshon/blob/master/ql.hpp#L64>

A megoldás forrása Bátfai Norbert tulajdona.

Perceptronról a Mesterséges Intelligenciák, és a neurális hálók témakörében lehet szó. Ellenőrzi a bemenetet, és egy feltétel alapján eldönti, hogy mi legyen a kimenet, Egy példa:

Van három bemeneti adatunk amikbe pozitív egész számokat várunk. Ha a három bemeneti számból kettő kisebb mint 0, akkor a kimeneti adat -1 lesz, ha viszont a háromból legalább kettő pozitív szám, akkor a számok összege lesz a kimeneti adat.

Ekkor kimondhatjuk, hogy 1 a hibahatár, mert ekkor még megkapjuk az általunk kért dolgot, viszont ha már kettőt hibáztunk akkor már -1 lesz a válasz.

Ezt a hibahatárt szokták finomítani. Nagyon magas hibahatárnál kezdenek, és egyre kisebbé teszik egészen addig, amíg elfogadható a hibák mennyisége.

Persze a mi három bemeneti adatok példánknál nem sokat lehet finomítani, de ha több millió bemeneti adatról beszélünk, ott ez egy elég fontos dolog.

4.7. Vörös Pipacs Pokol/írd ki, mit lát Steve

Megoldás videó: <https://youtu.be/-GX8dzGqTdM>

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ebben a feladatban a kódhoz hozzáadunk egy olyan részt, ami kiírja, hogy mi található a karakter 3x3-as környezetében, merre néz a karakter, és azt is, hogy éppen mire néz. A cél az, hogy a karakter kiírja a képernyőre, hogy éppen egy vörös virágot lát.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás videó: [https: Saját videó](https://www.youtube.com/watch?v=...)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyiMandelbrot/mandelbrot.cpp

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Mielőtt bármihez hozzáfazdenénk egy nagyon fontos információ. Ahhoz, hogy leforduljon a programunk, szükséges a png++. Ezt a legegyszerűbben a **sudo apt install png++** paranccsal lehet megtenni. Most hogy ezt letudtuk, jöhet pár alapvető információ.

A Mandelbrot halmaz lényege az, hogy komplex számokkal, és egy egyenlettel dolgozik. Azok a számok amelyek kielégítik ezt az egyenletet egy nagyon szép képet alkotnak, ha levetítjük őket egy kétdimenziós síkra.

A program legelején includeoljuk a png++-t, hiszen nagyrészt ezt fogja használni a program.

```
#include <png++-0.2.9/png.hpp>
```

Ezek után létrehozunk végleges értékeket N-nek és M-nek, valamint megadjuk X és Y lehetséges minimum és maximum értékét is.

```
#define N 500
#define M 500
#define MAXX 0.7
#define MINX -2.0
#define MAXY 1.35
#define MINY -1.35
```

két egymásba ágyazott for ciklus használatával megadjuk a C, a Z, és a Zuj nevű Komplex számok valós és imaginárius értékét. Ezek korábban lettek létrehozva a mainen belül, és a Komplex nevű struktúrához tartoznak.

```
struct Komplex
{
```

```
double re, im;  
};
```

```
struct Komplex C, Z, Zuj;
```

Végül pedig a `GeneratePNG(tomb)` nevű függvény használatával a program legenerálja a PNG fájlt pixelről pixelre.

A programot a következőképpen tudjuk fordítani: `g++ mandelbrot.cpp -lpng16 -o mandelbrot` Futtatni pedig a szokásos módon a `./mandelbrot` parancsal tudjuk.

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

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyN/Mandelbrot/mandelbrotkomplex.cpp

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az [UDPROG](#) repóban.

Ebben a feladatban a végeredmény ugyan az kellene hogy legyen, mint az előző feladatban. Illetve azóta még a mandelbrot halmaz lényege sem változott.

A `png++` ebben az esetben is kelleni fog, így ha nincs leszedve, akkor pillants az előző feladat magyarázatára, ahol megtalálod a szükséges dolgokat ahhoz, hogy le tudjon fordulni a program.

Ebben az esetben az `std::complex` osztályt fogjuk használni a program megvalósításához. Ez az osztály, ahogy a neve is utal rá, a komplex számok kezelése miatt jött létre.

A program által használt függvényei a következők:

A `real(C)` a komplex szám valós részét határozza meg.

A `imag(C)` a komplex szám képzetes részét határozza meg.

Legelőször a program

```
#include <png++-0.2.9/png.hpp>  
#include <complex>
```

beincludeolja a `png++`-t és a komplex osztályt

Ezek után az előző feladathoz hasonlóan itt is megadjuk a végleges értékeket az `N`, `M` valamint `X` és `Y` maximum és minimum értékeinek.

Legnagyobb részben ennek a feladatnak a megoldása megegyezik az előző feladat megoldásával, ezért azt nem írnám le újra, inkább arra koncentrálnék, hogy miben más ez a forrás mint az előző.

Az érdemi különbség a két forrás között az az, hogy itt az `std::complex` osztályt használva, már nem kell létrehoznunk egy saját struktúrát a komplex számoknak.

Ehelyett szimplán létrehozzuk a `double` típusú komplex számokat a következőképpen:

```
std::complex<double> C, Z, Zuj;
```

Illetve a for cikluson belül sem a struktúrán belüli elemek imaginárius és valós részére hivatkozunk, hanem a `real()` és `imag()` nevű függvényeket meghívva mondjuk meg a komplex szám részeinek értékét.

```
real(C) = MINX + j * dx;  
imag(C) = MAXY - i * dy;
```

A programot fordítani és futtatni ugyan úgy kell, mint az előző feladatot.

5.3. Biomorfok

Megoldás videó: [Saját videó](#)

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

A megoldás forrása [Bátfai Norbert](#) tulajdona.

A két előző feladathoz hasonlóan itt is szükségünk van a `png++` ra, ezért ha még nem szedted le akkor pillants rá a az első feladat magyarázatára, ahol részletesen le vannak írva az ehez szükséges parancsok.

Ez egy olyan mandelbrot program, ahol maga a user adja meg a határokat. Előnye hogy az eredetihez képest teljesen más képeket kapunk, hátránya viszont hogy ha a user nem tudja, hogy mit csinál akkor az egész kép egy nagy fekete semmi lesz.

Először is

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

includeoljuk az `iostream`et a `png++`t és a `complex` osztályt.

A main argumentumai a bemeneti adatok, amikből előállítjuk magát a képet.

Ellenőrzi a program, hogy megfelelő mennyiségű bemeneti értéket adott e meg a felhasználó, és ha nem, akkor felvilágosítja, hogy hogyan kell használni a programot.

```
if ( argc == 12 )  
{  
    szelesseg = atoi ( argv[2] );  
    magassag =  atoi ( argv[3] );  
    iteraciosHatar =  atoi ( argv[4] );  
    xmin = atof ( argv[5] );  
    xmax = atof ( argv[6] );  
    ymin = atof ( argv[7] );  
    ymax = atof ( argv[8] );  
    reC = atof ( argv[9] );  
    imC = atof ( argv[10] );  
    R = atof ( argv[11] );  
}  
else  
{  
    std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c ↵  
    d reC imC R" << std::endl;
```



```
    return -1;  
}
```

Ha viszont megfelelő mennyiségű argumentumot adott meg a felhasználó, akkor létrehozza a képet aminek a szélessége és a magassága a felhasználó által megadott szélesség és magasság lesz.

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

Ezek után a program két egymásba ágyazott for ciklus segítségével kiszámolja, létrehozza a képet és el menti a felhasználó által megadott néven.

A fordítása az előző két programhoz hasonlóan működik, a futtatása azonban már így néz ki:

./3.1.3 fajlnev szelesseg magassag n a b c d reC imC R Erre egy példa:

./3.1.3 biomorf.png 800 800 10 -2 2 -2 2 .285 0 10

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/CUDA/mandelpngc_60x60_100

A megoldás forrása Bátfai Norbert tulajdona.

Először is közérdekű közlemény, hogy ennek a programnak a sikeres fordításához szükségünk lesz egy CUDA magokat használó NVIDIA kártyára, illetve az nvidia-cuda-toolkit re amit a következő paranccsal tudunk feltenni:

sudo apt install nvidia-cuda-toolkit

Ez a program ugyanúgy a mandelbrot halmazt rajzolja ki, mint az előzőek, azonban itt egy nagyon fontos különbség az, hogy míg az előző feladatoknál a képet a CPU számolta ki és készítette el, addig itt, az NVIDIA kártyák CUDA magjait használjuk a kép kiszámításához.

Ez azért fontos, mert az előző feladatoknál egyetlen egy mag dolgozott és számolt ki mindent, addig itt az én GTX 1660 SUPER videokártyám esetében 1408 darab cuda mag számolná és rajzolná ki a képet. Azért írom feltételes módban, mert a virtuális gép egyik hátránya, hogy nem tudja a gép minden részét kihasználni.

Ez nyilvánvalóan egy sokszor gyorsabb futási időt eredményez. Az én esetemben például amikor CPU-val futtattam volna a programot akkor egy kb 20 másodperces futási idő jött volna ki.

Azonban ha már a fentebb említett GTX 1660 SUPER kártyát használva futtatom a programot, akkor már egy kicsit hamarabb lefut a program.

A programot a gcc helyett az nvcc nevű paranccsal kell fordítani. Futtatni pedig a szokásos módon.

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 videó: https://bhaxor.blog.hu/2018/09/02/ismerkedes_a_mandelbrot_halmazzal.

Megoldás forrása: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/binom/Batfai-Barki/frak/>

Visszatérünk a Mandebrot halmazokhoz, de most nem csak egy képet szeretnénk végeredményként kapni, hanem egy olyan ablakot, ahol képesek vagyunk ránagyítani a kapott halmazunkra. Ez lehetséges egy képfájl esetén is, de ott egy idő után a képet nem lehet lesz értelmezni, olyan szinten pixeles lesz.

Amit mi itt használni fogunk ennél egy kicsit bonyolultabb, emiatt talán ez a fejezet legnehezebb feladata. A megoldás erre a problémára az, hogy az egérrel kijelölt részre az ablak nem ránagyít, hanem újragenerálja azt a részt. Így elkerülhető az a probléma, hogy nagyon pixelessé válik a kép, ugyanis a pixelszám azonos lesz.

Az első probléma még a kódolás előtt megjelenik, hogyan készítsük el ezt az ablakot. Egy nagyszerű megoldás a Qt eszköztár használata, amelyben elkészíthető egy gui(grafikus interfész). Ennek az interfésznek a futtatásához 5 kódrészre van szükségünk, ezek szerepei a következők:

frakablak.h

Ebben a kódrészben egy osztályt hozunk létre, aminek van védett, privát és publikus tagja is. A publikus részben a tartományokat/határokat szabjuk meg, amik között mozoghatunk. A védett részben egy függvényt deklarálunk, amely figyelemmel követi, mit csinál a felhasználó. Ennek fontos szerepe lesz, hogy a program tudja azt, hogy mikor a halmazra ránagyítania. Végül a privát tagban a nagyítandó terület van meghatározva.

frakszal.h

Ismét egy osztályt láthatunk és még pár változót. Ezek segítenek majd a programnak a számolásban és rajzolásban.

frakszal.cpp

Ebben a kódrészben történik a legfontosabb dolog: a Madelbrot halmaz elemeinek kiszámolása, ehhez hasonló programot írtunk az első feladatokban is.

main.cpp

Itt kerül meghívásra a Qt konstruktor, és itt tesszük a kódhoz a Qt könyvtárat.

frak.pro

A frak.pro lesz a "főnök", alatta fognak futni az előbb bemutatott kódok, az ő feladata lesz a teljes folyamat kezelése.

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás Forrása: https://github.com/Viktorpalffy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgylMandelbrot/MandelJava

A megoldás forrása [Bátfai Norbert](#) tulajdona.

A feladat az ezt megelőző feladat átírása Java nyelvre. A GUI megírásához szükség van egy keretrendszerre, ami jelen esetben az Abstract Window Toolkit lesz.

Először nézzük a *Mandelbrothalmaz.java* fájlt.

A main-ben a `MandelbrotHalmaz()` meghívásával létrehozunk egy új halmazt a megadott paraméterekkel. Ezek a paraméterek a tartományok koordinátái, a halmazt tartalmazó tömb szélessége, és a számítás pontossága.

Utána a felhasználó tevékenységeit figyeli a program, és megfelelően reagál rájuk, valamint a GUI ablak tulajdonságait adja meg, illetve kirajzolja magának a halmaznak a képét.

A következő fájlunk a *MandelbrotHalmazNagyító.java*

A nevéből adódóan ez végzi a halmazon a nagyítást, illetve magának a halmaznak a kirajzolását is. Maga a `MandelbrotHalmazNagyító` osztály figyeli a felhasználó egér tevékenységeit, azzal kapcsolatban, hogy hol szeretné nagyítani a képet, illetve kirajzolja az új, nagyított képet. Ezen kívül ez végzi az elmentendő képek készítését, és elmentését is.

Végül pedig a *MandelbrotIterációk.java* fájl szerepe.

Ez a programrészlet a nagyított mandelbrot halmazok pontjait tartja nyilván. Ez egy számításra létrehozott osztály, ami a kiválasztott ponthoz tartó utat mutatja meg.

5.7. Vörös Pipacs Pokol/fel a láváig és vissza

Megoldás videó: <https://youtu.be/I6n8acZoyoo>

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ebben a feladatban az a lényeg, hogy a karakter fusson fel az aréna aljától a tetejéig, majd ha a 3x3-as látóterében lávát érzékel, akkor a lávával együtt szépen sétáljon vissza le az aréna aljáig.

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás videó:

C++ forrás: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/polargen/>

Java forrás: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/kezdo/elsojava/PolarGen.java#l10>

Ehhez a programhoz java-ban szükségünk lesz az `util.random`, az `io.*` illetve a `lang.math` java könyvtárakra. Először is a `bExist` változót hamisra állítjuk a konstruktoron belül, majd pedig inicializálunk egy randomot, és gyakorlatilag ennyi a konstruktor.

Ezek után a `PolarGet` függvény az, ami az érdemi munkát végzi. Először is ellenőrzi, hogy volt-e már generálás. Ha volt akkor azt adja vissza, ha nem, akkor a matekos algoritmus segítségével legenerálja a két random normált és `bExists`-et átállítja az ellentétjére.

6.2. LZW

Valósítsd meg C++-ban az LZW algoritmus fa-építését!

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyelWelch/lzw.cpp

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az **UDPROG** repóban.

Mindkét esetben a bináris fa felépítésének a lépései a következők:

Ha 1-est szeretnénk betenni a fába, akkor először megnézzük, hogy az aktuális csomópontnak van-e már ilyen eleme. Ha még nincs, akkor egyszerűen betesszük neki az 1-es gyermekének az 1-et. Azonban ha már van ilyen gyermeke, akkor létre kell hozni egy új csomópontot és az ő gyermekének adjuk át az 1-et.

Ez hasonlóan működik akkor is, ha nullást szeretnénk betenni, annyi különbséggel, hogy nem az egyeseket vizsgáljuk, hanem a nullásokat. Ezt a lépést a programban a következő rész oldja meg:

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

A megadott fájl tartalma alapján felépíti az LZWBInfa csomópontjait. Jelen esetben ezt a Bináris Fát in order bejárással dolgozzuk fel, ami annyit jelent, hogy először a fa bal oldalát dolgozzuk fel, majd a fának a gyökerét, és legvégül pedig a jobb oldalt. A következő feladatban ezen viszont már változtatunk.

Fordítása a szokásos módon történik a futtatása pedig a következőképpen:

`./lzw bemenet -o kimenet`

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyiWelch/fabe.cpp

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az **UDPROG** repóban.

Az előző feladatban tárgyalt fát In Order módszerrel járta be a program. Ez azt jelenti, hogy először a részfa bal oldalát dolgozzuk fel, majd a részfa gyökerét, és utoljára pedig a részfa jobb oldalát.

Erre ugyanúgy megmaradt a lehetőségünk, csupán a következőképp kell futtatni a programot:

`./lzw bemenet -o kimenet i`

Ezzel szemben itt két másik fajta bejárési módszerrel dolgozzuk fel a fát. Az egyik a Pre Order bejárési mód, a másik pedig a Post Order.

A Pre Order bejárési módnál először a részfa gyökerét dolgozzuk fel, másodjára a részfa bal oldalát, és utoljára pedig a részfa jobb oldalát. A Pre Order bejárési mód használatához a következőképpen kell futtatni a programot:

./lzw bemenet -o kimenet r

A Post Order bejárési módnál pedig legelőször a részfa bal oldalát dolgozza fel a program, majd a jobb oldalát, és utoljára pedig a részfa gyökerét. A Post Order bejáráshoz a következő parancs használatával kell futtatni a programot:

./lzw bemenet -o kimenet r

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://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyelWelch/tag.cpp

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az **UDPROG** repóban.

Ez a program az eredeti Bevezetés a Programozásba tárgyon már tanult *z3a7.cpp* nevű program szerint működik, hiszem itt a csomópont már kompozícióban van a fával. Az egész az LZWBInfa osztállyal kezdődik, aminek van privát, és publikus része is. A publikus részen belül található a konstruktor, és a destruktor deklarációja. Itt kerülnek vizsgálatra a bemenő elemek, és jönnek létre a nullás illetve egyes elemek is. Túlterhelődik az operátor, és megvizsgálja a program, hogy létezik-e már nullás gyermek. Ha nincs, akkor létrejön. Egyes gyermeknél ugyanez a helyzet.

A kiír függvény pedig kiírja a csomópontokat.

Majd jön az LZWBInfa privát része. Itt található meg a Csomópont osztály amin belül a konstruktor megkapja a gyökeret. Még a Csomópont osztályon belül találhatóak azok a függvények, amivel le tudjuk kérdezni, hogy ki az aktuális csomópont nullás illetve egyes gyermeke, valamint az `ujNullasGyermek()` illetve `ujEgyesGyermek()` függvények, amik létrehozzák az új nullás és egyes gyermekeket

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/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyelWelch/gyoker.cpp

Ehhez a feladathoz az **UDPROG** repóban megtalálható BinFa programot vettem alapul.

Ebben a megoldásban az előző feladathoz képest kicsit másképp a megoldás. A következő dolgokat kell átírni a már meglévő programban:

Először is a 315. sorban a csomopont után tegyünk egy *-ot ezzel mutatóvá téve a gyökeret. Ha így megpróbáljuk lefordítani a programot akkor nagyon sok szintaktikai hibát fogunk kapni a fordítótól válaszként. Nem kell megijedni. Az a dolgunk, hogy ezeket a hibákat egyesével kijavítsuk. Az első két hiba kijavításához a következő részletet kell átírni.

A 92. és a 93. sorban a

```
szabadit (gyoker.egyenesGyermekek ());  
szabadit (gyoker.nullasGyermekek ());
```

utasítások helyett a

```
szabadit (gyoker->egyenesGyermekek ());  
szabadit (gyoker->nullasGyermekek ());
```

utasításokat kell használni.

Ez után már kettővel kevesebb hibát kapunk. Az összes többi hibát a referenciák okozzák. Ahoz hogy ezeket a hibákat megoldjuk a következő sorokban kell tevékenykednünk: 92, 132, 147, 170, 210, 336, 344, és 356. Azonban a hibát minden sorban ugyan azzal a módszerrel kell javítani, ami nem más mint hogy a

```
&gyoker
```

helyett azt kell írni hogy

```
gyoker
```

Vagyis kiszedjük a referenciákat, mivel alapból a memóriacímek lesznek átadva.

Ezek után a programunk ugyan lefordul, de amikor megpróbáljuk futtatni, akkor szegmentálási hibát kapunk. Ennek a javításához a konstruktort kell átírni a következőképpen:

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

Ezzel foglalunk helyet a memóriában a gyökérnek. Viszont amit lefoglalunk, azt fel is kell szabadítani, éppen ezért a destruktort is módosítani kell a következőképpen:

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

Mostmár fel is szabadul, amit lefoglaltunk.

6.6. Mozgató szemantika

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

Megoldás videó: [Védési videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyiWelch/mozgato.cpp

A megoldás forrásának az alapja megtalálható az [UDPROG](#) repóban. Én ezt módosítottam.

Maga az LZWBinFa osztály felépítése úgy néz ki, hogy az osztályon belül léteznek a beágyazott csomópont objektumok amik a fát alkotják. Ezek alapján a fa másolása nem más, mint ezeknek a csomópontoknak a másolása. Ehhez létre kell hoznunk a mozgató illetve mozgató értékadás konstruktorokat.

```
LZWBinFa (LZWBinFa&& masik){
    gyoker=nullptr;
    *this= std::move(masik);
}

LZWBinFa& operator= (LZWBinFa&& masik){
    std::swap(gyoker,masik.gyoker);
    return *this;
}
```

Először a mozgató értékadásról (alsó) szólnék pár szót, ami csupán annyit jelent, hogy ha egyenlőségjel operátort használunk, akkor az `std::swap()` függvénnyel megcserélődik a két gyökér mutatója.

Másodszor pedig a mozgató konstruktor. Itt először is `nullptr` (nullpointer) értéket adunk abban a binfában lévő gyökérnek, amelyik fába akarjuk mozgatni a ("masik") fát. Majd a "masik" nevű fát átmozgatjuk az `std::move()` függvénnyel, ami annyit jelent, hogy a gyökér mutató mostmár a paraméterként kapott "masik" fára mutat, ami azért történhetett meg, mert az `std::move()` függvény tulajdonképpen nem is mozgat semmit, hanem a paraméterül kapott értéket jobbérték referenciává alakítja.

6.7. Vörös Pipacs Pokol/5x5x5 ObservationFromGrid

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ebben a feladatban a karakter látóterét 3x3x3-asról 5x5x5-ösre bővítettük, ezáltal az eddigi kettő szint helyett már hármat is tudunk egyszerre vizsgálni. Az 5x5x5-ös blokkban lévő elemeket pedig kiíratjuk vele a konzolra, vagy csak szimplán felhasználhatjuk az adatokat.

7. fejezet

Helló, Conway!

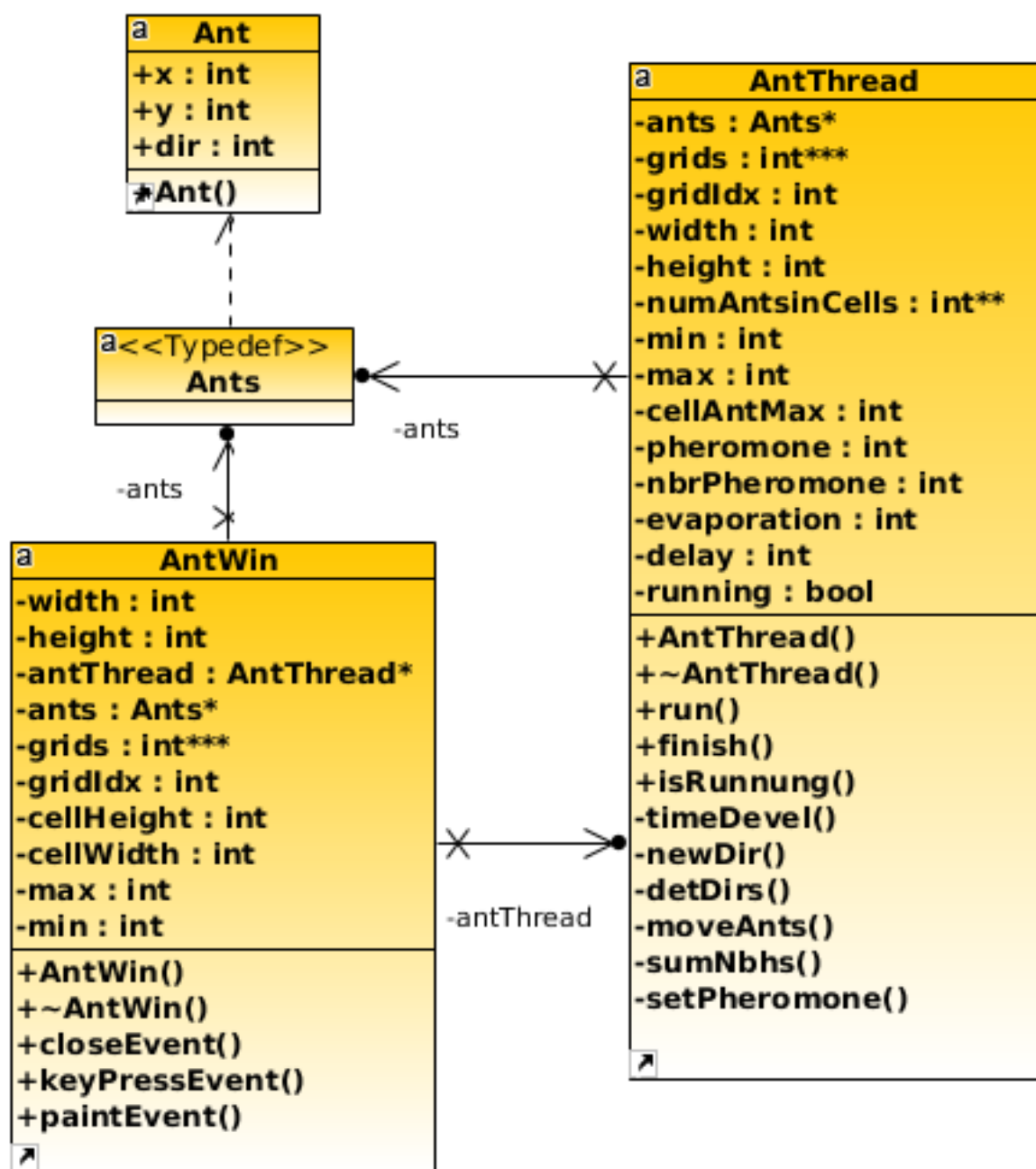
7.1. Hangyaszimulációk

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://github.com/Viktorpalfy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyNConway/Ant

Az osztálydiagram:



A megoldás forrása, illetve az UML osztálydiagram [Bátfai Norbert](#) tulajdona.

Ebben a feladatban hangyákat kell szimulálnunk. A megoldás azt a biológiából már ismert tényt használja, hogy a hangyák a való életben szagokkal (feromonokkal) kommunikálnak egymással. Ha például egy hangya valamilyen érdekes dolgot talált, akkor ott hagyja a nyomát, illetve megjelöli az útvonalat. Az éppen arra járó többi hangya ezt megérzi, a legfrissebb feromon nyomát követve ők is el fognak jutni a célba. Ezeket felhasználva készítették el ezt a hangyaszimulációt.

Az osztálydiagrammon belül négy egységet találhatunk, ezek a következők: **Ant**; **Ants**; **AntWin**; és **AntThread**.

Ezek a programunk osztályai, ezeken az egységeken belül vannak megadva az adott osztály változó és függvényei is. Ilyen például az **AntWin** egységen belül található *width* és *height* változók, amik a képernyő

hosszúságát és szélességét adják meg. Vagy például a `closeEvent()` és a `keyPressEvent()` függvények, amik szintén az `AntWin` osztály részei. Ezek alapján meghatározhatjuk, hogy az `AntWin` osztály a szimuláción belül a világot kezeli.

Az `AntThread` osztály kezeli a hangyákat, mozgásukat, illetve a virtuális feromonok terjedéséért is ez az osztály felelős.

7.2. Java életjáték

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

Megoldás videó: [Saját videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_Igykon Conway/életjatek.java

A Megoldás forrása [Bátfai Norbert](#) tulajdona.

Arról, hogy mi az az életjáték, illetve, hogy mik a szabályai, a következő feladat leírásában részletesebben kifejtem. Röviden az életjáték szabályai:

Ha egy négyzetnek pontosan három élő szomszédja van, akkor abban a négyzetben egy új sejt jön létre.

Ha egy már élő sejtnek pontosan kettő vagy három darab szomszédja van, akkor az a sejt továbbra is életben marad.

Ha viszont egy már meglévő élő sejtnek háromnál több élő szomszédja van (túlnépesedés), vagy kettőnél kevesebb, akkor az a sejt meghal. Ezt szimulálja az életjáték.

Ezek a szabályok az `időFejlődés()` függvényben vannak lefektetve.

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

Igaz ugyan, hogy az életjáték egy úgynevezett nullszemélyes játék, de ebben a példában a játékos mégis tudja irányítani kicsit a dolgokat. Ugyanis a program figyeli a billentyűzet bizonyos gombjait (`k`, `n`, `l`, `g`, `s`), illetve az egér mozgását, és kattintásait is. Ezt három függvénnyel teszi. Az `addKeyListener(new java.awt.event.KeyAdapter())` függvénnyel figyeli a billentyűzetet. Ezen a függvényen belül egy `if-else` szerkezet állapítja meg, hogy éppen melyik gombot nyomta le a felhasználó a

```
if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K){}
```

feltétel a K betű lenyomását azonosítja, és csökkenti a sejtek méretét.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N){}
```

feltétel az N betű lenyomását azonosítja, és növeli a sejtek méretét.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S){}
```

feltétel az S betű lenyomását azonosítja, és készít egy képet a sejtterről.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
```

feltétel a G betű lenyomását azonosítja, gyorsítja a szimuláció sebességét.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L
```

feltétel az L betű lenyomását azonosítja, és lassítja a szimuláció sebességét.

Az egér mozgását, illetve kattintásait pedig a `addMouseListener(new java.awt.event.MouseAdapter()` illetve a `addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {})` függvények figyelik. Az egér kattintásaival egy sejt állapotát tudjuk megváltoztatni. Az egér mozgásával pedig az összes érintett sejt élő állapotba kerül.

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalfy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyNConway/Qt

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Az életjátékot John Conway találta ki, és nem teljesen hiteles rá a játék kifejezés, mert ez egy úgynevezett nullszemélyes játék. Magának a játékosnak annyi a dolga, hogy megadja a kezdőalakzatot, majd pedig megfigyelheti, hogy mi lesz az eredmény.

Alapja egy négyzetrácsos tér, amikben élhetnek sejtek, de minden egyes négyzetben csak egy darab sejt élhet. A játék szabályai a következők:

Ha egy sejtnak kettő vagy három élő szomszédos sejtje van, akkor a sejt meg fogja élni a következő generációt. Az összes többi esetben viszont kihal a sejt, akár azért mert túl sok, akár azért mert túl kevés szomszédja van.

Ahol azonban egy üres négyzetrácsnak pontosan három élő sejt a szomszédja, akkor ott új sejt jön létre.

Érdekes az, hogy milyen hatást váltott ki az emberekből. Voltak akik napi rutinjukká tették azt, hogy az életjátékkal "játszanak", egyfajta függők lettek, és voltak azok, akik nem értették hogy mi a jó benne.

Maga a program ugyanúgy működik, mint a java verzió. Mind a két program két darab mátrixsal dolgozik, viszont itt a teljes kód megírása helyett a Q-t is segítségül hívjuk.

A programot a következőképpen tudjuk fordítani és futtatni: **qmake Sejtauto.pro make ./Sejtauto**

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalfy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyNConway/BrainB

A megoldás forrása [Bátfai Norbert](#) tulajdona.

A programhoz szükségünk lesz az OpenCV-re, aminek a feltelepítéséhez a lépéseket [ezen a linken](#) elérhető weblapon találjuk.

Ez egy miniatűr játék, ami a felhasználó szem-kéz koordinációjáról, illetve a megfigyelőképességéről gyűjt össze információkat.

Amikor elindítjuk a játékot akkor egy ablak fogad minket, és a lényege az, hogy a **Samu entropy** nevű négyzetben belül lévő fekete pöttyön belül tartuk az egerünk mutatóját.

A játék a teljesítményünk alapján lesz könnyebb, vagy nehezebb. Minél jobban játszunk, annál több Entropy lesz a képernyőn, ezáltal megnehezítve a Samu entropy követését. Viszont ha már nem tudjuk nyomon követni a Samu entropy-t akkor folyamatosan eltünteti a hozzáadott entropy-kat, ezáltal megkönnyítve a játékot.

A programot futtatni az előző feladathoz hasonlóan szintén a **qmake** és **make** parancsokkal lehet fordítani.

7.5. Vörös Pipacs Pokol/19 RF

Megoldás videó: <https://youtu.be/VP0kfvRYD1Y>

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

A Red Flower Hell II. körében a célunk ugyanaz mint eddig volt: a lehető legtöbb pipcsaot gyűjteni, amíg a láva leér az aréna aljába. Azonban az első körhöz képest néhány változtatást hozott Tanár Úr a szabályokban: tilos az XML fájl adatait felhasználni, tilos abszolút mozgási formát használni, tilos játékmódot változtatni, tilos az XML fájlt változtatni és tilos átvinni az egér fölé az irányítást.

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

Megoldás videó: [Saját videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyN_Schwarzenegger

Ebben a feladatban egy neurális hálót építünk Pythonban, a Tensorflow használatával. A Tensorflow a Google gépi tanulási platformja.

Ez a példa 60000 mintából épít fel egy neurális hálót. A mintákon kézzel írt számjegyek láthatóak. A háló célja az, hogy a megtanult minták alapján meg tudja mondani egy képről, hogy azon milyen szám található.

Amint ez a videóban is látható, 5 futtatás után a pontosság egy kicsivel több mint 97 százalék.

8.2. Mély MNIST

Python

Megoldás videó: [Saját videó](#)

Megoldás forrása: https://github.com/Viktorpalfy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyN_Schwarzenegger

Ebben a feladatban az előzőhöz hasonlóan ismét egy neurális hálót tanítunk számjegyek felismerésére. Az előzőleg használt 60000 mintából dolgozunk, viszont itt már eredményt is fogunk kapni. Futtatásnál bemeneti paraméterként megadhatunk neki egy képfájlt, amiről eldönti, hogy milyen számot ábrázol. Ebben az esetben 10 epoch-ra futtatjuk ~98,5%-os pontossággal. Viszont ahogy a videóban látszik, sajnos nem találta el a megadott számot ennek a nagy pontosságnak az ellenére sem (8-ast adtam meg neki és 6-ost kaptam eredményül).

8.3. Minecraft-MALMÖ

Most, hogy már van némi ágensprogramozási gyakorlatod, adj egy rövid általános áttekintést a MALMÖ projektről!

Megoldás videó: initial hack: <https://youtu.be/bAPSu3Rndi8>. Red Flower Hell: <https://github.com/nbatfai/-RedFlowerHell>.

Megoldás forrása: a Red Flower Hell repójában.

A Minecraft Malmö projekt egy mesterséges intelligencia fejlesztésére szolgáló platform. A projekt a Minecraft nevű játékra épül, mert ahogy annak mottója is hirdeti: "Csak a kreatitásod szab hatrárt!". Ezt a játékot sokan csak virtuális LEGO-nak hívják a játékos szabadsága miatt, hogy tényleg bármit el lehet benne készíteni. A projekt nagyon sikeres a fiatalabb tanulók körében, hiszen mégse csak száraz kódot kell írniuk, hanem látják is a munkájuk eredményét egy pillanat alatt.

A BHAX Malmö projektben egy fordított piramis pályán kell minél több vörös pipacsot összegyűjtenie a karakterünknek 300 másodperc alatt, aminek a mozgását vagy mi, vagy pedig az általunk megírt program szabályozza. A célunk az, hogy előbb utóbb egy teljesen "önműködő" karaktert tudjunk létrehozni, aki a saját belátása szerint, az általa legoptimálisabbnak ítélt módon szedje fel az lehető legtöbb virágot, mielőtt a láva a pálya aljára ér, ezzel őt megölve. Hétről hétre egyre több elemmel bővítjük a tudását. Eleinte csak kezdetleges mozgási parancsokat adtunk ki neki, aztán látóteret is kapott (először 3x3x3-ast, azóta 7x7x7-nél járunk), utasításokat, hogy mit tegyen ha a látóterében virág található. A következő kihívás pedig az, hogy mihez kezdjen egy olyan pályával, ami minden futtatás alkalmával más és más.

8.4. Vörös Pipacs Pokol/javíts a 19 RF-en

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ez a feladat megegyezik az előző fejezetben lévő Malmö-s feldattal, csupán annyi benne a különbség, hogy itt a virágokat nem lentről felfelé haladva kezdjük el gyűjteni, hanem a karakter az induláskor azonnal a pálya teteje felé veszi az irányt és onnan halad lefelé a megszokott csigavonalban.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Iteratív megoldás: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyChaitin/iter.lisp

Rekurzív megoldás: https://github.com/Viktorpalfy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyChaitin/rek.lisp

Ebben a feladatban a program faktoriális számol iteratív illetve rekurzív módon. A lisp a második magas szintű programozási nyelv. Egyedül a fortran előzte meg. Először nézzük az iteratív módszert.

```
(defun fact(n)
  (setf f 1)
  (do ((i n (- i 1))) ((= i 1))
    (setf f (* f i)))
  )
fact(4))
```

A program első sorában definiáljuk magát a függvényt `fact` néven. Majd egy `f` nevű változót, aminek az értéket 1-re állítjuk. Ezután jön egy ciklus, ami `i`-nek átadja a számot, aminek a faktoriálisát ki kell számolni. A program `i`-ből folyamatosan kivon 1-et egészen addig, amíg `i` értéke nem lesz egyenlő 1-el. A ciklus törzsében pedig `f` értéke $f*i$ lesz. Végül a program meghívja magát a `fact()` függvényt.

Ezzel szemben a rekurzív módszert valamennyivel könnyebb olvasni.

```
(defun fact(n)
  (if (= n 0) 1
      (* n (fact(- n 1)))
  )
)
fact(4)
```

Először ebben a példában is a `fact()` függvény kerül definiálásra. Azonban ezek után itt egy `if` szerepel, ami azt ellenőrzi, hogy `n` egyenlő-e 0-val. Ha nem, akkor szimplán meghívja a függvény saját magát, azonban itt már $n-1$ amivel számol, így számolva ki a faktoriális értékét.

9.2. Gimp Scheme Script-fu: króm effekt

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

Megoldás videó: https://youtu.be/OKdAkI_c7Sc

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

Ez egy program a Gimp-hez, ami a megadott szöveghez króm effektet ad. Aki nem ismeri a Gimpet, most gyorsan bemutatom: a neve a "GNU Image Manipulation Program" rövidítéséből ered, a program egy ingyenesen használható bittérképes képszerkesztő, ami támogatja a rétegek kezelését és az átlátszóságot is.

Maga a forrás egy tömbbel indul, ami a króm effekt megvalósításához szükséges információkat tartalmazza.

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

A következő függvény a betűk méretét határozza meg. A szükséges méreteket a GIMP a beépített függvényeivel határozza meg a következőképpen:

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

Ez annyit jelent, hogy a `gimp-text-get-extents-fontname` első értékét (ami maga a méret) állítja be a `text-width` illetve a `text-height` változóknak a `set` utasítás használatával.

Majd a `script-fu-bhax-chrome-border` függvény hozza létre a tényleges króm effektet szöveg. Ezt egy új rétegen (layer) teszi. Ennek az új rétegnek a háttere fekete, a rá kerülő szöveg pedig fehér színű lesz.

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

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

Végül a program regisztrálásra kerül a Gimp-ben, hogy el tudjuk érni.

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Az előző feladathoz hasonlóan itt is egy Gimp kiegészítőről van szó. Itt azonban a bemenő szövegből egy név-mandala fog készülni. A mandala egy szimmetrikus kör alakú kép, ami a Hindu vallásban nagy szerepet játszik az isteneik ábrázolásában.

Először a program meghatározza a szöveg hosszát, a `gimp-text-get-extents-fontname` függvény használatával. A kapott értéket a `set!` utasítással a `text-width` változó értékének adja.

```
(set! text-width (car (gimp-text-get-extents-fontname text fontsize PIXELS  
font)))
```

Ebben a feladatban, ugyanúgy határozzuk meg a szöveg méretét, mint az előzőben:

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

A GIMP beépített `gimp-text-get-extents-fontname` függvényét, és a `set!` utasítást felhasználva a `text-width` és a `text-height` változók értékei lesznek a szükséges méretek.

Ezek után jön a mandala. Először létrejön egy réteg (layer)

```
(gimp-image-insert-layer image layer 0 0)  
(gimp-context-set-foreground '(0 255 0))  
(gimp-drawable-fill layer FILL-FOREGROUND)  
(gimp-image-undo-disable image)
```

Amit feltöltünk a felhasználó által megadott adatokkal. Ezek a szöveg, a szöveg betűtípusa. Ezután a réteget tükrözi a program, ezzel elérve a szimmetriát, majd a program elforgatja a képet, és megismétli a tükrözést. Majd a réteget felnagyítja a kép teljes méretére:

```
(gimp-layer-resize-to-image-size textfs)
```

Ezután két körnek álcázott ellipszist illeszt a program a képre ezzel létrehozva a mandalát. Az egyik kör vastagsága 22, a másiké pedig 8 lesz.

```
(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width 2) (/  
textfs-width 2)) 18)  
(- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) (+  
textfs-height 36))  
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)
```

```
(gimp-context-set-brush-size 22)
(gimp-context-set-brush-size 8)
```

Végül pedig megjeleníti a képet:

```
(gimp-display-new image)
```

Ezek után ismét már csak a GIMP-be regisztrálás van hátra.

9.4. Vörös Pipacs Pokol/javíts tovább a javított 19 RF-edet

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

10. fejezet

Helló, Gutenberg!

10.1. Juhász István - Magas szintű programozási nyelvek 1 olvasó-naplója

A számítógépet programozó nyelveknek három szintje van. Ezek a gépi nyelv, az assembly nyelv és a magas szintű nyelv. Mi a magas szintű programozási nyelvekkel foglalkozunk, ami az emberek által a legjobban érthető. Az ilyen nyelven megírt programot nevezzük forrásprogramnak. Azonban a processzorok csak az adott gépi nyelven írt programokat tudják végrehajtani. Ezért forrásprogramot át kell írni gépi kódra. Ezt a munkát végzik el a fordítók.

Minden programnyelvnek van saját szabványa, ez a hivatkozási nyelv. Pontosan meg vannak adva a nyelvtani szabályok, amiket be kell tartani, különben vagy szintaktikai, vagy szemantikai hibát fogunk kapni. A szintaktikai hiba az, amit a fordító észrevesz, és jelez nekünk, hogy gond van. Míg a szemantikai hiba esetén a fordító nem kapja el a hibát, de a program nem megfelelően fog működni.

Ezen kívül minden nyelvnek vannak Adattípusai is. Ezek lehetnek beépített, vagy a programozó által kreáltak is. Ilyen típusok például az egész számok, a karakterek, a karakterláncok, a tömbök, a listák, a mutatók.

Léteznek nevesített konstansok is. Ezek is lehetnek beépítettek, vagy létrehozottak is. Ilyen konstans például a π . Létrehozni pedig `c++` nyelvben a `#define`-al míg `java`-ban a `final` utasítással tudunk.

A legalapvetőbb dolgok azonban a változók. Ezekben tároljuk a számunkra szükséges dolgokat. Egy változónak van típusa és értéke. A típusa lehet szám, karakter, karakterlánc, illetve logikai. Az értéke pedig a lehetséges típusok alapján lehet szám, karakter, karakterek sorozata, illetve igaz/hamis.

A programozási nyelvekben használunk még Kifejezéseket is. Ezek egyfajta műveletek. Egy kifejezésnek három része van. A művelet bal szélén valamilyen változó áll, aminek szeretnénk egy értéket adni, középen egy műveleti jel, és jobb oldalt pedig vagy egy másik változó, vagy egy konkrét érték, ami a bal oldalt álló változónak az új értéke lesz.

A gépi kódot a fordító az utasítások alapján generálja. Ezek az utasítások a következők lehetnek : Értékadó utasítás; Üres utasítás; Ugró utasítás; Elágaztató utasítások; Ciklusszervező utasítások; Hívó utasítás; Vezérlésátadó utasítások; I/O utasítások; Egyéb utasítások.

A ciklusokat is rengetegszer használják a programozók. Ezek segítségével a megadott parancsokat egymás után többször is elvégzi a program. A ciklusokhoz tartoznak a Vezérlő utasítások, amik a következők: A

Continue parancs esetén a ciklus jelenlegi lépésében a hátra lévő utasításokat nem hajtja végre, hanem a következő cikluslépésre ugrik. A Break parancs esetén a ciklus megáll, és nem fut tovább. A Return parancs esetén leáll a ciklus és visszaadja az eredményt.

Az alprogramok, vagy másnéven függvények olyan programrészletek, amiket megírva később meg lehet hívni őket, és a megadott értékekből előállítanak egy eredményt. Az alprogramoknak van neve és vannak argumentumai. A nevével hívjuk meg őket, az argumentumok pedig azok az értékek amikből a végeredmény áll elő.

A programokban a blokkok olyan programrészletek amik programrészletekben helyezkednek el. Ilyen például az if elágazás után a potenciálisan végrehajtandó utasítások.

10.2. Kerninghan és Richie olvasónaplója

Először is A vezérlési Szerkezetek a ciklusok, az elágazások

Az elágazásokba beletartozik az if, if-else, else-if, else, és a switch feltételvizsgálatok. Ezekkel értékeket tudunk vizsgálni, és ezek végrehajtani a megfelelő utasítást, utasításokat végrehajtani. Az if, if-else, else-if, else kifejezéseknél az if után vizsgáljuk meg az értéket, majd jönnek az utasítások, és végül opcionálisan else-if vagy else. Bármennyi else-if lehet egymás után, azonban else csak egy vagy nulla. Viszont célszerű else-t is használni, mert általában kevés esély van arra, hogy minden esetet lefedünk szimplán else-if használatával.

Ezzel szemben a switch esetében megadjuk az értéket, majd tetszőleges darabszámú case használatával megnézzük, hogy az e az érték, ami nekünk kell, és ha igen akkor az aktuális case utasításait hajtja végre. Célszerű megjegyezni, switch-case használatánál a program minden esetben végigellenőrzi az összes case-t, ezért ha nem szeretnénk, hogy az összeset ellenőrizze, ha már talált egy egyezést, akkor használjuk a break utasítást.

A Ciklusok esetében beszélhetünk for, while, és do while ciklusokról.

A for esetében a programozó adja meg, hogy hányszor fusson le a ciklus. A while és a do while esetében pedig addig fut a ciklus, amíg egy feltétel nem teljesül. Éppen ezért vigyázni kell, nehogy véletlenül egy végtelen ciklus alakuljon ki. Fontos különbség még a while és a do while ciklusok között, hogy míg a while ciklus először ellenőrzi, hogy teljesült-e a feltétel, majd pedig lefuttatja az utasításokat, addig a do while ciklus először lefuttatja az utasításokat, majd pedig ellenőrzi, hogy teljesült-e már a feltétel.

A C nyelv alapvető adattípusai az int, a float, a double, a char, és a bool.

Az intek (integerek) egész számok amik lehetnek pozitívak és negatívak is. Az int mérete 4bájtt, azaz 32bit

A float és a double típusú változókban valós, úgynevezett lebegőpontos számokat lehet tárolni. Ilyen például a 0.5. A különbség a két változó között azonban az, hogy, hogy míg a float mérete csak 4bájtt, addig a double mérete 8bájtt.

A char (character) típusú változóban meglepő módon egy karaktert lehet eltárolni. A char mérete 1bájtt.

Az alapvető adattípusokon túl a C nyelvnek vannak Állandói is. Ilyen például a #define, amivel meg tudunk adni meg nem változtatható értékeket. Ezekre később hivatkozni tudunk. De ilyen állandók még az escape sorozatok, amiket az adatok kiíratásánál tudunk alkalmazni. Ilyen például a \n amivel egy új sort kezdünk.

10.3. Benedek Zoltán, Levendovszky Tihamér - Szoftverfejlesztés C++ nyelven olvasónaplója

A C++ egy objektum orientált programozási nyelv, ami egyben alacsonyabb szintű elemeket is támogat.

A C++-ban ha egy függvényt paraméterek nélkül hívunk meg, akkor az egyenértékű egy void paraméterrel. Aminek pont az a jelentése, hogy a függvénynek nincs paramétere. További különbség, hogy míg a C nyelvben egy függvényt csak a neve alapján azonosítunk, addig C++ ban egy függvényt a neve és az argumentumai határoznak meg. Ezáltal C++ ban előfordulhat két ugyan olyan nevű függvény különböző argumentumokkal. További változás, hogy C++ ba be lettek vezetve a referenciák, valamint egy új típus is bevezetésre került, ami nem más mint a bool. A bool egy logikai változó ami lehet igaz vagy hamis értékű

A C++ bevezette az osztályokat, amik az adatok, és metódusok együttese. Innen ered az objektum orientáltság, mivel az objektum a egy darab osztály egy darab előfordulása. A metódus pedig az osztálynak egy olyan eleme, egy olyan függvény, ami az osztályba tartozó adatokat manipulálja.

A konstruktorok és destruktorok előredefiniált függvénymezők, amelyek kulcsszerepet játszanak a C++ nyelvben. Alepvető probléma a programozásban az inicializálás. Mielőtt egy adatstruktúrát elkezdénénk használni, meg kell bizonyosodnunk arról, hogy megfelelő méretű tárterületet biztosítsunk a számára, és legyen kezdeti értéke. Ezt a problémát orvosolják a konstruktorok.

A destruktorok pedig egy konstruktor által már létrehozott objektum törlésében segítenek. Törlik a tartalmát, és felszabadítják az objektum által elfoglalt helyet. Ha mi nem hozunk létre destruktort, akkor a C++ a saját alapértelmezett változatát fogja használni.

Létezik még másoló konstruktor is, ami egy már meglévő objektumból hoz létre egy újat. Lefoglal a memóriában egy részletet, és annak az értékét felülírja a már létező objektum értékeivel.

A C++ nyelven az osztályok adattagjai előtt szerepelhet a static szó. Ez azt jelenti, hogy ezeket a tagokat az osztály objektumai megosztva használják.

Gyakran kerülünk olyan helyzetbe, hogy egy adott típusnak úgy kellene viselkednie, mint egy másiknak. Ekkor kell típuskonverziót alkalmazni. Ezt meg lehet tenni implicit és explicit módon is.

Implicit konverziót általában haonló típusokon lehet elvégezni. Ilyen például ha egy integer változó értékét szeretnénk átadni egy long típusú változónak.

```
int x = 5;
long y = x;
```

Mind a ketten egész szám típusok, viszont a long nagyobb méretű, ezért a konverzió gond nélkül megtörténik.

Ez a módszer explicit konverzió esetén nem biztos, hogy működni fog, és még adatvesztéssel is járhat. Ilyen például ha egy integer változó értékét szeretnénk átadni egy byte értékű változónak. A byte mérete kisebb mint az it, ezért a változó előtt kell lennie egy zárójelnek benne a típussal.

```
int x = 300;
byte y = (byte)x;
```

Itt például az y értéke 44 lesz, mert a 300-at kilenc biten kell felírni, azonban a byte csak 8 bitet tárol, ezért az x -nek csak az első 8 bitjét fogja eltárolni.

C++ ban lehetőségünk van függvénysablonok és osztálysablonok létrehozására is, ezek a templatek. A template argumentumai eltérnek a hagyományos argumentumoktól. Egyrészt már a fordítás közben kiértékelődnek, ezért a futás közben már konstansok. Éppen e miatt az argumentumok típusok is lehetnek, nem csak értékek.

10.4. Python nyelvi bevezetés

Rövid olvasónapló a [?] könyvről.

A Python nyelv egy nagyon magas szintű programozási nyelv. Ezt a nyelvet a programozás megkönnyítésére hozták létre, ennek ellenére nagyon sokrétű a felhasználhatósága. Használják a gépi tanuláshoz, adatbázisok kezeléséhez. Híres programok amik rá épülnek: Blender, The Sims 4, World Of Tanks.

III. rész

Második felvonás

DRAFT

**Bátf41 Haxor Stream**

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

DRAFT

11. fejezet

Helló, Berners-Lee!

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

Az objektumorientált paradigma alapfoglamai. Osztály, objektum, példányosítás

Először értsük meg, hogy mi is az objektum, és hogy keletkezik, és mi játszik fontos szerepet a működésében. Az objektum a java programozási nyelv alapvető eleme, éppen ezért a Java egy objektum orientált programozási nyelv. Az objektum a valódi világ egy elemének a rá jellemző tulajdonságai és viselkedései által modellezett eleme. Az objektumokkal általában valamilyen feladatot szeretnénk megoldani. Egy objektum tulajdonságokból(változók), és viselkedésekből(metódusok) áll. A változókkal írhatjuk le az adott objektum állapotát, minden egyednek saját készlete van a változókból, éppen ezért ezeket példányváltozóknak nevezzük. A metódus nagyrészt hasonlít egy függvényre. Azaz utasításokat hajt végre, kaphat paramétereket, és egy értékkel tér vissza. Az osztályok pedig az azonos típusú objektumok modelljét írják le. A program a működése során példányosítja az osztályokat, azaz konkrét objektumokat hoz létre, vagyis amikor egy objektumot létrehozunk, azt valójában egy osztályból hozzuk létre. Amikor egy új egyedet szeretnénk létrehozni, akkor azt a konstruktor fogja felépíteni. Az előzőekben említettem, hogy a változókat a metódusok kezelik. Azonban alaphelyzetben ez nem igaz. Ha csak úgy megírunk egy osztályt, akkor annak a változóhoz kívülről is hozzá lehet férni, a metódusok figyelembe vétele nélkül. Ez pedig nem jó dolog. Ahhoz, hogy egy objektum biztonságos legyen, priváttá kell tennünk a változóit, ezáltal csak az adott objektum férhet hozzá a saját változóhoz közvetlenül. Minden más csak a metódusain keresztül férhet hozzájuk. Az, hogy a változók és metódusok egy helyen vannak tárolva az osztályokban (egységbe záras), valamint az, hogy egy objektum változóhoz csak ellenőrzött körülmények között lehet hozzáférni (egységbe záras) együttesen az adatabsztrakciót, azaz az objektumorientált paradigma egyik alapját alkotják.

C++-ban az osztályok példányokat tárolnak, például különböző bankszámlák. Ezen kívül megtalálhatóak a példányok tulajdonságai is, mintpéldául a számlán lévő egyenleg, illetve a számlán végzett műveletek (pénz betétele, felvétele). Az ilyen egyedeket nevezzük objektumoknak. Fontos, hogy egy objektum tulajdonságaihoz csak a műveletein keresztül lehessen hozzáférni. Ha például egy bankszámla egyenlegét csak úgy át lehetne írni, az nem lenne jó hatással a társadalomra. Éppen ezért az objektumok tulajdonságait és műveleteit egységbe kell zárni, illetve biztosítani kell, hogy az objektum tulajdonságaihoz a program többi része ne tudjon hozzáférni. Ezt hívjuk adatrejtésnek. Ha van egy madár, illetve egy papagáj osztályunk, akkor a két osztály között egyfajta kapcsolat van, mivel a papagáj maga is egy madár. Ezt nevezzük specializációnak. A

madár általánosabb fogalom mint a papagáj. Éppen ezért a speciálisabb osztály rendelkezik az általánosabb osztály tulajdonságaival és műveleteivel, másszóval öröklí ők. Szóval egy papagájt bármikor kezelhetünk madárként. Ez a három fogalom (adatretjtés, specializáció, öröklődés) alkotja az objektumorientált programozás alapelveit. Azonban a C++-ban (sok OOP, köztük a JAVA nyelvvel ellentétben) megtalálható a típusátmogatós is. Ez azt jelenti, hogy az osztályok ugyan úgy működhetnek, mint a beépített típusok.

Öröklődés, osztályhierarchia. Polimorfizmus, metódustúlterhelés. Hatáskörkezelés. A bezárási eszközrendszer, láthatósági szintek. Absztrakt osztályok és interfészek.

A legegyszerűbb példa az öröklődésre az az, amikor egy osztály egy már meglévő osztály kiterjesztéseként definiálunk. Ez lehet új műveletek, vagy új változók bevezetése is, maga az osztály pedig lehet public, illetve nem public is. Az eredeti osztályt szülőosztálynak, a kiterjesztettét pedig gyermek osztálynak nevezzük. A gyermekosztály megöröklí a szülőosztály változóit és metódusait, ha a láthatósági szintje az adott változónak/metódusnak lehetővé teszi azt. A láthatósági szint lehet public, ami azt jelenti, hogy az adott változót vagy metódust nem csak a gyermek osztályok, hanem bármely másik osztály objektumai is elérí. A láthatósági szint lehet protected is. Ebben az esetben már csak az adott osztályból kiterjesztett gyermekosztályok érí el ők. A harmadik lehetőség pedig a private, amikor pedig csak a szülőosztály objektumai tudják elérni az adott változót/metódust. Azonban a gyermek osztály nincs csupán ezekre korlátozva, vagyis a gyermekosztályoknak lehetnek saját változói, és metódusai, illetve fel is tudják írni a szülőtől öröklött metódusokat.

Mivel a gyermek osztály a szülő osztály minden változójával és metódusával rendelkezik, ezért használhatóak minden olyan esetben, amikor a szülő használható. Egy változó pedig nem csak a deklarált típusú, hanem egy leszármazott objektumra is hivatkozhat. Ezt polimorfizmusnak, azaz többalakúságnak nevezzük.

Ha egy kiterjesztett osztálybeli metódusnak ugyan az a szignatúrája, és visszatérési értéke, mint a szülőosztály metódusának, akkor a leszármazott osztály felülírja a szülőosztály metódusát. Ez lehetővé teszi, hogy egy osztály örökljön egy olyan szülőosztálytól, aminek hasonló a viselkedése, majd szükség esetén ezen változtasson. A felülíró metódus neve, paramétereinek a száma és típusa megegyezik a felülírt metódussal.

Alapértelmezetten egy újonnan létrehozott osztálynak az Object nevű osztály lesz az őse. Ez áll a Java osztályhierarchia csúcsán. Ebből kiindulva lehet ábrázolni az osztályok hierarchiáját egy fa adatszerkezetben.

Lehetőség van Absztrakt osztályokat is létrehozni az *abstract* módosítóval. Az ilyen osztályok tartalmazhatnak absztrakt, azaz törzs nélküli metódusokat, amiket szintén az *abstract* módosítóval kell jelölni. Az ilyen osztályok nem példányosíthatóak, mivel a példányokra nem lenne értelmezve minden metódus. Ennek ellenére van értelme absztrakt típusú változókat és paramétereket deklarálni, mivel az ilyen változók az adott absztrakt osztály bármely leszármazottjának példányára hivatkozhatnak.

C++ ban az öröklés során egy osztály specializált változatait hozzuk létre, amelyek öröklí a szülőosztály jellemzőit és viselkedését. Ezeket az osztályokat alosztályoknak nevezzük. Az alosztályok megváltoztathatják az öröklött tulajdonságokat, és új metódusokat is adhatunk hozzá (a Java nyelvhez hasonlóan). Az öröklődés fajtája lehet egyszeres öröklés, és többszörös öröklés is (az utóbbi a Java nyelvben csak az Absztrakt osztályok használatánál lehetséges). Az egyszeres öröklés esetén minden származtatott osztály pontosan egy közvetlen szülőosztály tagjait öröklí, míg a többszörös öröklődés során a származtatott osztály több közvetlen szülőosztály tagjait öröklí. Például létrehozhatunk egymástól független autó és hajó osztályokat, majd pedig ezekből örökléssel definiálhatunk egy kételtű osztályt, ami egyaránt rendelkezik az autó és a hajó jellemzőivel és viselkedésével is. Ebben különbözik a C++ nagyon sok magasszintű programozási nyelvtől (Java, C# ...), mivel azok csak az egyszeres öröklést támogatják.

Modellező eszközök és nyelvek. AZ UML és az UML osztálydiagramja.

Az UML, azaz Unified Modeling Language, vagy magyarul egységesített modellezőnyelv segítségével fejlesztési modelleket lehet szemléltetni. Egy integrált diagramkészletből áll, amelyet a szoftverfejlesztők számára fejlesztettek ki a programok megjelenítésére, felépítésére és dokumentálására. Az objektumorientált szoftverfejlesztési folyamat nagyon fontos részre. Többnyire grafikus jelöléseket használ a projektek tervezésére. Rengeteg diagram, azaz modell van hozzá. Az UML használható bármelyik ma ismert programozási nyelvvel, mivel azoktól független absztrakciós szinten fogalmazza meg a rendszer modelljét. Maga az UML egy grafikus modellező nyelv, azaz a diagramok téglalapokból, vonalakból, ikonokból, és szövegből állnak.

A Class diagram egy központi modellezési technika, amely szinte minden objektum-orientált módszert átfut. A rendszerben található osztályokat, interfészeket, egyéb típusokat, és a közöttük lévő kapcsolatokat írja le. Rendkívül jól lehet osztálydiagrammal megmutatni egy program felépítését, valamint az osztályok között fennálló asszociációt, aggregációt, és kompozíciót. Ha egy modellben két osztálynak kommunikálnia kell egymással, akkor szükségünk van egy kapcsolatra a két osztály között. Ezt a kapcsolatot reprezentálja az asszociáció. Az asszociációt egy a két osztály között lévő vonal, valamint az azon lévő irányt mutató nyíl(ak) jelöli(k). Ha a vonal mindkét oldalán van nyíl, akkor az asszociáció kétirányú. Az aggregáció és kompozíció az asszociáció részalmazai, vagyis az asszociáció különleges esetei. Mind a két esetben egy osztály objektuma "birtokol" egy másik osztály másik objektumát, de van a kettő között egy kis különbség. Az aggregáció egy olyan kapcsolatot jelent, amiben a gyerek a szülőtől függetlenül létezhet. Például ha van tanóra, ami a szülőosztály, és tanuló, ami a gyerekosztály. Ha töröljük a tanórát, attól a tanulók még léteznek. Ezzel szemben a kompozíció esetében egy olyan kapcsolatról van szó, amiben a gyerek nem létezhet a szülő nélkül. Például ha van egy ház szülőosztályunk, és egy szoba gyerekosztályunk. A szoba nem létezhet a ház nélkül. Az aggregációt és a kompozíciót is vonal+rombusz kombinációval lehet jelölni, azonban az aggregációnál a rombusz üres, a kompozíciónál pedig nem.

Objektumorientált programozási nyelvek programnyelvi elemei: karakterkészlet, lexikális egységek, kifejezések, utasítások.

Java-ban a charset egy osztályt jelöl, ami tizenhat bites Unicode kód egységek és a bájt szekvenciák megnevezett leképezése. Ez az osztály meghatározza a dekóderek és kódolók létrehozásának, valamint a karakterkészlethez társított különféle nevek lekérésének módszereit. Ez az osztály immutable, azaz állandó. Ennek az osztálynak vannak statikus metódusai, amiknek a segítségével le lehet tesztelni, hogy egy adott karakterkészlet támogatott-e. Ezen kívül lehet ezeket a metódusokat karakterkészletek név szerinti keresésére is használni. A Java platform minden implementációjának támogatnia kell a következő karakterkészleteket: US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, UTF-16. A JVM minden példányának van egy alapértelmezett karakterkészlete, ami a JVM bootolásakor kerül meghatározásra, és általában az operációs rendszer karakterkészletétől függ.

A kifejezés egy olyan változókból, operátorokból, és metódushívásokból álló programrészlet, amelynek egyetlen érték lesz az eredménye. A kifejezés által visszaadott érték típusa a kifejezésben használt elemektől függ. Például az `int a = 0;` kifejezés `int` értékkel tér vissza. De egy kifejezés más típusú értékkel is visszatérhet, például logikai, vagy string. A Java programozási nyelv lehetővé teszi összetett kifejezések létrehozását különféle kisebb kifejezésekből, feltéve, hogy a kifejezés egyik részének a típusa megegyezik a kifejezés másik részének a típusával. De ilyen esetben figyelni kell a számolási sorrendre is intek esetében. Ha például csak szorzás szerepel az összetett kifejezésben, akkor teljesen mindegy a sorrend, de ha van összeadás és szorzás is, akkor előbb a szorzás lesz elvégezve. Ha azt szeretnénk, hogy előbb az összeadás hajtsódjon végre, akkor zárójeleket kell használni. Az utasítások nagyjából megegyeznek a természetes nyelvek mondataival. Egy utasítás egy teljes végrehajtási egységet képez.

Objektumorientált programozási nyelvek típusrendszere, Típusok tagjai: mezők, (nevesített) konstansok,

tulajdonságok, metódusok, események, operátorok, indexelők, konstruktorok, destruktorok, beágyazott típusok.

Java nyelvben négy fontos típus létezik. Ezek a primitív típusok, amelyek a primitív értékeknek megfelelő típusok, például int, short, long, byte, char, float, double, boolean. Null típusok, tömb típusok, valamint az osztály típusok.

A primitív típusok olyan primitív értékeket tárolnak, mint az egész számok, lebegőpontos számok, karakterek és logikai értékek. A négy egész szám típus abban különbözik egymástól, hogy milyen nagy az a szám, amit maximum el tudnak tárolni. Hasonlóképpen a lebegőpontos számok két típusa, között is az a különbség, hogy milyen nagy számot tudnak eltárolni. Egy kisebb típusú változó értékét gond nélkül bele lehet helyezni egy nagyobb típusú változóba, azonban ha egy nagyobb típusú szám értékét szeretnénk behelyezni egy kisebb típusú változóba, akkor át kell alakítanunk azt. A tömb olyan adatstruktúra, amely állandó hozzáférést tesz lehetővé a tömb elemeihez. A tömbök rögzített méretűek. Az osztály típusok pedig az objektumok típusai.

Java-ban egy mező egy osztályon belüli változó, amit a következő szintaxissal lehet létrehozni: *[elérés] [static] [final] típus név [= kezdőérték];*. A szögletes zárójel azt jelenti, hogy az adott rész opcionális. Először is egy mezőhöz deklarálhatunk hozzáférési módosítót. A hozzáférés-módosító meghatározza, hogy mely osztályok objektumai férhetnek hozzá a mezőhöz. Másodszor pedig, a mező típusát kell megadni. Ez lehet pl int, string, boolean stb. Harmadszor, a Java mező statikusnak nyilvánítható. A Java-ban a statikus mezők az osztályhoz tartoznak, nem pedig az osztály példányai. Így bármely osztály minden objektuma ugyanazt a statikus mezőváltozót fogja elérni. A nem statikus mező értéke az osztály minden objektumánál eltérő lehet. Negyedszer, a Java mező konstansnak is nyilvánítható. A konstans mező értékét nem lehet megváltoztatni. Ötödször, a Java mezőnek nevet kell adni. Végül pedig adhatunk a mezőnek kezdőértéket. A nevesített konstans egy azonosító, amely állandó értéket képvisel. A változó értéke a program végrehajtása során változhat, de a nevesített konstansok állandó adatokat képviselnek, amelyek soha nem változnak. Ilyen például pi. Két féle állandó létezik java-ban, final és static final. Ha csak simán final, akkor az értéknek csak az adott osztálypéldányban kell állandónak lennie, ha viszont static final, akkor az értéknek minden osztálypéldányban ugyan annak kell lennie.

A Properties osztály tulajdonságok halmazát reprezentálja. Ezek a tulajdonságok elmenthetők streambe, vagy pedig betölthetők egy streamből. A Properties a Hashtable osztály alosztálya. Egy értéklista fenntartására szolgál, amelyben a kulcs és az érték is egy string. A Properties osztály egyik hasznos képessége, hogy meghatározhat egy alapértelmezett tulajdonságot, amely visszaadódik, ha egy adott kulcshoz nincs érték társítva. A metódusok olyan utasítások összessége, amelyek egy konkrét feladatot hajtanak végre, és az eredményt visszaadják annak, ami meghívta a metódust. De az is lehet, hogy csak végrehajtanak egy feladatot, és nem térnek vissza semmivel. A metódusok lehetővé teszik a kód újra felhasználását a kód újraírása nélkül. Java-ban, a C++-al ellentétben minden metódusnak egy osztály részének kell lennie. A konstruktor objektumokat inicializál, amikor azok létrejönnek. Ugyan az a neve, mint az osztálynak, a szintaktikája pedig hasonló egy metódushoz. Jellemzően konstruktorokkal adjuk meg az objektumok kezdőértékeit. Java-ban minden osztály rendelkezik konstruktorral, attól függetlenül, hogy a programozó definiált-e egyet vagy nem, mert ha nem, akkor a Java automatikusan megad egy alapértelmezett konstruktort, amely az összes tagváltozó értékét null-ra inicializálja.

Interfészek. Kollektívok. és Funkcionális nyelvi elemek. Lambda kifejezések.

Pont úgy mint az osztályoknak, az interfészeknek lehetnek metódusai és változói. De az interfészben deklarált metódusok alapértelmezés szerint absztraktak. Az interfészek meghatározzák, hogy az osztálynak mit kell tennie, és nem pedig azt hogy hogyan. Az interfész meghatározza azokat a metódusokat, amelyeket az osztálynak végre kell hajtania. Interface deklarálásához bármilyen meglepő, az interface kulcsszót

kell használni. Ez a teljes absztrakció biztosítására szolgál. Ez azt jelenti, hogy az interfészben az összes metódus üres testtel kerül deklarálásra, hozzáférhetőségük `public`, és alapértelmezés szerint minden mező publikus, statikus és `final`. Az interfészt megvalósító osztálynak végre kell hajtania az interfészben deklarált összes metódust. A `Collection` az egyedi objektumok csoportja, amely egyetlen egységként van ábrázolva. A Java biztosít egy `Collection Framework`-öt, amely meghatároz több osztályt és interfészt azért, hogy egy objektumcsoportot egyetlen egységként képviseljen. A `Collection` interfész és a `Map` interfész a két gyökér interfésze a Java `Collection` osztályának. A Lambda kifejezések, amelyek Java 8-tól érhetőek el, alapvetően a funkcionális interfészek példáit fejezik ki (egy absztrakt módszerrel rendelkező interfészt funkcionális interfésznek hívunk. Példa erre a `java.lang.Runnable`). A lambda kifejezések megvalósítják az egyetlen elvont funkciót, éppen ezért megvalósítják a funkcionális interfészeket. A lambda kifejezések lehetővé teszik a funkcionalitás metódus argumentumként kezelését, vagy a kód adatként kezelését. A lambda kifejezést úgy lehet átadni, mintha objektum lenne, és igény szerint végrehajtható. A lambda kifejezés teste tartalmazhat nulla, egy vagy több állítást is. Ha egynél több állítás van, akkor kapcsos zárójelbe kell őket zárni.

Adatfolyamok kezelése, streamek és állománykezelés. Szerializáció.

A Java 8-ban bevezetett `Stream API` az objektumgyűjtemények feldolgozására szolgál. A stream nem adatszerkezet, hanem egy `Collection`-ból, tömbből vagy I/O csatornából származó adatot veszi be. A streamek nem változtatják meg az eredeti adatszerkezetet, csak az eredményt mondják meg a megvalósított metódusok alapján. Támogat különféle műveleteket az ilyen elemekre vonatkozó számítások elvégzéséhez. A stream műveletek közbenső vagy terminális műveletek. A közbenső műveletek stream-mel térnek vissza, ezért több is követheti egymást pontosvessző nélkül. A terminális műveletek vagy voidok, vagy nem stream adatot eredményeznek. A `java.io` csomag szinte minden olyan osztályt tartalmaz, amelyre esetleg szükség lehet a Java bemenet és kimenet (I/O) megvalósításához. Kétféle stream létezik, az `InputStream`, ami egy forrásból származó adatok olvasására szolgál, és az `OutputStream`, ami pedig az adatot egy célhelyre írja. A Java byte streameket 8 bit méretű bájtok írására és olvasására lehet használni. Sok ilyen van, de a leggyakrabban használtak a `FileInputStream` és `FileOutputStream`. Az össze programozási nyelv támogatja azt a szabványos i/o-t ahol a felhasználó a konzolon keresztül tud adatot megkapni és megadni. Ilyen Java-ban a `Standard Input`, ami olvassa a felhasználó által megadott adatot, a `Standard Output`, ami a program által előállított adatot adja át a felhasználónak, és a `Standard Error`, ami a program által előállított hibák adatait közli a felhasználónak. A Java biztosít egy olyan mechanizmust, amelyet objektum-szerializációnak nevezünk, ahol egy objektum egy olyan bájt sorozatként ábrázolható, amely tartalmazza az objektum adatait, valamint az objektum típusára és az objektumban tárolt adatok típusára vonatkozó információkat.

Kivételkezelés és Reflexió. A fordítást és a kódgenerálást támogató nyelvi elemek.

A Kivétel egy nem kívánt vagy váratlan esemény, amely egy program végrehajtásakor fordul elő, azaz futási időben, és megzavarja a program utasításainak normál folyamatát. Egy hiba olyan komoly problémát jelent, amelyet egy egyszerű alkalmazásnak nem szabad catchelnie. A kivétel ezzel szemben olyan feltételeket jelent, amelyeket egy egyszerű alkalmazás megpróbálhat catchelni. Az összes kivétel és hiba típus a `Throwable` osztály alosztálya, amely a hierarchia alaposztálya. Az egyik ág az `Exception` a feje. Ez az osztály kivételes körülmények között használatos, amelyeket a felhasználói programoknak el kellene kapniuk, például `NullPointerException`. Másik ág, azaz a Hiba, JVM használatával jelzi azokat a hibákat, amelyek a JRE-vel kapcsolatosak, pl: `StackOverflowError`. A Java kivételkezelés öt kulcsszón keresztül történik: `try`, `catch`, `throw`, `throws`, és `finally`. Azok a kódrészletek amelyek exception-t okozhatnak, egy `try` blokkba kerülnek. Ha exception történik a `try` blokkban, akkor azt eldobja (`throw`), amit a `catch` blokk elkap, és kezel. Az a kód pedig, amelyet mindenképp végre kell hajtani exception esetén is, a `finally`-be kerül.

A reflexió egy olyan API, amelyet a metódusok, osztályok, interfészek viselkedésének megvizsgálására

vagy módosítására használnak futási időben. A reflexióhoz szükséges osztályokat a `java.lang.reflect` csomag tartalmazza. Reflexión keresztül metódusokat lehet futtatni, függetlenül a metódusok hozzáférési szintjétől. A reflexió segítségével információkat szerezhetünk az Osztályról, a `getClass()` metódussal, ez a metódus annak az osztálynak a nevét adja vissza, amelyhez az adott objektum tartozik. Információt szerezhetünk a konstruktorról, a `getConstructors()` metódussal. Ezzel annak az osztálynak a nyilvános konstruktorait kapjuk meg, amelyhez az adott objektum tartozik. Valamint lekérhetjük még a metódusokat is, a `getMethods()` függvénnyel, amely annak az osztálynak a publikus metódusait adja vissza, amelyhez az adott objektum tartozik. Az annotációk kiegészítő információkkal szolgálnak a programról. Az annotációk `@` karakterrel kezdődnek, és nem változtatják meg a program működését, de megváltoztathatják azt, hogy hogyan kezeli a fordító a programot. Három féle annotáció létezik. Az első a Jelölő Annotációk, amelyek deklarációt jelölnek, a második az egyértékű Annotációk, a harmadik pedig a Teljes Annotációk.

Multiparadigmás nyelvek és Programozás multiparadigmás nyelveken.

A paradigmát úgy is nevezhetnénk, mint egy módszer valamilyen probléma megoldására vagy valamilyen feladat elvégzésére. A programozási paradigma pedig a probléma megoldásának megközelítése valamilyen programozási nyelv használatával. Sok féle paradigma létezik a különböző igények kielégítésére. Én most az Imperatív programozási paradigmáról fogok írni, amely az egyik legrégebbi programozási paradigma. Szoros kapcsolatban áll a gép architektúrájával. Úgy működik, hogy a program állapotát hozzárendelési utasításokkal változtatja meg. Az állapot megváltoztatásával lépésről lépésre hajtja végre a feladatot. A fő hangsúly a cél elérésének módja. Az előnyei, hogy nagyon egyszerű implementálni, valamint ciklusokat, változókat, stb. tartalmaz. Hátránya, hogy összetett problémákat nem lehet vele megoldani, valamint a párhuzamos programozás nem lehetséges vele. Az imperatív programozást három kategóriába lehet sorolni: eljárási, OOP és párhuzamos. Ezek a következők: A Procedurális programozási paradigma az eljárásokat hangsúlyozza. Az Objektumorientált programozás alapján készült program a kommunikációra szánt osztályok és objektumok gyűjteményeként van írva. A legkisebb és alapvető entitás az objektum, és mindenféle számítást csak az objektumokon végeznek. Nagyobb hangsúly van az adatokon az eljárás helyett. Szinte minden valós problémát képes kezelni, amelyek manapság forgatókönyvbe kerülnek. A Párhuzamos feldolgozási megközelítés jelentése a program utasításainak feldolgozása több processzor közötti elosztással. A párhuzamos feldolgozási rendszer számos processzort tartalmaz, azzal a céllal, hogy a feladatokat rövidebb idő alatt végezze el azok megosztásával.

11.2. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobil-programozásba

A Python programozási nyelvet Guido van Rossum alkotta meg 1990-ben. Maga a python egy magas szintű, dinamikus, objektumorientált, és platformfüggetlen programozási nyelv. Leginkább egyszerű alkalmazások készítésére használatos. Viszonylag könnyen meg lehet tanulni a használatát, ezért hamar el lehet vele érni látványos eredményeket. Különlegessége más nyelvekkel szemben (pl. C, C++, Java), hogy nincs szükség a programkód fordítására. Elegendő egy forrás fájlt megírni, és az automatikusan fut is. A python programok általában sokkal rövidebbek, mint ugyanazon programok C++ vagy Java nyelven. Ennek több oka is van. Egyrészt az adattípusai lehetővé teszik, hogy összetett kifejezéseket írjunk le rövid állításokban. Másrészt nincs szükség a változók definiálására. És végül, a legkedveltebb ok, hogy a python nyelv nem használ se zárójeleket, se pontosvesszőket. Ezek helyett a kód csoportosítása új sorral és tabulátorral történik. Pythonban egy programblokk végét egy kisebb behúzású sor jelzi, az utasítások pedig a sorok végéig tartanak. Éppen ezért nincs szükség pontosvesszőre. Ha viszont egy utasítás nem fér el egy sorban, akkor az adott sor végére egy `\` jelet kell tenni, a megjegyzéseket pedig kettőskereszt jellel tujuk jelezni.

A python nyelvben a változók az objektumokra mutató referenciák. Egy változó hozzárendelését a del kulcsszóval tudjuk törölni, ha pedig egy objektumra már egy változó se mutat, akkor a garbage collector fogja törölni az adott objektumot. Érdekeség ezzel kapcsolatban, hogy a változóknak nem kell konkrét típust adnunk, mivel kitalálja, hogy mire gondolunk. Az adattípusok a következők lehetnek: számok, sztringek, ennesek, listák, és szótárak. A számok lehetnek egészek, komplexek, és lebegőpontosak is, a sztringeket pedig idézőjelek, illetve aposztrófok közé írva lehet megadni.

Maguk a változók lehetnek globálisak vagy lokálisak. Alapvetően a lokális az alapértelmezett, ezért ha azt szeretnénk, hogy egy változó globális legyen, akkor azt a változót a függvény elején kell felvenni, illetve elérni a global kulcsszót. A különböző típusok közötti konverziók támogatottak, ha van értelmük. Például int, long, float, illetve complex típusok közötti konverzió. De sztringekből is képezhetünk számot. Ehhez csak a használt számrendszert kell megadni, pl: int. Ezeknek a változóknak a kiíratását a print függvénnyel lehet megoldani. Ha több változó értékét szeretnénk kiíratni, akkor vesszővel kell elválasztani őket egymástól. Ezekon kívül a python nyelvben ugyanúgy elérhetőek az elágazások, illetve a ciklusok is, mint más magasszintű programozási nyelvekben. A for, illetve a while ciklus is elérhető, azokon pedig a break, illetve a continue utasítások is használhatóak. Léteznek címkék, amiket a label kulcsszóval kell elhelyezni a kódban, majd pedig a kód más részeiről a goto utasítás használatával a labelhez ugorhatunk.

Python nyelven a függvényeket a def kulcsszóval lehet definiálni. A függvényekre úgy is lehet tekinteni, mint értékekre, mivel továbbadhatóak más függvényeknek, és objektumkonstruktoroknak is. Ettől függetlenül a függvényeknek vannak paraméterei, amelyeknek adhatunk alapértelmezett értéket is. A legtöbb paraméter érték szerint adódik át, ezalól kivételek a mutable típusok, amelyeknek a függvényben történő megvalósítása hatással van az eredeti objektumra is. A függvény hívásánál a paraméterek úgy követik egymást, mint a függvény definíciójában. Emellett van lehetőség közvetlenül az egyes konkrét argumentumoknak értéket adni a függvény hívásakor, ha a zárójelben elé írjuk a változó nevét és egy egyenlőségjelet. A függvényeknek egy visszatérési értékük van.

A Python nyelvben -más nyelvekhez hasonlóan- létrehozhatunk osztályokat, és ezekből példányosíthatunk objektumokat. Az osztályok tartalmazhatnak metódusokat, amiket akár örökölhetnek is más osztályokból. Az osztály metódusait ugyanúgy lehet definiálni, mint a globális függvényeket, azonban az első paraméterük a self kell hogy legyen, amelynek az értéke mindig az az objektumpéldány lesz, amelyen a metódust meghívták. Ezen kívül az osztályoknak lehet egy speciális, konstruktor tulajdonságú metódusa, az __init__.

Léteznek különböző modulok, amelyeknek a célja a fejlesztés megkönnyebbítése. Ilyen például az appuifw, ami a felhasználói felület kialakítását, kezelését segíti. A messaging modul az SMS és MMs üzenetek kezelését segíti. A sysinfo a mobilkészülékekkel kapcsolatos információk lekérdezésére használható. A camera modullal lehet elvégezni minden, a készülék kamerájával kapcsolatos műveletet. Az audio modul pedig a hangfelvételek készítéséért és lejátszásáért felelős.

Más nyelvekhez hasonlóan a Python nyelvben is van lehetőség a kivételkezelésre a try, except és opcionálisan egy else utasítással. A try kulcsszó után szerepel az a kódblokk, amelyben a kivétel előállhat. Ha bekövetkezik a hiba, akkor az except részre ugrik a program, és az ott lévő utasításokat hajtja végre.

12. fejezet

Helló, Arroway!

12.1. OO szemlélet

A módosított polártranszformációs normális generátor beprogramozása Java nyelven. Mutassunk rá, hogy a mi természetes saját megoldásunk (az algoritmus egyszerre két normálist állít elő, kell egy példánytag, amely a nem visszaadottat tárolja és egy logikai tag, hogy van-e tárolt vagy futtatni kell az algoritmust) és az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua.! C++ ban

Megoldás forrása: [Forrás](#)

Ebben a feladatban a prog1-en már tárgyalt Polárgenerátor megírása volt a feladat. Ez az objektumorientált paradigma alapjaihoz, az osztály, objektum, és példányosítás alapfogalmakhoz tökéletes példaként szolgálhat. Ennek a programnak az a lényege, hogy először generál két értéket. Az egyik értéket eltárolja, a másikat pedig visszaadja. Majd amikor következőnek ismét generálna, akkor először megnézi, hogy van-e már tárolt érték. Ha van akkor azt a tárolt értéket adja vissza, ha viszont nincs, akkor generál két értéket, amiből az egyiket eltárolja, a másikat pedig visszaadja. Azt, hogy van-e tárolt érték, egy boolean változóban tartja nyilván.

A program a PolárGenerátor osztállyal kezdődik:

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

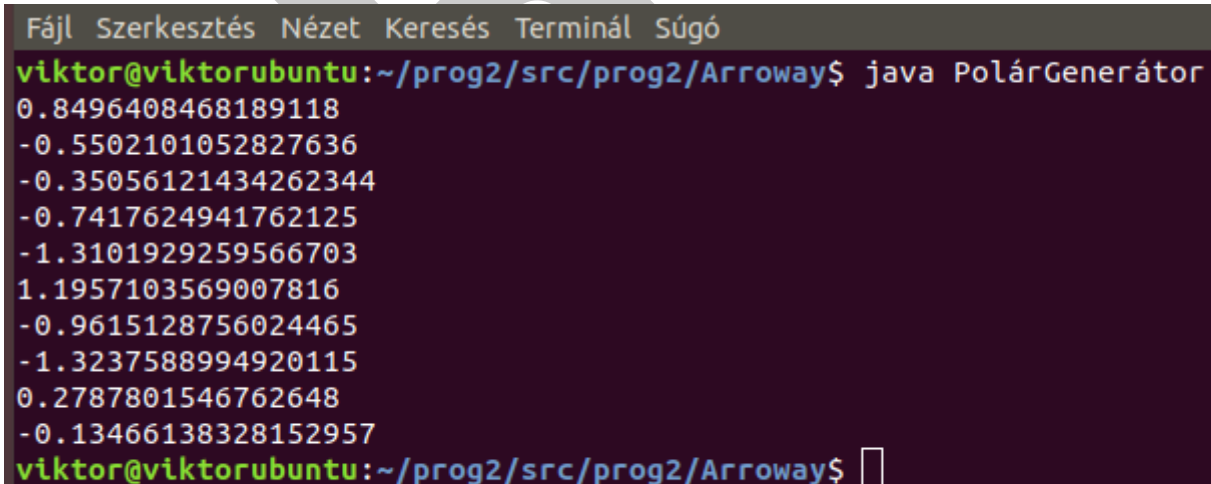
Itt kezdőértéknek meg van adva, hogy nincs tárolt érték, valamint egy double változó a majdani tárolt értéknek. Ezek után következik a következő nevű függvény, ami az érdemi munkát végzi. Ha már van tárolt értékünk, akkor a nincsTárolt változó értékét az ellenkezőjére változtatja, és visszaadja a tárolt értéket. Ha viszont nincs, akkor egy do while ciklusban először is az u1 és u2 változókhoz két véletlenszerű értéket rendel, majd pedig a Vx változók értékét úgy határozza meg, hogy Ux-et megszorozza kettővel, és a kapott eredményből kivon 1-et. Ezek után a w változónak az értéke v1 négyzetének, és v2 négyzetének az összege. Ez a ciklus addig fog futni, amíg w értéke nagyobb mint 1. Ha véget ért a ciklus akkor az újonnan deklarált r változó kezdőértékét úgy határozza meg, hogy -2-vel megszorozza w logaritmusát, majd azt elosztja w-vel, és a kapott értéknek a négyzetgyöke lesz az eredmény. Ezek után a nincsTárolt változó

értékét az ellenkezőjére állítja. Az eltárolt érték a r és $v2$ szorzata lesz, a visszaadott érték pedig r és $v1$ szorzata.

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

Végül pedig a main, ami létrehoz egy PolárGenerátor objektumot, és egy for ciklussal 10 alkalommal futtatja a függvényt, és az eredmény.

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



```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Arroway$ java PolárGenerátor
0.8496408468189118
-0.5502101052827636
-0.35056121434262344
-0.7417624941762125
-1.3101929259566703
1.1957103569007816
-0.9615128756024465
-1.3237588994920115
0.2787801546762648
-0.13466138328152957
viktor@viktorubuntu:~/prog2/src/prog2/Arroway$
```

12.2. 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!

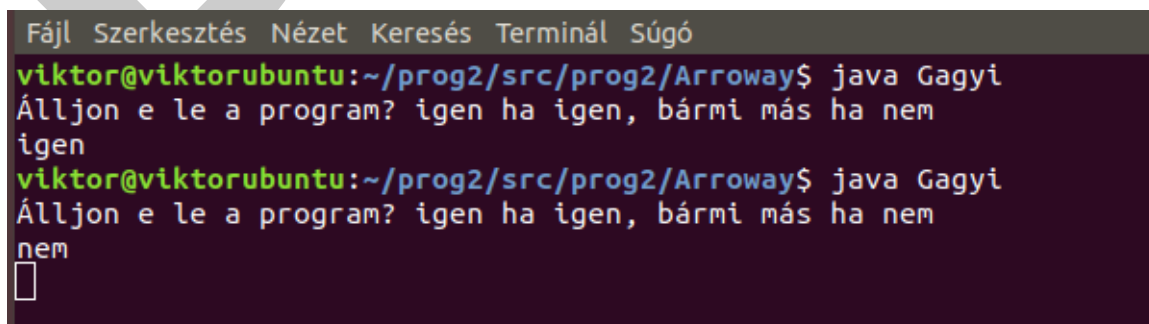
Megoldás videó:

Megoldás forrása:[Forrás](#)

Ebben a Feladatban létrehoztunk két Int objektumot 'x' és 't' néven. Az egyik alkalommal mindkét Int értékét 127-re, a másik alkalommal pedig 128-ra állítjuk be. Ezután következik egy while ciklus ami addig fut, amíg x kisebb vagy egyenlő t-vel és x nagyobb vagy egyenlő t-vel és t nem egyenlő x-el.

```
import java.util.Scanner;
public class Gagyi{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        System.out.println("Álljon e le a program? igen ha igen, bármi más ha ←
            nem");
        Integer t;
        Integer x;
        if(sc.nextLine().equals("igen")){
            x = 127;
            t = 127;
        }
        else{
            t = 128;
            x = 128;
        }
        while (x <= t && x >= t && t != x);
    }
}
```

Az érdekesség ebben a feladatban az, hogy amikor x és t értéke 127, akkor leáll a ciklus, míg amikor 128, akkor pedig egy végtelen ciklust kapunk:



```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Arroway$ java Gagyi
Álljon e le a program? igen ha igen, bármi más ha nem
igen
viktor@viktorubuntu:~/prog2/src/prog2/Arroway$ java Gagyi
Álljon e le a program? igen ha igen, bármi más ha nem
nem
█
```

És hogy ez miért történik? Erre a választ a JDK Integer.java forrásában kaphatunk.

```
778 */
779
780 private static class IntegerCache {
781     static final int low = -128;
782     static final int high;
783     static final Integer cache[];
784
785     static {
786         // high value may be configured by property
787         int h = 127;
788         String integerCacheHighPropValue =
789             sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
790         if (integerCacheHighPropValue != null) {
791             try {
792                 int i = parseInt(integerCacheHighPropValue);
793                 i = Math.max(i, 127);
794                 // Maximum array size is Integer.MAX_VALUE
795                 h = Math.min(i, Integer.MAX_VALUE - (-low) - 1);
796             } catch (NumberFormatException nfe) {
797                 // If the property cannot be parsed into an int, ignore it.
798             }
799         }
800         high = h;
801
802         cache = new Integer[(high - low) + 1];
803         int j = low;
804         for(int k = 0; k < cache.length; k++)
805             cache[k] = new Integer(j++);
806
807         // range [-128, 127] must be interned (JLS7 5.1.7)
808         assert IntegerCache.high >= 127;
809     }
810
811     private IntegerCache() {}
812 }
813
814 /**
815  * Returns an {@code Integer} instance representing the specified
816  * {@code int} value. If a new {@code Integer} instance is not
817  * required, this method should generally be used in preference to
818  * the constructor {@code Integer(int)}, as this method is likely
819  * to yield significantly better space and time performance by
820  * caching frequently requested values.
821  *
822  * This method will always cache values in the range -128 to 127,
823  * inclusive, and may cache other values outside of this range.
824  *
825  * @param i an {@code int} value.
826  * @return an {@code Integer} instance representing {@code i}.
827  * @since 1.5
828  */
829 public static Integer valueOf(int i) {
830     if (i >= IntegerCache.low && i <= IntegerCache.high)
831         return IntegerCache.cache[i + (-IntegerCache.low)];
832     return new Integer(i);
833 }
834
835 /**
836  * The value of the {@code Integer}.
837  */
```

Vagyis a -128 és az alapértelmezetten 127 (de ez konfigurálható) közötti értékekre egy már létező pool-ból fogjuk megkapni a nekünk kellő objektumot. Ami azt jelenti, hogy az x és a t ugyan azt az objektumot fogja viszakapni, vagyis ugyanarra a memóriacímre fognak mutatni. Éppen ezért le fog állni a while ciklus az $x!=t$ feltétel miatt. Ezzel szemben, ha az érték 128, akkor nem az előre elkészített poolból fogják megkapni az értéküknek megfelelő objektumot, hanem a `return new Integer(i);`-vel fognak értéket kapni. Ez azt jelenti, hogy x-nek és t-nek két különböző című objektuma lesz. Ebben az esetben pedig már az $x!=t$ feltétel is igaz lesz, aminek az eredménye pedig egy végtelen ciklus.

12.3. 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: [Forrás](#)

A sokak által tanult összehasonlítási módszer szerint az egyenlőségjel bal oldalára kell kerülnie a változónak minden esetben. Azonban ezzel az a probléma, hogy ha a változónak null az értéke, akkor a programunk le fog állni egy `java.lang.NullPointerException`-nel. Erre ad megoldást a Yoda conditions, aminek az a lényege, hogy az összehasonlítás bal oldalára írjuk az értéket, a jobb oldalára pedig a változót.

```
import java.util.Scanner;
public class Main {
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        String pelda = null;
        String legyen;
        System.out.println("Kapjunk-e NullPointerException-t? I/N");
        for(;;){
            legyen = sc.nextLine();

            if( legyen.equalsIgnoreCase("I") ){
                if( pelda.equals("abrakadabra") ){
                    break;
                }
            }
            else if( legyen.equalsIgnoreCase("N")){
                if(!"abrakadabra".equals(pelda) ){
                    System.out.println("Nem Kaptunk.");
                    break;
                }
            }
            else{
                System.out.println("Nem Tudom értelmezni amit írtál. ↵
                próbáld újra.");
            }
        }
    }
}
```

Ebben a példában a 'pelda' egy String aminek null az értéke. A felső elágazás során a program le fog állni a fent említett `NullPointerException` hibával, mivel a stringet egy null pointerhez hasonlítanánk, ami nem lehetséges. Ezzel szemben az alsó esetben szimplán csak egy hamis értéket fogunk kapni eredményként. És a végeredmény:

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Arroway$ java Yoda
Kapjunk-e NullPointerException-t? I/N
N
Nem Kaptunk.
viktor@viktorubuntu:~/prog2/src/prog2/Arroway$ java Yoda
Kapjunk-e NullPointerException-t? I/N
I
Exception in thread "main" java.lang.NullPointerException
    at Yoda.main(Yoda.java:15)
viktor@viktorubuntu:~/prog2/src/prog2/Arroway$
```

12.4. Kódolás from scratch

Induljunk ki ebből a tudományos közleményből: <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/bbp-alg.pdf> és csak ezt tanulmányozva írjuk meg Java nyelven a BBP algoritmus megvalósítását! Ha megakadsz, de csak végső esetben: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#pi_jegyei (mert ha csak lemásolod, akkor pont az a fejlesztői élmény marad ki, melyet szeretném, ha átélnél).

Megoldás videó:

Megoldás forrása: [Forrás](#)

Ebben a feladatban a BBP algoritmust kell megírunk. A program a main-nel kezdődik:

```
public static void main(String[] args) {
    int k = 6;
    System.out.println(magic(k));
}
```

Amiben a `k` változó értéke azt mondja meg, hogy pi-nek 10 a hányadikon számjegyétől kezdve írjuk ki a következő pár számjegyet hexadecimális alakban. Jelen esetben `k` értéke 6, ami az jelenti, hogy pi-nek az első 10^6 számjegye utáni pár számjegyét fogjuk megkapni. Majd kiíratjuk a `magic(k)` nevű függvény eredményét.

Ezután következik a `magic` függvény, ami egy értékadással kezdődik:

```
double s1 = solve(Math.pow(10,k), 1);
double s4 = solve(Math.pow(10,k), 4);
double s5 = solve(Math.pow(10,k), 5);
double s6 = solve(Math.pow(10,k), 6);
```

Az `s1`, `s4`, `s5`, és `s6` változóknak úgy adunk értéket, hogy meghívjuk a `solve()` függvényt, ami pedig majd a `mod()` függvényt fogja használni. A `solve()` függvény egyik paramétere az 10^k lesz, a másik pedig egy szám. Ha megvan az értékadás, akkor ugyan ezeket a változókat ráeresztjük a `cut()` függvényre:

```
s1 = cut(s1);
s4 = cut(s4);
```

```
s5 = cut(s5);  
s6 = cut(s6);
```

A `cut()` függvénynek annyi a feladata, hogy visszaadja a paraméterként kapott `double` változó nem egész részét. Ezt úgy csinálja, hogy ha az értéke negatív, akkor hozzáadja saját magához saját maga egész részét, ha viszont pozitív, akkor pedig kivonja.

```
public static double cut(double db) {  
    if(db < 0) {  
        return db - (int)db+1;  
    }  
    else {  
        return db - (int)db;  
    }  
}
```

Ezek után létrehozuk, és értéket adunk `pi`-nek, majd ennek az értéknek kiszámoljuk a nem egész részét, valamint létrehozuk még a hexadecimális jeleket és a végeredményt is. Egy `while` ciklusban addig számoljuk `pi` értékét, amíg a nem egész részének az értéke nem egyenlő nullával. Ha nem egyenlő akkor `pi` értékét megszorozzuk 16-tal. Majd, ha az egész része `pi`-nek nagyobb vagy egyenlő mint 10, akkor a végeredményt konkatenáljuk az értéknek megfelelő Hexadecimális jellel. Egyébként pedig szimplán csak a stringgé alakított számjegyeket konkatenáljuk a végeredménnyel, majd pedig elvesszük `pi`-ből az egész részét, és kezdődik előről a ciklus. Legvégül pedig visszaadjuk a végeredményt.

```
double pi = 4*s1 -2*s4 - s5 -s6;  
pi = cut(pi);  
String[] hexa = {"A", "B", "C", "D", "E", "F"};  
String result = "";  
while(cut(pi) != 0) {  
    pi = pi*16;  
    if((int)pi >= 10) {  
        result = result.concat(hexa[(int)pi - 10]);  
    }  
    else {  
        result = result.concat(Integer.toString((int)pi));  
    }  
    pi = cut(pi);  
}  
return result;
```

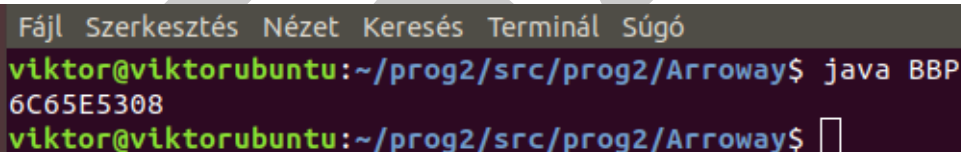
A `solve()` függvény egy összeget számol. Egy `for` ciklus addig megy, amíg az első kapott paraméter, azaz `d` értéke nagyobb, vagy egyenlő `i`-vel. A cikluson belül pedig minden egyes lépésnél hozzáadja az összeg értékéhez a `mod()` függvény által kiszámolt értéket.

```
public static double solve(double d, double num) {  
    double sum = 0.0;  
    for(int i = 0; i <= d; i++) {  
        sum += mod(16, (d-i), 8*i+num) / (8*i + num);  
    }  
    return sum ;  
}
```

Végül pedig a `mod()` függvény. Létrehoz két `double` változót `'t'` és `'r'` néven, és az 1 kezdőértéket adja mind a kettőnek. Ezután egy `while` ciklus addig megy amíg `'t'` kisebb vagy egyenlő mint `'n'`. Az `'n'` a második paramétere a függvénynek. A cikluson belül pedig minden egyes iterációban `'t'` értékét megszorozza kettővel. Ezek után következik még egy `while` ciklus, ami `break` utasítással fog leállni. Végül pedig a függvény visszaadja `'r'` értékét.

```
public static double mod(double b, double n, double k) {
    double t = 1;
    double r = 1;
    while(t <= n) {
        t = t * 2;
    }
    while(true) {
        if(n >= t) {
            r = (b * r) % k;
            n = n - t;
        }
        t = t / 2;
        if(t >= 1) {
            r = (r*r) % k;
        }
        else {
            break;
        }
    }
    return r;
}
```

Az eredmény pedig:



```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Arrowway$ java BBP
6C65E5308
viktor@viktorubuntu:~/prog2/src/prog2/Arrowway$
```


13. fejezet

Helló, Liskov!

13.1. 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 fólia) (számos példa szerepel az elv megsértésére az UDPROG repóban, lásd pl. [source/binom/Batfai-Barki/madarak/](#))

Ez a feladat egy nagyon jól bemutatja az öröklődést, az osztályhierarchiát, valamint a metódustúlterhelést a gyakorlatban. A liskov elv azt jelenti, hogy ha S altípusa T-nek, akkor minden olyan helyen ahol T-t felhasználjuk S-t is minden gond nélkül behelyettesíthetjük anélkül, hogy a programrész tulajdonságai megváltoznának. Vagyis ha S osztály T osztály leszármazottja, akkor S szabadon behelyettesíthető minden olyan helyre (paraméter, változó, stb...), ahol T típust várunk. Ezt kellett megsérteni c++ ban és java-ban is. Ehhez én egy madár-pingvin szülőosztály-gyerekosztály kombinációt használtam.

C++:

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

class Madar {
public:
    void repul() {
        cout << "Repül";
    };
};

class Sas : public Madar
{};

class Pingvin : public Madar
{};
```

És java:

```
static class Madar{
    public void repul(){
        System.out.println("Repülök");
    }
};
```

```
    }  
}  
  
static class Sas extends Madar{  
  
}  
  
static class Pingvin extends Madar{  
  
}
```

A programok mindkét esetben úgy kezdődnek, hogy létrehozuk a szülőosztályt, ami a madár. Ennek az osztálynak van egy olyan metódusa, hogy `repul()` ami jelen esetben csak annyit csinál, hogy kiírja a konzolra azt, hogy Repül vagy Repülök. Majd jön két újabb osztály, amiket a Madárból származtatunk, vagyis ők is meg fogják kapni a `repul()` metódust. Az egyik osztály a sas, ami tud repülni és még madár is, szóval itt nincs probléma. Azonban a másik osztály a Pingvin, ami igaz, hogy madár, de repülni nem tud.

C++:

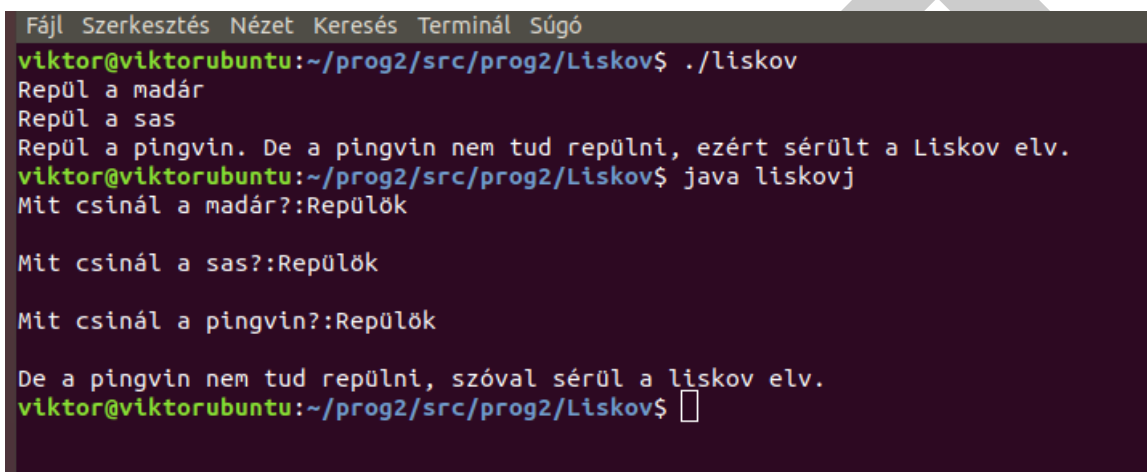
```
int main ( int argc, char **argv )  
{  
    Madar madar;  
    madar.repul();  
    cout << " a madár\n";  
  
    Sas sas;  
    sas.repul();  
    cout << " a sas\n";  
  
    Pingvin pingvin;  
    pingvin.repul();  
    cout << " a pingvin. De a pingvin nem tud repülni, ezért sérült a ↔  
        Liskov elv.\n";  
}
```

és Java:

```
public static void main(String args[]){  
    Madar madár = new Madar();  
    Sas sas = new Sas();  
    Pingvin pingvin = new Pingvin();  
  
    System.out.print("Mit csinál a madár?:");  
    madár.repul();  
  
    System.out.print("\nMit csinál a sas?:");  
    sas.repul();  
  
    System.out.print("\nMit csinál a pingvin?:");
```

```
pingvin.repul();  
System.out.println("\nDe a pingvin nem tud repülni, szóval sérül a liskov ↔  
    elv.");  
}
```

Ezek után a main-ben mind a két esetben példányosítunk, azaz létrehozunk egy madarat, egy sast és egy pingvint is. Majd mind a három objektummal meghívjuk a repül függvényt. Az első kettővel nincs, és nem is lenne gond, mivel alapvetően tudnak repülni, viszont a pingvin, mint tudjuk nem tud repülni. Azonban ez a pingvin ahelyett, hogy hibát dobna a program, boldogan repked a virtuális térben, ami nekünk nem jó.



```
Fájl Szerkesztés Nézet Keresés Terminál Súgó  
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ ./liskov  
Repül a madár  
Repül a sas  
Repül a pingvin. De a pingvin nem tud repülni, ezért sérült a Liskov elv.  
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ java liskovj  
Mit csinál a madár?:Repülök  
  
Mit csinál a sas?:Repülök  
  
Mit csinál a pingvin?:Repülök  
  
De a pingvin nem tud repülni, szóval sérül a liskov elv.  
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$
```

Erre egy megoldás a jobb OO tervezés. Vagyis ha például ha a Madár osztályunk megmadarna, de lenne két származtatott osztálya. Az egyik osztályba kerülnének a repülni tudó madarak, a másikba pedig azok a madarak, amik nem tudnak repülni. És ezekből az osztályokból származtathatnánk tovább a sast, ami egy repülni tudó madár, illetve a pingvint is, ami pedig nem tud repülni.

13.2. 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)

Megoldás videó:

Ebben a feladatban be kellett bizonyítani, hogy ha a gyerekosztályban létrehozunk egy metódust, akkor ha a gyerekosztályt szülőosztályként szeretnénk használni, akkor a gyerekosztály saját metódusait nem fogjuk tudni használni. Én az előző feladathoz hasonlóan maradtam a Madár-Sas példánál. A Madár a szülő, a Sas a gyerekosztály.

C++ kód:

```
class Madar{  
public:  
  
};
```

```
class Sas : public Madar{
public:
    void repul() {
        std::cout << "Repül";
    }

};
```

És java:

```
static class Madar{
    protected int szarnyhossz;
    public void setSzarnyhossz(int szarnyhossz) {
        this.szarnyhossz = szarnyhossz;
    }
}

static class Sas extends Madar{
    public int getSzarnyhossz(){
        return szarnyhossz;
    }
}
```

A C++ kód esetében a gyerekosztálynak van egy `repul()` metódusa, ami szimplán csak kiírja a konzolra, hogy "Repül". Ezzel szemben a java példa egy kicsit bonyolultabb, mivel itt a szülő osztálynak, azaz a madárnak van egy `szarnyhossz` tulajdonsága, illetve egy `setSzarnyhossz()` metódusa, amivel a `szarnyhossz` tulajdonságot lehet beállítani. A gyermek osztály természetesen megörökli ezt a tulajdonságot, illetve metódust, szóval neki is szabadon belehet állítani a `szarnyhossz` tulajdonságát. De ezek mellett van egy `getSzarnyhossz()` metódusa is, ami visszaadja a sas objektum `szarnyhossz`-át. Ez eddig teljesen normális, az érdekesség akkor kezdődik, amikor a main-be érünk.

C++:

```
int main(){

    Madar* sas = new Sas();
    Sas* sas2 = new Sas();

    sas->repul();
    sas2->repul();
}
```

És java:

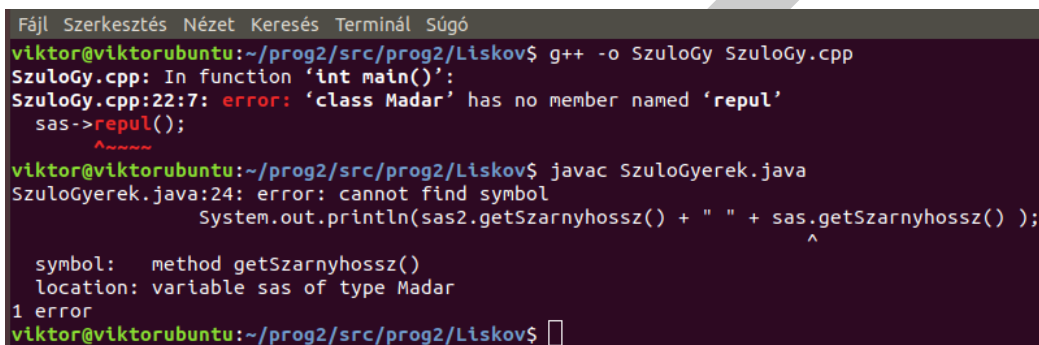
```
public static void main(String args[]){
    Madar sas = new Sas();
    sas.setSzarnyhossz(80);

    Sas sas2 = new Sas();
    sas2.setSzarnyhossz(50);
}
```

```
System.out.println(sas2.getSzarnyhossz() + " " + sas.getSzarnyhossz() ) ←  
;  
}
```

Mind a két esetben létrehozunk egy Sas típust Sas típusként, amivel nincs is gond, de létrehozunk egy Madár típust is Sas típusként. Ezek után a C++ kódban mind a két objektumok megpróbáljuk reptetni a `repul()` metódust használva. A Sas típusnak ezzel nem is lenne gondja, azonban a Madárnak igen.

A java kód itt is egy kicsit másképp működik. Itt a két típus létrehozása után mind a kettőnek beállítjuk a szárnyhosszát a `setSzarnyhossz()` segítségével, amivel nincs is gond, mivel ez eredetileg a Madár osztály metódusa, amit a Sas megörökölt. Azonban ezek után a `getSzarnyhossz()` metódus segítségével megpróbáljuk kiírni mind a két objektum szárnyhosszát, ami csak az egyik esetben sikerülne. A végeredmény pedig:



```
Fájl Szerkesztés Nézet Keresés Terminál Súlyó  
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ g++ -o SzuloGy SzuloGy.cpp  
SzuloGy.cpp: In function 'int main()':  
SzuloGy.cpp:22:7: error: 'class Madar' has no member named 'repul'  
    sas->repul();  
        ^~~~~~  
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ javac SzuloGyerek.java  
SzuloGyerek.java:24: error: cannot find symbol  
    System.out.println(sas2.getSzarnyhossz() + " " + sas.getSzarnyhossz() );  
                        ^  
symbol:   method getSzarnyhossz()  
location: variable sas of type Madar  
1 error  
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$
```

13.3. Anti OO

A BBP algoritmust 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:

Ehhez a feladathoz a Bárfai Norbert által [biztosított](#) forráskódokat használtam. A programokat egy 4.5 Ghz-re overclockolt AMD-FX8350 és 8GB ramot tartalmazó gépen futtattam. Lássuk is az eredményeket:

A java program 10^6 számjegyet 0.906 másodperc alatt, 10^7 számjegyet 11.132 másodperc alatt, 10^8 számjegyet pedig 135.7 másodperc alatt számolta ki. Ahhoz, hogy hány számjegyet számoljon ki a program, a következőt kell alkalmazni. Ahhoz, hogy 6 számjegyet számoljon ki, nem kell semmit se módosítani az alap programon. Ahhoz, hogy 7 számjegyet számoljon ki, az alábbi for ciklus fejében kellett hozzáírni a d inicializálásához, illetve a feltételhez egy-egy nullás számjegyet. Ahhoz pedig, hogy 8 számjegyet számoljon ki, ugyan abban a for ciklusban ugyan azokra a helyekre ismét oda kell írni még egy-egy számjegyet.

```
for(int d=100000000; d<1000000001; ++d)
```

```
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ javac AntiJava.java
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ java AntiJava
6
0.906
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ javac AntiJava.java
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ java AntiJava
7
11.132
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ javac AntiJava.java
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ java AntiJava
12
135.7
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$
```

A C++ program 10^6 számjegyet 2.37792 másodperc alatt, 10^7 számjegyet 26.5749 másodperc alatt, 10^8 számjegyet pedig 296.292 másodperc alatt számolta ki. Itt is érvényes az a tény, hogy ahhoz, hogy 6 számjegyet számoljon ki, nem kell semmit se módosítani az alap programon. Ahhoz, hogy 7 számjegyet számoljon ki, az alábbi for ciklus fejében kellett hozzáírni a d inicializálásához, illetve a feltételhet egy-egy nullás számjegyet. Ahhoz pedig, hogy 8 számjegyet számoljon ki, ugyan abban a for ciklusban ugyan azokra a helyekre ismét oda kell írni még egy-egy számjegyet.

```
for(int d=100000000; d<1000000001; ++d)
```

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ g++ -o AntiCpp AntiCpp.cpp
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ ./AntiCpp
6
2.37792
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ g++ -o AntiCpp AntiCpp.cpp
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ ./AntiCpp
7
26.5749
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ g++ -o AntiCpp AntiCpp.cpp
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ ./AntiCpp
12
296.292
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$
```

A C program 10^6 számjegyet 1.243691 másodperc alatt, 10^7 számjegyet 15.384056 másodperc alatt, 10^8 számjegyet pedig 186.029669 másodperc alatt számolta ki. És igen. Ahhoz, hogy 6 számjegyet számoljon ki, nem kell semmit se módosítani az alap programon. Ahhoz, hogy 7 számjegyet számoljon ki, az alábbi for ciklus fejében kellett hozzáírni a d inicializálásához, illetve a feltételhet egy-egy nullás számjegyet. Ahhoz pedig, hogy 8 számjegyet számoljon ki, ugyan abban a for ciklusban ugyan azokra a helyekre ismét oda kell írni még egy-egy számjegyet.

```
for(int d=100000000; d<1000000001; ++d)
```

```

Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ gcc -o AntiC AntiC.c -lm
AntiC.c:77:1: warning: return type defaults to 'int' [-Wimplicit-int]
main ()
^~~~
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ ./AntiC
6
1.243691
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ gcc -o AntiC AntiC.c -lm
AntiC.c:77:1: warning: return type defaults to 'int' [-Wimplicit-int]
main ()
^~~~
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ ./AntiC
7
15.384056
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ gcc -o AntiC AntiC.c -lm
AntiC.c:77:1: warning: return type defaults to 'int' [-Wimplicit-int]
main ()
^~~~
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ ./AntiC
12
186.029669
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ 

```

És végül a C# program 10^6 számjegyet 0.950131 másodperc alatt, 10^7 számjegyet 11.643125 másodperc alatt, 10^8 számjegyet pedig 140.350363 másodperc alatt számolta ki. Végül pedig szintén itt is ahhoz, hogy 6 számjegyet számoljon ki, nem kell semmit se módosítani az alap programon. Ahhoz, hogy 7 számjegyet számoljon ki, az alábbi for ciklus fejében kellett hozzáírni a d inicializálásához, illetve a feltételhet egy-egy nullás számjegyet. Ahhoz pedig, hogy 8 számjegyet számoljon ki, ugyan abban a for ciklusban ugyan azokra a helyekre ismét oda kell írni még egy-egy számjegyet.

```
for(int d=100000000; d<1000000001; ++d)
```

```

viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ mono AntiCs.exe
6
0,950131
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ mcs AntiCs.cs
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ mono AntiCs.exe
7
11,643125
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ mcs AntiCs.cs
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ mono AntiCs.exe
12
140,350363
viktor@viktorubuntu:~/prog2/src/prog2/Liskov$ 

```

Az egész összesítve egy táblázatba:

Ebből az látszik, hogy a sort a Java és a C# fej fej mellett haladva vezeti, 10^6 számjegynél a C# nyer, ám a másik két esetben pedig a Java. A dobogó harmadik helyét a C nyelv foglalja el, leghátul pedig a C++ kullog.

	Java	C++	C#	C
10^6	0.906	2.37792	0.950131	1.243691
10^7	11.132	26.5749	11.643125	15.384056
10^8	135.7	296.292	140.350363	186.029669

13.1. táblázat. Összehasonlítás

13.4. Ciklomatikus komplexitás

Számoljuk ki valamelyik programunk függvényeinek ciklomatikus komplexitását! Lásd a fogalom tekintetében a https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_2.pdf (77-79 fóliát)!

A ciklomatikus komplexitás egy forráskód összetettségét jelenti, amit gráfelmélettel kell kiszámolni. A képlet hozzá $M=E-N+2P$ ahol E a gráf elemeinek a száma, N a gráfban lévő csúcsok száma, és P pedig az összefüggő komponensek száma. Én ezt a feladatot a <http://www.lizard.ws/> oldal segítségével oldottam meg. Szimplán csak be kell illeszteni a forráskódot, ki kell választani, hogy milyen nyelven van írva a program, és ki is számolja nekünk. Én az első csokorban átnézett BBP és LZWBinFa programok java változatait számoltattam ki. Az eredmények:

Code analyzed successfully.

File Type **.java**

Token Count **1174**

NLOC **204**

Function Name	NLOC	Complexity	Token #	Parameter #
LzwBinFa::LzwBinFa	3	1	9	
LzwBinFa::egyBitFeldolg	22	4	104	
LzwBinFa::kiir	4	1	26	
LzwBinFa::kiir	4	1	22	
LzwBinFa::Csomopont::Csomopont	5	1	21	
LzwBinFa::Csomopont::nullasGyerek	3	1	8	
LzwBinFa::Csomopont::egyGyerek	3	1	8	
LzwBinFa::Csomopont::ujNullasGyerek	3	1	11	
LzwBinFa::Csomopont::ujEgyGyerek	3	1	11	
LzwBinFa::Csomopont::getBetu	3	1	8	
LzwBinFa::kiir	15	3	107	
LzwBinFa::getMelyseg	5	1	21	
LzwBinFa::getAtlag	6	1	32	
LzwBinFa::getSzoras	13	2	68	
LzwBinFa::rmelyseg	11	3	51	
LzwBinFa::ratlag	12	4	66	
LzwBinFa::rszoras	12	4	78	
LzwBinFa::usage	3	1	14	
LzwBinFa::main	60	14	401	

Itt viszont csak a különböző programrészek komplexitását kapjuk meg, nem pedig az egész programét. Ha összeadjuk a programrészek komplexitását akkor megkapjuk hogy a teljes program ciklomatikus komplexitása 45. De most nézzük a BBP algoritmust, az kicsit egyszerűbb:

Code analyzed successfully.

File Type

.java

Token Count

468

NLOC

67

Function Name	NLOC	Complexity	Token #	Parameter #
BBP::main	4	1	26	
BBP::magic	25	3	224	
BBP::cut	8	2	36	
BBP::solve	7	2	63	
BBP::mod	21	5	99	

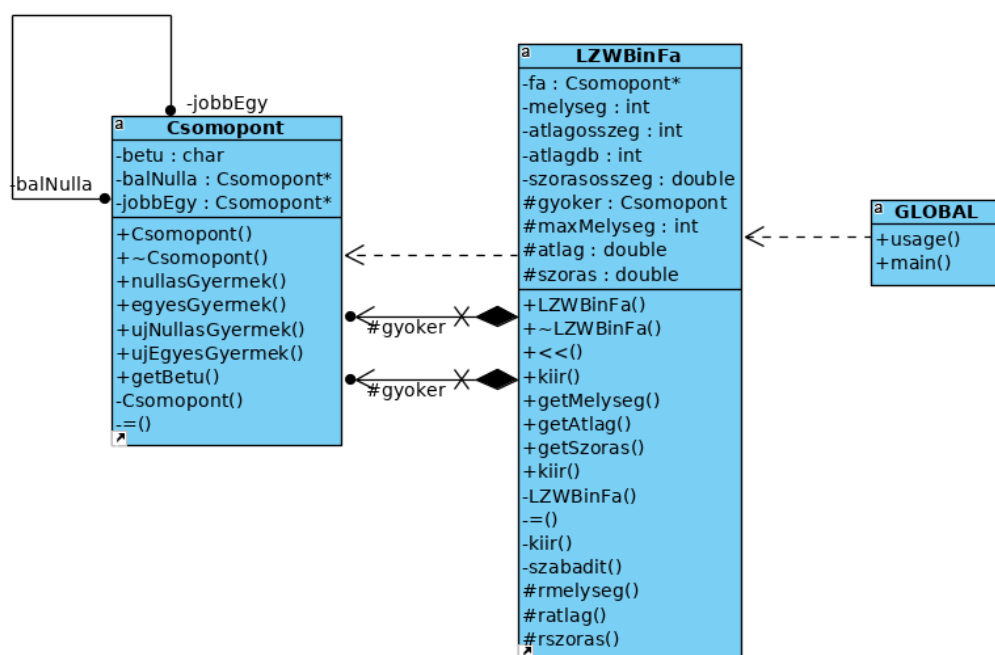
Itt is ugyan az érvényes, mint az előző programnál. Azaz a teljes program ciklomatikus komplexitásának a meghatározásához össze kell adnunk az egyes programrészek ciklomatikus komplexitását. Ez által azt kapjuk, hogy a BBP algoritmus java változatának ciklomatikus komplexitása 13.

14. fejezet

Helló, Mandelbrot!

14.1. Reverse engineering UML osztálydiagram

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



Ebben a feladatban a BevProgon és Prog1-en már tárgyalt [z3a7.cpp](#) program forrásából kellett UML diagramot létrehozni. Én ehhez a [Visual Paradigm](#) 30 napos ingyenes próbaverzióját használtam. A diagramot roppant egyszerű legenerálni. A visual paradigm feltelepítése és elindítása után oda kell navigálnunk, hogy Tools > Code > Instant Reverse..., majd pedig ki kell választani, hogy milyen nyelvű forrásból szeretnénk diagramot generálni, és meg kell adni a fájl helyét. Ezek után már csak ki kell választani azokat az osztályokat, amelyeket meg szeretnénk mutatni a diagrammon, és kész is. Az általam generált diagramot a fentebb

található képen lehetett látni. Ez a feladat nagyon jó gyakorlati tudást ad a modellező nyelvekhez, valamint az UML-hez.

Ezen kívül rá kell mutatni az aggregáció és kompozíció kapcsolatára. Azonban, mielőtt ezt megtehetnénk, először az asszociáció fogalmát kell tisztázni.

Ha egy modellben két osztálynak kommunikálnia kell egymással, akkor szükségünk van egy kapcsolatra a két osztály között. Ezt a kapcsolatot reprezentálja az asszociáció. Az asszociációt egy a két osztály között lévő vonal, valamint az azon lévő irányt mutató nyíl(ak) jelöli(k). Ha a vonal mindkét oldalán van nyíl, akkor az asszociáció kétirányú.

Az aggregáció és kompozíció az asszociáció részhalmazai, vagyis az asszociáció különleges esetei. Mind a két esetben egy osztály objektuma "birtokol" egy másik osztály másik objektumát, de van a kettő között egy kis különbség.

Az aggregáció egy olyan kapcsolatot jelent, amiben a gyerek a szülőtől függetlenül létezhet. Például ha van tanóra, ami a szülőosztály, és tanuló, ami a gyerekosztály. Ha töröljük a tanórát, attól a tanulók még léteznek.

Ezzel szemben a kompozíció esetében egy olyan kapcsolatról van szó, amiben a gyerek nem létezhet a szülő nélkül. Például ha van egy ház szülőosztályunk, és egy szoba gyerekosztályunk. A szoba nem létezhet a ház nélkül.

Az aggregációt és a kompozíciót is vonal+rombusz kombinációval lehet jelölni, azonban az aggregációnál a rombusz üres, a kompozíciónál pedig nem.

Ezek alapján meg tudjuk mondani, hogy a fentebb látható ábrán a

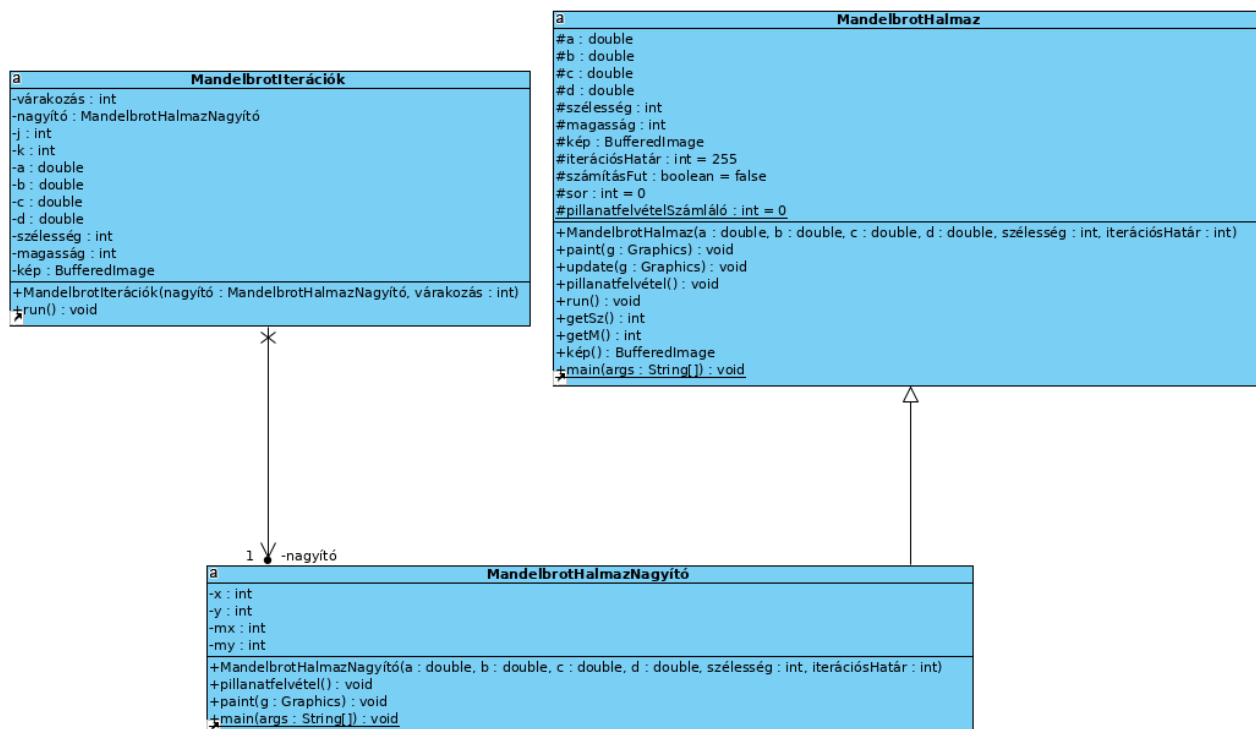
Csomopont gyoker;

elem a kompozíció.

14.2. Forward engineering UML osztálydiagram

UML-ben tervezzünk osztályokat és generáljunk belőle forrást!

Ebben a feladatban UML-ben kellett osztályokat megtervezni, majd pedig a diagramból forrást generálni. Ez a feladat nagyon jó gyakorlati tudást ad a modellező nyelvekhez, valamint az UML-hez. Én ehhez a feladathoz, ha már Mandelbrot a csokor neve, a MandelbrotHalmazt próbáltam meg lemodellezni a visual paradigm nevű szoftver segítségével. Ehhez a következő diagrammot sikerült összeállítani:



Ahogy láthatjuk a MandelbrotIterációk forrásfájl nagyító objektuma asszociációban áll a MandelbrotHalmazNagyító osztállyal, ami pedig kompozícióban áll a MandelbrotHalmaz osztállyal. Ezek után a kód generálása már gyerekjáték. Annyit kell tenni, hogy rákattintunk a Tools > generate java code... opcióra, majd pedig a kapott menüben megadjuk azt, hogy hova szeretnénk generálni a forrást, és a visual paradigm automatikusan legenerálja nekünk. Ha a fent megjelölt forrást összehasonlítjuk az eredetivel, akkor elmondhatjuk, hogy a generált forrás nagyon hasonlít az eredetire, hiszen a források szerkezete megegyezik.

```

public void update(java.awt.Graphics g) {
    // TODO - implement MandelbrotHalmaz.update
    throw new UnsupportedOperationException();
}

public void pillanatfelvétel() {
    // TODO - implement MandelbrotHalmaz.pillanatfelvétel
    throw new UnsupportedOperationException();
}

public void run() {
    // TODO - implement MandelbrotHalmaz.run
    throw new UnsupportedOperationException();
}

public int getSz() {
    // TODO - implement MandelbrotHalmaz.getSz
    throw new UnsupportedOperationException();
}

```

```
public int getM() {
    // TODO - implement MandelbrotHalmaz.getM
    throw new UnsupportedOperationException();
}

public java.awt.image.BufferedImage kép() {
    // TODO - implement MandelbrotHalmaz.kép
    throw new UnsupportedOperationException();
}

/**
 *
 * @param args
 */
public static void main(String[] args) {
    // TODO - implement MandelbrotHalmaz.main
    throw new UnsupportedOperationException();
}
```

Ez a generált forrásnak egy része. És ha ezt összehasonlítjuk az eredeti forrásnak ugyan ezen részével, akkor észrevehetjük, hogy a két forrás szerkezete megegyezik, viszont nyilvánvalóan, ha kódot generálunk, akkor csak a függvények létrehozása történik meg, a törzsük viszont üres marad. Azonban, ha már valakinek összeállt a fejében egy ötlet arról, hogy hogyan fog kinézni a forrásának a felépítése, akkor annak az embernek nagyon jól tud jönni, ha azt a vázat létre tudja hozni egy UML diagramban, és abból generálni tud forrást.

14.3. Egy esettan

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

Ebben a feladatban a Szoftverfejlesztés C++ nyelven című könyvben szereplő részletet kellett feldolgozni. Az elméleti része főképp az UML nyelvet mutatja be. Szó esik az osztálydiagrammokról, azon belül arról, hogy az osztályokat téglalapokban lehet ábrázolni, amik három részre vannak osztva. Ezen kívül szó esik még az osztály nevééről, ami a téglalap felső részében található meg, az osztály attribútumairól, amik a téglalap középső részében helyezkednek el, illetve az osztály műveleteiről is, amelyek pedig a téglalap alsó részében találhatók.

Majd pedig szó esik a láthatóságról, ami ugyebár lehet public, protected, és private. Ezeken kívül szó esik még a kapcsolatokról is, amit a könyv egy üres háromszöggel jelöl, valamit szó esik az asszociációkról, amelyet pedig egy nyíllal jelöl a könyv. Említést kap a kompozíció aminek a jelölése egy teli rombusz, illetve az aggregáció is, aminek pedig egy üres rombusz. Valamint ezek jelentésére is kitér. Ahogy azt már egy korábbi feladat leírásában említettem:

Az aggregáció egy olyan kapcsolatot jelent, amiben a gyerek a szülőtől függetlenül létezhet. Például ha van tanóra, ami a szülőosztály, és tanuló, ami a gyerekosztály. Ha töröljük a tanórát, attól a tanulók még léteznek. Ezzel szemben a kompozíció esetében egy olyan kapcsolatról van szó, amiben a gyerek nem létezhet a szülő nélkül. Például ha van egy ház szülőosztályunk, és egy szoba gyerekosztályunk. A szoba nem létezhet a ház nélkül.

Végül pedig a kódgenerálásról és a kód visszafejtésről. Ezen belül arról, hogy pontosan mik ezek, ugye forward és reverse engineering, azaz a kész kódból UML diagram generálása, valamit kész UML diagramból kód generálása, amikről az első két feladat szólt. Éppen ezért lehet hogy nem utolsóként kellett volna megcsinálni ezt a feladatot, hanem elsőként. Mivel nagy segítség lehetett volna a többi feladat megoldásánál, ha már ismerem ezeket a dolgokat.

Ezek után következett maga az esettanulmány, ami egy program elkészítéséből állt. Maga a feladat egy számítógép kereskedéssel volt kapcsolatos. Elégge összetett programról van szó, ami támogatja a termékek állományból való betöltését, képernyőre történő listázását, állományba való kiírását, és az árképzés rugalmas alakítását. És ha ez még nem lenne elég, még a lehetséges jövőbeli befektetésekre is gondolni kell, azaz a teljesen új termékcsaládok értékesítésének bevezetésére is lehetőséget kell biztosítani. Magához a feladathoz volt megadva forrás, azonban még kell mellé írni saját magunktól is. Na de nézzük, hogy mi mit is jelent a programunk kódja

Az első a product osztály, ami a programunk szülőosztálya, magyarra lefordítva termék. Ebből az osztályból lesz származtatva a többi három osztály, a display, azaz kijelző, a harddisk, azaz merevlemez, és a compositeproduct, ami pedig az összetett termék, például egy kész számítógép.

A product osztálynak három tagváltozója van. Ezek a name, ami a termék neve InitialPrice, azaz a termék eredeti ára, illetve a dateOfAcquisition, ami pedig a termék beszerzési ideje. Ezek a tagváltozók protectedek, éppen ezért szükség van getter függvényekre is hozzájuk. Valamint van két még másik getter függvényünk is, a getAge, ami a beszerzési, és a jelenlegi idő felhasználásával kiszámolja, hogy milyen idős az adott termék. Valamint a getCurrentPrice, ami a termék jelenlegi árát adná vissza, azonban jelenleg ennek a függvénynek a visszatérési értéke az eredeti ár. Maga az osztály:

```
time_t Product::getDateOfAcquisition() const {
    return dateOfAcquisition;
}

int Product::getInitialPrice() const {
    return initialPrice;
}

std::string Product::getName() const {
    return name;
}

Product::Product() {}

Product::Product(std::string name, int initialPrice, time_t ↵
    dateOfAcquisition): name(name), initialPrice(initialPrice),
    dateOfAcquisition(dateOfAcquisition) {
}

int Product::getAge() const{
    time_t currentTime;
    time(&currentTime);
    double timeDiffInSec = difftime(currentTime, dateOfAcquisition);
    return (int)(timeDiffInSec/(3600*24));
}
```

```
int Product::getCurrentPrice() const {
    return initialPrice;
}
```

Az `inputstream`, illetve az `outputstream` operátorok segítségével a következő függvények fogják elvégezni a termékek beolvasását, illetve kiíratását: `print()`, amely megadja a termék típusát, és nevét, a `printParams()`, amely megadja a termék paramétereit, azaz a eredeti árat, a beszerzési időt, a termék korát, és a termék jelenlegi árát, a `writeParamsToStream()`, amely megadja a termék nevét, eredeti árát, és a beszerzési idejét stringgé alakítva, és a `loadParamsFromStream()`, amely pedig beolvassa a termékeket. Maga az osztály:

```
void Product::print(std::ostream &os) const {
    os << "Type: " << getType() << ", ";
    os << "Name: " << getName();
    printParams(os);
}

void Product::printParams(std::ostream &os) const {
    char strDateOfAcquisition[9];
    strftime(strDateOfAcquisition, 9, "%Y%m%d",
             gmtime(&dateOfAcquisition));

    os << ", " << "Initial price: " << initialPrice
       << ", " << "Date of acquisition: " << strDateOfAcquisition
       << ", " << "Age: " << getAge()
       << ", " << "Current price: " << getCurrentPrice();
}

void Product::writeParamsToStream(std::ostream &os) const {
    char strDateOfAcquisition[9];
    tm* t = localtime(&dateOfAcquisition);
    strftime(strDateOfAcquisition, 9, "%Y%m%d", t);
    os << " " << name << " " << initialPrice << " " << strDateOfAcquisition;
}

void Product::loadParamsFromStream(std::istream &is) {
    is >> name;
    is >> initialPrice;

    char buff[9];
    is.width(9);
    is >> buff;
    if (strlen(buff) != 8)
        throw range_error("Invalid time format");

    char workBuff[5];
    tm t;
    int year;
    strncpy(workBuff, buff, 4); workBuff[4] = '\0';
    year = atoi(workBuff); t.tm_year = year - 1900;
```



```
    strncpy(workBuff, &buff[4], 2); workBuff[2] = '\\0';
    t.tm_mon = atoi(workBuff) - 1;
    strncpy(workBuff, &buff[6], 2); workBuff[2] = '\\0';
    t.tm_mday = atoi(workBuff);
    t.tm_hour = t.tm_min = t.tm_sec = 0;
    t.tm_isdst = -1;

    dateOfAcquisition = mktime(&t);
}

std::istream& operator>>(istream& is, Product& product) {
    product.loadParamsFromStream(is);
    return is;
}

std::ostream& operator<<(ostream& os, Product& product) {
    os <<product.getCharCode();
    product.writeParamsToStream(os);
    return os;
}
```

A harddisk, azaz merevlemez osztálynak a tagváltozói szintén a name, initialPrice, dateOfAcquisition, amelyek megtalálhatóak a product osztályban is, azonban, itt még van egy speedRPM változó is, amely megmondja, hogy hány RPM-es a merevlemez. Ezen kívül itt az io függvényekhez hozzá lett fűzve, hogy az RPM-et is ki kell írni, illetve be kell olvasni. Valamint a getCurrentPrice() függvény itt már úgy működik, hogy ha a termék fiatalabb 30 napnál, akkor az eredeti árat, ha 30 és 90 nap közötti, akkor az eredeti ár 80 százalékát, ha pedig idősebb mint 90 nap, akkor pedig az eredeti ár 80 százalékát adja vissza jelenlegi árként. Illetve található egy getSpeedRPM() függvény is, ami visszaadja a merevlemez RPM értékét. Maga az osztály:

```
int HardDisk::getCurrentPrice() const{
    int ageInDays = getAge();
    if(ageInDays < 30)
        return initialPrice;
    else if (ageInDays >= 30 && ageInDays < 90)
        return (int)(0.9 * initialPrice);
    else
        return (int)(0.8 * initialPrice);
}

HardDisk::HardDisk() {}

HardDisk::HardDisk(std::string name, int initialPrice, time_t ←
    dateOfAcquisition, int speedRPM):
    Product(name, initialPrice, dateOfAcquisition), speedRPM(speedRPM) {}

int HardDisk::getSpeedRPM() const {
    return speedRPM;
}
```

```
void HardDisk::printParams(std::ostream& os) const {
    Product::printParams(os);
    os << ", " << "SpeedRPM: " << speedRPM;
}

void HardDisk::writeParamsToStream(std::ostream &os) const {
    Product::writeParamsToStream(os);
    os << ' ' << speedRPM;
}

void HardDisk::loadParamsFromStream(std::istream &is) {
    Product::loadParamsFromStream(is);
    is >> speedRPM;
}
```

A Display osztály tagváltozói a name azaz név, az initialPrice, azaz kezdő ár, a dateOfAcquisition, azaz beszerzési idő, az inchWidth, azaz a szélesség col-ban, illetve az inchHeight azaz a magasság colban. A két új tagváltozó kap gettereket, illetve az input output részben is beolvastatjuk, illetve kiíratatjuk ezeknek a változóknak az értékeit a programmal. A getCurrentPrice() pedig ugyan úgy működik itt is, mint a merevlemez esetében, azaz ha a termék fiatalabb 30 napnál, akkor az eredeti árat, ha 30 és 90 nap közötti, akkor az eredeti ár 80 százalékát, ha pedig idősebb mint 90 nap, akkor pedig az eredeti ár 80 százalékát adja vissza jelenlegi árként. Maga az osztály:

```
void Display::printParams(std::ostream& os) const {
    Product::printParams(os);
    os << ", " << "InchWidth: " << inchWidth;
    os << ", " << "InchHeight: " << inchHeight;
}

void Display::writeParamsToStream(std::ostream &os) const {
    Product::writeParamsToStream(os);
    os << ' ' << inchWidth << ' ' << inchHeight;
}

void Display::loadParamsFromStream(std::istream &is) {
    Product::loadParamsFromStream(is);
    is >> inchWidth >> inchHeight;
}

Display::Display() {}

Display::Display(std::string name, int initialPrice, time_t ↵
    dateOfAcquisition, int inchWidth, int inchHeight):
    Product(name, initialPrice, dateOfAcquisition), inchWidth(inchWidth), ↵
    inchHeight(inchHeight) {}

int Display::getCurrentPrice() const {
    int ageInDays = getAge();
    if(ageInDays < 30)
        return initialPrice;
```

```
    else if (ageInDays >= 30 && ageInDays < 90)
        return (int)(0.9 * initialPrice);
    else
        return (int)(0.8 * initialPrice);
}

int Display::getInchWidth() const {
    return inchWidth;
}

int Display::getInchHeight() const {
    return inchHeight;
}
```

És végül pedig képek a működésről:

```
viktor@viktorubuntu:~/prog2/src/prog2/Mandelbrot/esettan$ g++ *.cpp -o esettan
viktor@viktorubuntu:~/prog2/src/prog2/Mandelbrot/esettan$ ./esettan
Test1: create inventory and printing it to the screen.
0.: Type: Display, Name: TFT1, Initial price: 30000, Date of acquisition: 20191124,
Age: 0, Current price: 30000, InchWidth: 13, InchHeight: 12
1.: Type: HardDisk, Name: WD, Initial price: 25000, Date of acquisition: 20191124,
Age: 0, Current price: 25000, SpeedRPM: 7500
Press any key to continue...

Test2: loading inventory from a file (computerproducts.txt), printing it, and then
writing it to a file (computerproducts_out.txt).
End of reading product items.The content of the file is:
0.: Type: Display, Name: TFT1, Initial price: 30000, Date of acquisition: 20011001,
Age: 6627, Current price: 24000, InchWidth: 12, InchHeight: 13
1.: Type: Display, Name: TFT2, Initial price: 35000, Date of acquisition: 20060930,
Age: 4802, Current price: 28000, InchWidth: 10, InchHeight: 10
2.: Type: ComputerConfiguration, Name: ComputerConfig1, Initial price: 70000, Date
of acquisition: 20060930, Age: 4802, Current price: 70000
Items:
0. Type: Display, Name: TFT3, Initial price: 30000, Date of acquisition: 20011001,
Age: 6627, Current price: 24000, InchWidth: 12, InchHeight: 13
1. Type: HardDisk, Name: WesternDigital, Initial price: 35000, Date of acquisition
: 20060930, Age: 4802, Current price: 28000, SpeedRPM: 7000
3.: Type: HardDisk, Name: Maxtor, Initial price: 25000, Date of acquisition: 200502
28, Age: 5381, Current price: 20000, SpeedRPM: 7000

The content of the inventory has been written to computerproducts_out.txt

Done.
viktor@viktorubuntu:~/prog2/src/prog2/Mandelbrot/esettan$
```

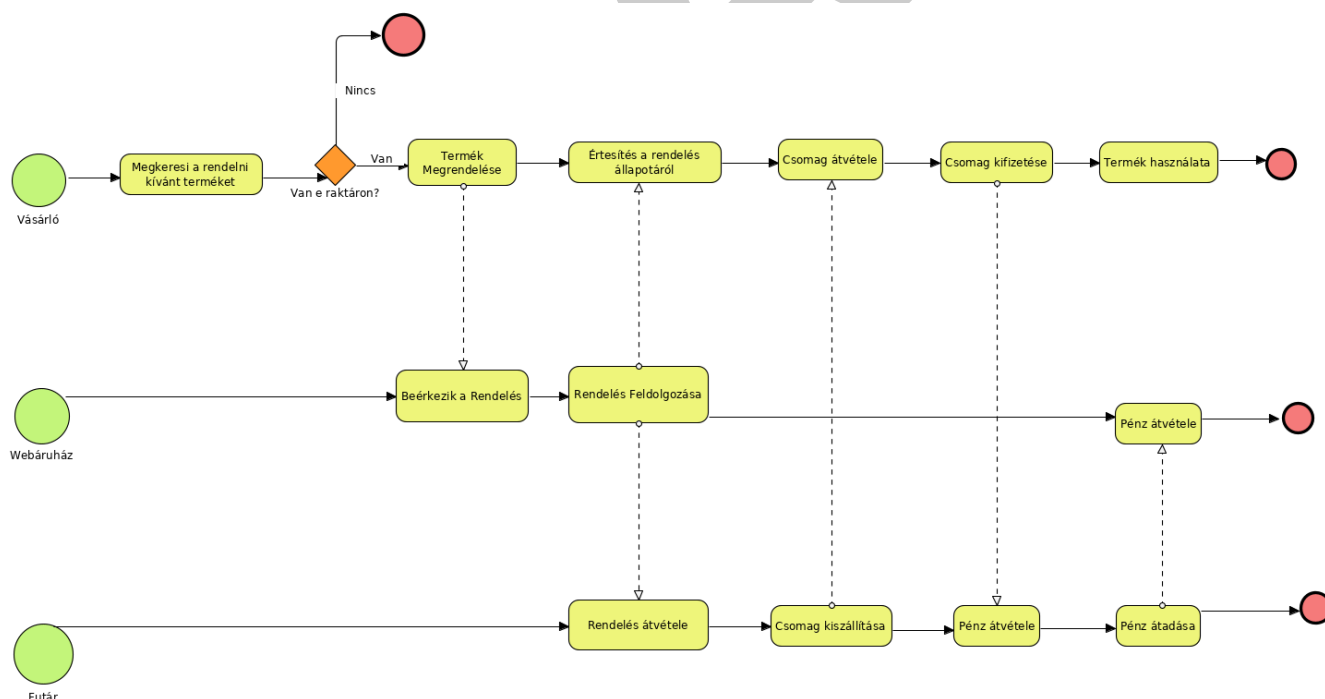
```
Megnyitás ▾ computerproducts_out.txt
~/prog2/src/prog2/Mandelbrot/esettan

d TFT1 30000 20011002 12 13
d TFT2 35000 20061001 10 10
c ComputerConfig1 70000 20061001 2
d TFT3 30000 20011002 12 13
h WesternDigital 35000 20061001 7000
h Maxtor 25000 20050301 7000
```

14.4. BPMN

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

Ebben a feladatban BPMN-ben, azaz Business Process Model and Notation használatával kellett modellezni valamit. Ez a feladat nagyon jól bemutat egy modellező nyelvet a gyakorlatban. Maga a BPMN egy folyamatábra, egy grafikai reprezentációja az üzleti folyamatoknak. Az UML-hez hasonlóan szintén egy modellező eszköz. Az én példám egy mindennapi esetet ír le, egy csomag megrendelését egy webáruházból, vagyis, hogy mi történik a között, hogy a vásárló megrendeli, és megkapja a csomagot. Ennek a feladatnak a megoldásához a visual paradigm nevű szoftvert választottam. A folyamatábra a következőképpen néz ki:



Látható a folyamatábrán, hogy három különböző entitás dolgozik a csomagért. Az egyik a vásárló, aki a csomagot rendeli, a második a webáruház, ami a csomagot eladja, és a harmadik pedig egy futárszolgálat, ami pedig házhoz viszi a csomagot. Maga a folyamat rendkívül egyszerű. Először is a vásárló meglátogatja a webáruházat, itt kezdődik a folyamat. Majd megkeresi a csomagot, amit rendelni szeretne. Ezek után jön egy elágazás, mégpedig hogy van e a keresett termék raktáron. Amennyiben nincs, úgy itt véget is ér a

folyamat. Azonban ha van, akkor megrendeli. Itt történik egy interakció a vásárló és a webáruház között. A webáruház megkapja a rendelést, azt feldolgozza és előkészíti a szállításra. Ezek után két interakció is történik. Egyrészt a webáruház átadja a futárszolgálatnak a csomagot, másrészt pedig szól a vásárlónak, hogy át lett adva a csomagja a futárnak. Majd a futárszolgálat kiszállítja a csomagot a vevőnek, amit az átvesz és kifizet. Ezek után a vevő már csak használja a terméket, ezzel az ő folyamata véget ér. A futár pedig a kapott pénzt átadja a webáruháznak, és ezzel mindkettőjük folyamata véget ér. Ez nyilván egy nagyon egyszerű példa, amit lehetett volna sokkal bonyolultabb is, de a BPMN működésének a bemutatására tökéletes.

15. fejezet

Helló, Chomsky!

15.1. Encoding

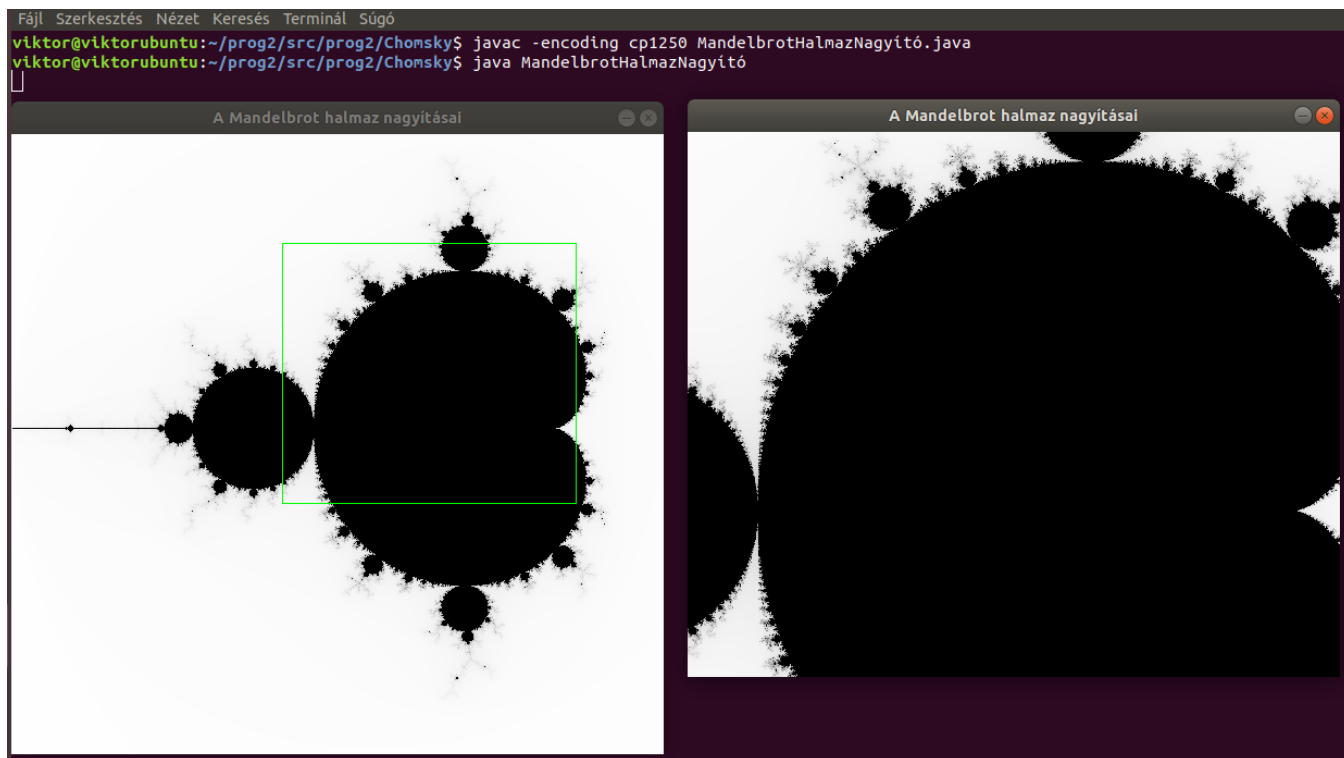
Fordítsuk le és futtassuk a Javat tanítók könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezetes betűket! <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/adatok.html>

Ebben a feladatban a Bátfa Norbert által megadott [MandelbrothalmazNagyító.java](#) forrást kellett futtatni. Ezzel csupán annyi a gond, hogy a forrás tele van ékezetes betűkkel. Éppen ezért, amikor megpróbáljuk lefordítani, akkor a képen látható hibákat kapjuk:

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Chomsky$ javac MandelbrotHalmazNagyító.java
MandelbrotHalmazNagyító.java:2: error: unmappable character (0xED) for encoding UTF-8
 * MandelbrotHalmazNagyító.java
    ^
MandelbrotHalmazNagyító.java:2: error: unmappable character (0xF3) for encoding UTF-8
 * MandelbrotHalmazNagyító.java
    ^
MandelbrotHalmazNagyító.java:4: error: unmappable character (0xED) for encoding UTF-8
 * DIGIT 2005, Javat tanítók
    ^
MandelbrotHalmazNagyító.java:5: error: unmappable character (0xE1) for encoding UTF-8
 * Bátfa Norbert, nbatfa@inf.unideb.hu
    ^
MandelbrotHalmazNagyító.java:9: error: unmappable character (0xED) for encoding UTF-8
 * A Mandelbrot halmazt nagyító és kirajzó osztály.
    ^
MandelbrotHalmazNagyító.java:9: error: unmappable character (0xF3) for encoding UTF-8
 * A Mandelbrot halmazt nagyító és kirajzó osztály.
    ^
MandelbrotHalmazNagyító.java:9: error: unmappable character (0xE9) for encoding UTF-8
 * A Mandelbrot halmazt nagyító és kirajzó osztály.
    ^
MandelbrotHalmazNagyító.java:9: error: unmappable character (0xF3) for encoding UTF-8
 * A Mandelbrot halmazt nagyító és kirajzó osztály.
    ^
MandelbrotHalmazNagyító.java:9: error: unmappable character (0xE1) for encoding UTF-8
 * A Mandelbrot halmazt nagyító és kirajzó osztály.
    ^
MandelbrotHalmazNagyító.java:11: error: unmappable character (0xE1) for encoding UTF-8
 * @author Bátfa Norbert, nbatfa@inf.unideb.hu
    ^
MandelbrotHalmazNagyító.java:14: error: unmappable character (0xED) for encoding UTF-8
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {
    ^
MandelbrotHalmazNagyító.java:14: error: unmappable character (0xF3) for encoding UTF-8
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {
    ^
```

Ez a feladat nagyon jó gyakorlati tudást ad a java karakterkészletekhez. Mint nagyon sokszor, a fordító most is a barátunk: "Unmappable character for encoding UTF-8". Vagyis a forrás kódolásával van a gond. Vagyis ezek a karakterek nem találhatók meg az UTF-8 kódolásban. Ez azt jelenti, hogy egy másik kódolásra kell

átállítani a forrást, amit a -encoding kapcsolóval lehet bállítani. Mostmár csak arra kellett rájönni, hogy mire kellene átállítani a kódolást. Ehhez megkerestem a [Java által támogatott karakterkódolásokat](#). Itt amire felkaptam a fejem, az a windows-1250 kódolás, aminek a leírása az, hogy a Windows Kelet Európai karakterkódolása, és arra gondolva, hogy vagy jó vagy nem, kipróbáltam hogy működik-e, és működött.



15.2. Leetspeak

Lexelj össze egy l33t ciphert!

A program első szegmensében, azon belül is a l337d1c7 struktúrában megadjuk a saját leet-speech szótárunkat. Itt az angol ABC összes betűjére és a 0-9 számjegyekre hasonló speciális karakterekkel ábrázoljuk őket. Minden karaktert négyféleképpen adhatunk meg:

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

struct cipher {
    char c;
    char *leet[4];
} 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', {"l", "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 program következő szegmensében a kapott input minden karakterére megpróbálunk egy új helyettesítő-karaktert találni az általunk definiált szótár segítségével:

```

{
    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);
}
```

Ha megtaláljuk a karaktert a szótárunkban egy random számot generálunk 1-től 100-ig. Ez a szám dönt arról, hogy melyik karakterrel helyettesítjük a megadott 4-ből. (90% esély az elsőre, 4% a másodikra, 3-3% a harmadik és negyedikre)

Majd ezt követi a formázott szöveg kiírása, a Ctrl+D-vel megszakíthatjuk az inputot.

A program első szegmensében, azon belül is a l337d1c7 struktúrában megadjuk a saját leet-speech szótárunkat. Itt az angol ABC összes betűjére a 0-9 számjegyekre hasonlító speciális karakterekkel ábrázoljuk őket. Minden karaktert négyféleképpen adhatunk meg.

```
4 pr0gr4m 3|_ső sz3gm3nséb3n, 4z0|| b3lül 1s 4 lEETdIcT struktúráb4n
  m3g4djuk 4 s4ját l33t-sp33c[-] szótárunk4t. 1tt 4z 4ng0l @bc össz3s
  b3tűjér3 4 D-g számj3gy3kr3 h4$0nlító sp3clálls k4r4kt3r3kk3l
  áb12áz0ljuk ök37. m1nd3n k4r4kt3|27 négyfél3képp3n 4dh4tunk m3|.
```

15.3. Full screen

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

Megoldás videó:

Ebben a feladatban egy teljes képernyős java programot kellett írni, amihez én egy korábbi projektet használtam fel. Mivel maga a program elég hosszú, ezért csak azokra a részekre koncentrálnék, ami a teljes képernyő kialakulásában szerepet játszik. Ez azt jelenti, hogy az egyjátékos() függvényre fogok koncentrálni, mivel a program többjátékos része nem használ grafikus felületet. De az egyjátékos részben már rögtön az első sor is fontos, hiszen létrehozunk egy GraphicsDrive-ot, ami segíteni fog a teljes képernyőre váltásban. Ezek után pedig létrehozunk egy KeyListener-t, ami az ESC gomb lenyomására fog fülelni, hiszen amennyiben a felhasználó lenyomja azt a billentyűt, akkor a program kilép.

```
GraphicsDevice gd = GraphicsEnvironment.getLocalGraphicsEnvironment().
    getDefaultScreenDevice();

KeyListener listener = new KeyListener() {

    @Override
```

```
public void keyPressed(KeyEvent event){

    if(event.getKeyCode() == KeyEvent.VK_ESCAPE)
        System.exit(0);
}

@Override
public void keyReleased(KeyEvent event){}

@Override
public void keyTyped(KeyEvent event){}
};
```

Ezek után létrehozunk egy JFrame-t, ami tulajdonképpen az alkalmazás ablaka, és hozzá is adjuk a frame-hez a KeyListener-t, majd pedig létrehozunk pár gombot, illetve szövegmezőt is, és azokhoz is hozzárendeljük a KeyListener-t. Erre azért van szükség, mivel csak akkor fog működni a KeyListener-ünk, ha hozzá van rendelve ahhoz az elemhez, ami éppen fókuszban van, és nem elég csak a Frame-hez hozzárendelni. Vagyis ha a program épp egy gombra kattintásra vár, de ahhoz a gombhoz nincs hozzárendelve a KeyListener, akkor nem fog bezárulni az alkalmazás, hiába nyomogatjuk az ESC gombot (igen ezt tapasztalatból mondom sajnos).

```
JFrame options = new JFrame("");
options.setTitle("Egyjátékos mód");
options.addKeyListener(listener);
options.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

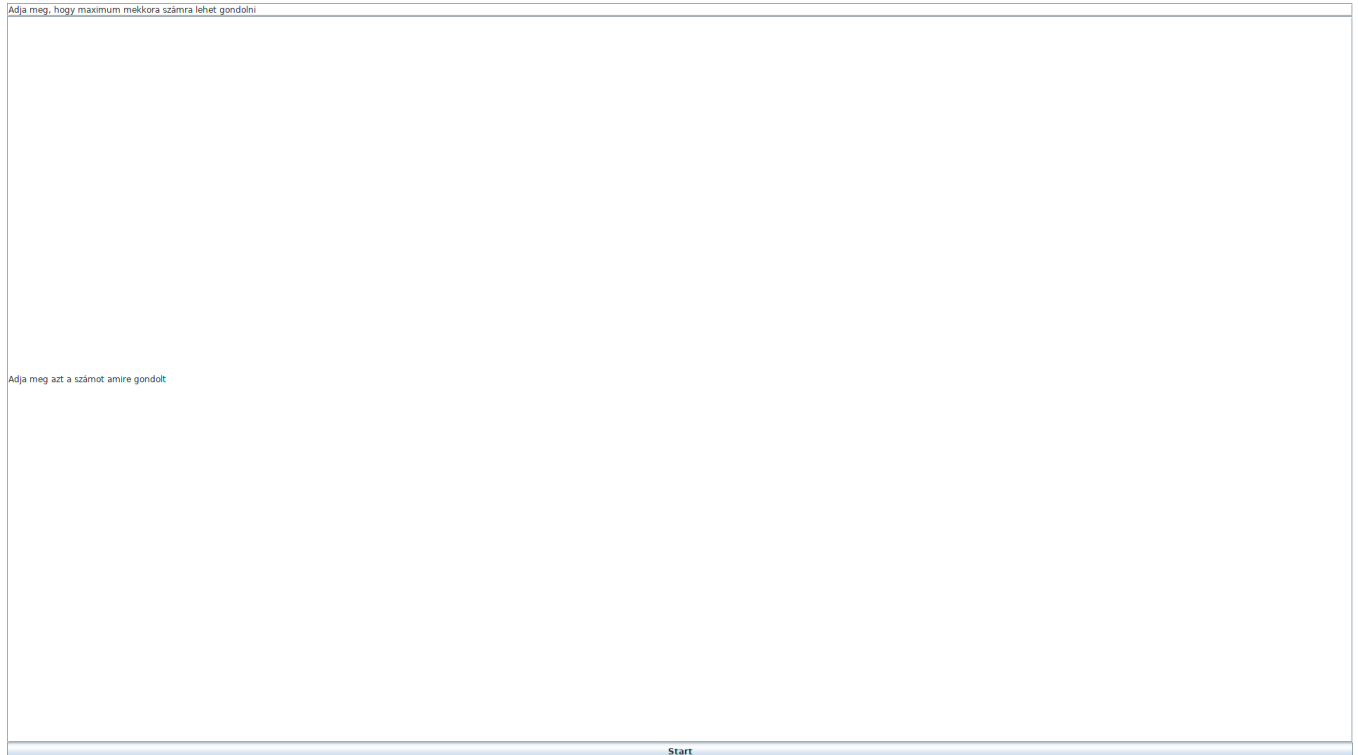
JTextField maxszam = new JTextField("Adja meg, hogy maximum mekkora ←
    számra lehet gondolni");
JTextField gondolt = new JTextField("Adja meg azt a számot amire ←
    gondolt");
JButton start = new JButton("Start");
JButton ujra = new JButton("Újra");
options.getContentPane().add(BorderLayout.NORTH,maxszam);
options.getContentPane().add(BorderLayout.CENTER,gondolt);
options.getContentPane().add(BorderLayout.SOUTH,start);
maxszam.addKeyListener(listener);
gondolt.addKeyListener(listener);
```

Majd pedig a teljes képernyőre váltás, ami úgy történik, hogy megnézzük, hogy támogatja-e a számítógépünk a teljes képernyőt a `gd.isFullScreenSupported()` függvénnyel. Ha igen, akkor `undecorated`-re állítjuk a frame-t, ami szükséges az igazi teljes képernyőhöz, majd pedig a `setFullScreenWindow()` függvénnyel teljes képernyőre állítjuk a frame-t.

```
if (gd.isFullScreenSupported()) {
    options.setUndecorated(true);
    gd.setFullScreenWindow(options);
}
else{
    System.err.println("Nem jó");
    options.setSize(600, 200);
}
```

```
options.setVisible(true);  
}
```

Az eredmény pedig egy teljes képernyős alkalmazás:



15.4. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Lásd vis_prel_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, textúrázás, a szintek jobb elkülönítése, kézreállóbb irányítás.

Megoldás videó:

Ebben a feladatban a Bátfai Norbert által [megadott](#) programon kellett apró változtatásokat elvégezni. Ehhez először fel kellett telepíteni a boost-ot, amit a következő paranccsal lehet megtenni: `sudo apt-get install libboost-all-dev`, illetve e mellett még az OpenGL-re is szükség van, amit pedig a következő parancs kiadásával lehet megtenni: `sudo apt-get install libglu1-mesa-dev freeglut3-dev mesa-common-dev` majd pedig a forráskódban a kommentekben megadott módon lehet fordítani és futtatni.

Az egyik dolog, amin én módosítottam, az a színvilág, amihez a `glColor3f()` függvényt kellett használni. Ennek a függvénynek három paramétere van, az első a piros, a második a zöld, a harmadik pedig a kék szín intenzitását állítja be. Ez azt jelenti, hogy a 0.1 0.0 0.0 értékek egy sötét piros színt adnának, a 0.8 0.0 0.0 értékek pedig egy intenzív világos piros színt eredményeznek. Én egy minimalista stílussal dolgoztam, ezért a világos és sötét szürke színekkel játszadoztam, ahogy az az alábbi képen is látszani fog.

A másik dolog, amin változtattam, az az irányítás. Eddig a kockákat a billentyűzetten található nyilakkal, illetve a page up és page down gombokkal lehetett forgatni. Azonban, mivel nagyítani pedig a + és - gombokkal lehet, ezért nekem a forgatás a W,A,S,D,Q,E billentyűkkel jobban kézreáll. Ehhez töröltem az `skeyboard()` függvényt, és átírtam a `keyboard()` függvényt, ami mostmár a következőképpen néz ki:

```
void keyboard ( unsigned char key, int x, int y )
{
    if ( key == '0' ) {
        index=0;
    } else if ( key == '1' ) {
        index=1;
    } else if ( key == '2' ) {
        index=2;
    } else if ( key == '3' ) {
        index=3;
    } else if ( key == '4' ) {
        index=4;
    } else if ( key == '5' ) {
        index=5;
    } else if ( key == '6' ) {
        index=6;
    } else if ( key == 'f' ) {
        transp = !transp;
    } else if ( key == '-' ) {
        ++fovy;

        glMatrixMode ( GL_PROJECTION );
        glLoadIdentity();
        gluPerspective ( fovy, ( float ) w/ ( float ) h, .1f, ←
            1000.0f );
        glMatrixMode ( GL_MODELVIEW );

    } else if ( key == '+' ) {
        --fovy;

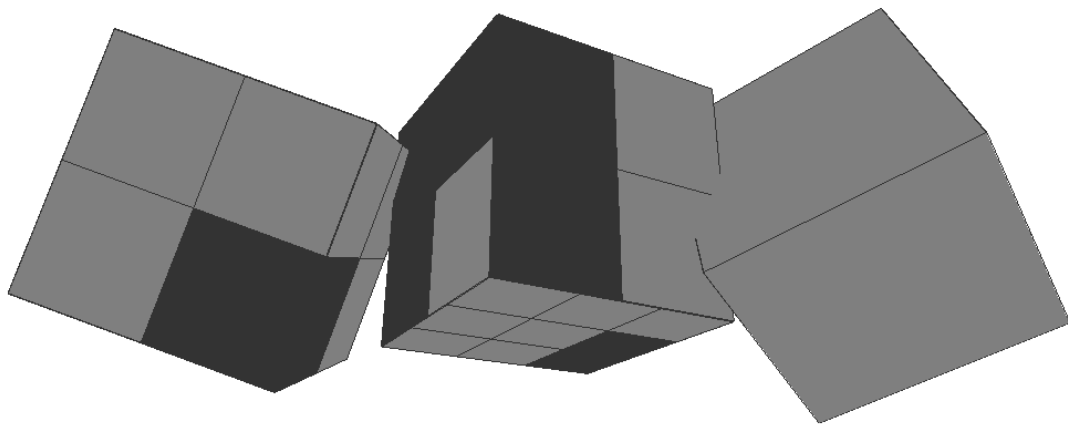
        glMatrixMode ( GL_PROJECTION );
        glLoadIdentity();
        gluPerspective ( fovy, ( float ) w/ ( float ) h, .1f, ←
            1000.0f );
        glMatrixMode ( GL_MODELVIEW );

    }

    else if ( key == 'w' ) {
        cubeLetters[index].rotx += 5.0;
    } else if ( key == 's' ) {
        cubeLetters[index].rotx -= 5.0;
    } else if ( key == 'd' ) {
        cubeLetters[index].roty -= 5.0;
    } else if ( key == 'a' ) {
        cubeLetters[index].roty += 5.0;
    } else if ( key == 'q' ) {
        cubeLetters[index].rotz += 5.0;
    } else if ( key == 'e' ) {
```

```
        cubeLetters[index].rotz -= 5.0;  
    }  
  
    glutPostRedisplay();  
  
}
```

Azt, hogy mostmár más gombokkal kell forgatni a kockákat, ugyan nem tudom megmutatni, de az új színeket viszon igen, amik a következőképpen néznek ki:



16. fejezet

Helló, Stroustrup!

16.1. 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)!

```
#include <iostream>
#include <string>
#include <boost/filesystem.hpp>

using namespace boost::filesystem;
using namespace std;

class Reader
{
private:
    int numberOfClasses;

public:
    Reader()
    {
        numberOfClasses = 0;
    }

    void readClasses(path path)
    {
        if (is_regular_file(path))
        {
            string ext(".java");
            if (!ext.compare(extension(path)))
            {
                cout << path;
                numberOfClasses++;
            }
        }
    }
}
```

```
else if (is_directory(path))
for (directory_entry &entry : directory_iterator(path))
{
    readClasses(entry.path());
}
}
int getNumberOfClasses()
{
    return numberOfClasses;
}
};

int main()
{
    Reader reader;
    reader.readClasses("src");
    cout << "Number of classes:" << reader.getNumberOfClasses() << "\n";

    return 0;
}
```

A feladat megoldásához használjuk a boost könyvtárat. A program végigmeny a src mappában található összes fájlban és mappában, keresve a .java kiterjesztésű fájlokat. A Reader osztályban található readClasses metódusnak átadjuk az src mappa nevét, amiben dolgozik majd. A path egy file-t fog jelenteni és ellenőrzi, hogy ha file és nem könyvtár és .java kiterjesztéssel rendelkezik, akkor kiiratjuk és megnöveljük a számlálót. Viszont ha a path egy könyvtár, akkor rekurzívan megyünk végig az egész könyvtárszerkezeten. A main-ben létrehozuk a Reader objektumot, a konstruktorja beállítja a osztályokat számoló változót 0-ra, majd meghívjuk a readClasses függvényt átadva a mappa nevét majd kiiratjuk a talált .java kiterjesztésű fájlok számát JDK13 esetén. A fordításkor fontos linkelni a boost könyvtárakat.

```
g++ -o jdkreader jdkreader.cpp -lboost_filesystem -lboost_system
```

```
Fájl Szerkesztés Nézet Választás Terminál Ségítség
java/jdk.internal.le/jdk/internal/org/jline/utils/InputStreamReader.java
java/jdk.internal.le/jdk/internal/org/jline/utils/AnsiWriter.java
java/jdk.internal.le/jdk/internal/org/jline/utils/NonBlockingReaderImpl.java
java/jdk.internal.le/jdk/internal/org/jline/utils/Levenshtein.java
java/jdk.internal.le/jdk/internal/org/jline/utils/AttributedStringBuilder.java
java/jdk.internal.le/jdk/internal/org/jline/utils/Signals.java
java/jdk.internal.le/jdk/internal/org/jline/utils/ClosedException.java
java/jdk.internal.le/jdk/internal/org/jline/utils/NonBlockingPumpReader.java
java/jdk.internal.le/jdk/internal/org/jline/utils/StyleResolver.java
java/jdk.internal.le/jdk/internal/org/jline/utils/Display.java
java/jdk.internal.le/jdk/internal/org/jline/utils/InfoCmp.java
java/jdk.internal.le/jdk/internal/org/jline/utils/AttributedString.java
java/jdk.internal.le/jdk/internal/org/jline/utils/NonBlockingReader.java
java/jdk.internal.le/jdk/internal/org/jline/utils/WCwidth.java
java/jdk.internal.le/jdk/internal/org/jline/utils/ExecHelper.java
java/jdk.internal.le/jdk/internal/org/jline/utils/Colors.java
java/jdk.internal.le/jdk/internal/org/jline/utils/Log.java
java/jdk.internal.le/jdk/internal/org/jline/utils/WriterOutputStream.java
java/jdk.internal.le/jdk/internal/org/jline/utils/PumpReader.java
java/jdk.internal.le/jdk/internal/org/jline/utils/OSUtils.java
java/jdk.internal.le/jdk/internal/org/jline/utils/DiffHelper.java
java/jdk.internal.le/jdk/internal/org/jline/utils/ShutdownHooks.java
java/jdk.internal.le/jdk/internal/org/jline/utils/package-info.java
java/jdk.internal.le/jdk/internal/org/jline/utils/Status.java
java/jdk.internal.le/jdk/internal/org/jline/utils/NonBlockingPumpInputStream.java
java/jdk.internal.le/jdk/internal/org/jline/utils/NonBlockingInputStreamImpl.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/Size.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/Attributes.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/TerminalBuilder.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/Cursor.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/spi/JansiSupport.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/spi/Pty.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/spi/JnaSupport.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/AbstractPosixTerminal.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/ExecPty.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/PosixPtyTerminal.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/MouseSupport.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/PosixSysTerminal.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/AbstractPty.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/AbstractWindowsTerminal.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/ExternalTerminal.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/LineDisciplineTerminal.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/NativeSignalHandler.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/DumbTerminal.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/AbstractTerminal.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/AbstractWindowsConsoleWriter.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/package-info.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/impl/CursorSupport.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/MouseEvent.java
java/jdk.internal.le/jdk/internal/org/jline/terminal/Terminal.java
java/jdk.internal.le/jdk/internal/org/jline/keymap/KeyMap.java
java/jdk.internal.le/jdk/internal/org/jline/keymap/BindingReader.java
java/jdk.internal.le/module-info.java
A JDK osztályainak a száma: 18332
viktor@viktorubuntu:~/str5
```

16.2. Másoló-mozgató szemantika + Összefoglaló

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: <https://www.youtube.com/watch?v=iKu9fHrϱLS0>

Másoló konstruktor akkor kerül meghívásra C++ esetén, ha inicializálatlan objektumhoz értéket rendelünk. Gyakran van szükség több példányra ugyanabból az objektumból, ekkor szükséges a másoló konstruktor.

Ha nem adunk meg saját másoló konstruktort, akkor automatikusan keletkezik egy, ami csak egyszerűen lemásolja a nem statikus adattagokat, ezt hívjuk shallow copynak. Ha az adott osztályunk mutatókat vagy referenciákat tartalmaz, ez az alapértelmezett másoló konstruktor nem megfelelő, mert nem a mutatókat, hanem a mutatót magát másolja át. Továbbá a másoló konstruktor paramétere az osztály egy objektumára vonatkozó referencia kell, hogy legyen, különben végtelen ciklus keletkezik, mert a paraméterátadáshoz is a másoló konstruktort használjuk. A másoló konstruktor hívódik meg az alábbi esetekben: változó kezdeti értékének beállításakor, nem referencia függvényparaméter átadásakor, függvény nem referenciával visszatérésekor és kivétel dobásánál és elkapásánál. A másoló konstruktor és az értékadó operátor lehet private elérésű, ekkor az osztály objektumai nem hozhatóak létre és nem másolhatóak, de az osztály publikus tagfüggvényei így is elérik őket. A másoló értékadás pedig akkor kerül meghívásra, ha már egy létező objektumnak akarunk egy olyan új értéket adni, ami egy már létező másik objektum.

Példaként a binfa másoló konstruktora, ami meghívásra kerül, amikor egy új binfa objektumot hozunk létre a régeből, ami egy másolata az eredetinek. A másoló konstruktorba referenciaként kapjuk a régi, lemásolandó objektumot. Ezután a régi fa minden csomópontjának értékén végig megyünk, hogy beállítsuk az új fa csomópontjait, ez a másol függvényben kerül sorra. Pontosan ugyanaz a fa lesz kiépítve az új binfába, mint a régibe ami úgy történik, hogy egyszerűen megyünk végig a régi fa csomópontjain és közben állítjuk be az új fa értékeit, a régi fa értékeivel. Az ujelem mutató mutat az aktuálisan lemásolt csomópontra, ami ha egyenlő az aktuális régi fa csomópontjával akkor beállítjuk az új fa aktuális csomópontjának. Másoló értékadás esetén (operator=) is hasonló dolog történik. Mielőtt a másolás megtörténik, először felszabadítjuk az eredeti fát, majd létrehozuk a gyokeret és az új gyokerbe masoljuk át a másolandó, régi fa elemeit.

```
//Másoló konstruktor
LZWBinFa ( const LZWBinFa & regi ) {
    std::cout << "LZWBinFa copy ctor" << std::endl;

    gyoker = masol(regi.gyoker,regi.fa);
}
Csomopont * masol ( Csomopont * elem, Csomopont * regifa ) {

    Csomopont * ujelem = NULL;

    if ( elem != NULL ) {
        ujelem = new Csomopont ( elem->getBetu() );

        ujelem->ujEgyesGyermeke ( masol ( elem->egyesGyermeke (), regifa ) );
        ujelem->ujNullasGyermeke ( masol ( elem->nullasGyermeke (), regifa ) );

        if ( regifa == elem )
            fa = ujelem;

    }

    return ujelem;
}
//Másoló értékadás
LZWBinFa &operator=(LZWBinFa &regi)
{
```

```
std::cout << "LZWBinFa copy assign" << std::endl;
szabadit(gyoker);
gyoker = new Csomopont();
gyoker = masol(regi.gyoker, regi.fa);
return *this;
}
```

Az első esetben egy új objektumot hozunk létre egy régiből, ezért meghívásra kerül a másoló konstruktor. A második esetben pedig másoló értékadás kerül meghívásra mivel egy létező objektumnak adunk egy létező objektum értékét. A harmadik esetben újra másoló konstruktor hívódik meg, mivel egy vektorba másolunk be egy objektumot.

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

LZWBinFa binFa;
//masolo konstruktor meghivodik
LZWBinFa binFa2 = binFa;
LZWBinFa binFa4 = binFa;
//masolo ertekadas meghivodik
binFa2 = binFa4;
//masolo konstruktor meghivodik
std::vector<LZWBinFa> b;
b.push_back(binFa);
```

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Stroustrup$ g++ -o mozgat mozgat.cpp
viktor@viktorubuntu:~/prog2/src/prog2/Stroustrup$ ./mozgat
Lista lista; //ctor
lista memóriacímek kiírása:
5      0x55ac43f7c2a0
3      0x55ac43f7c2c0
7      0x55ac43f7c2e0

Lista lista2(lista); //copy ctor
lista2 memóriacímek kiírása:
5      0x55ac43f7c320
3      0x55ac43f7c340
7      0x55ac43f7c360

Lista lista3; //ctor
lista3=lista2; //copy assign
lista3 memóriacímek kiírása:
5      0x55ac43f7c3c0
3      0x55ac43f7c3e0
7      0x55ac43f7c400

Lista lista4=std::move(lista3); //move ctor
lista4 memóriacímek kiírása:
5      0x55ac43f7c3c0
3      0x55ac43f7c3e0
7      0x55ac43f7c400

Lista lista5; //ctor
lista5 = std::move(lista4); //move assign
lista5 memóriacímek kiírása:
5      0x55ac43f7c3c0
3      0x55ac43f7c3e0
7      0x55ac43f7c400
viktor@viktorubuntu:~/prog2/src/prog2/Stroustrup$
```

A C++11 verzió egyik legfontosabb újítása a mozgató szemantika bevezetése: a mozgató értékadás és mozgató konstruktort. A másoló értékadás és konstruktor esetében egy objektum tartalmát csak átmásolja egy másik objektumba, azonban a másolás erőforrás igényes, mert a másolás idejkor egyszerre kell két objektumnak is memóriát foglalni és csak utána lehet felszabadítani a régit. Ezért ahol lehet és nincs szükség a másolandó adat megőrzésére, akkor kerülni kell a másolást, mivel a mozgató konstruktor és értékadás egy sokkal hatékonyabb megoldás mint a másolás. Ebben az esetben a régi objektum átadja minden erőforrását az újnak, ezáltal a régi, forrás objektumnak nem lesz tartalma, egy üres objektum lesz.

Példaként a binfa mozgató konstruktora és értékadása látható. A mozgató konstruktorban paraméterként kapott régi, mozgatandó objektumot megkapjuk, majd a jelenlegi új objektumnak a gyökér mutatóját nullra állítjuk. Ezután `*this = std::move(regi)` kényszeríti ki a mozgató értékadást, ahol pedig felcseréljük a régi és az új fa gyökerének mutatóit. Mivel kezdetben az új fa gyökér mutatóját nullra lett állítva, így a régi fa gyökere most nullra mutat, az új pedig a régi fa gyökerére. Ha nem tettük volna meg a null pointerre való állítást, akkor a régi fa destruktora került volna meghívásra és törölte volna a mozgatandó erőforrásokat, így nem történt volna meg a mozgatás.

Fontos megjegyezni, hogy a mozgató szemantikában egy nem konstans jobb érték referenciát kap paraméterként, (dupla referencia jel) míg a másolás esetén egy konstans bal érték referencia kerül paraméterként átadásra. Ezáltal a másolás esetén továbbra is használható lesz a másolt objektum, mozgatás esetén viszont csak ideiglenes szerepet kap a mozgatandó objektum és nem lesz utána már használható.

```
//Mozgató konstruktor
LZWBinFa(LZWBinFa &&regi)
```

```
{
std::cout << "LZWBinFa move ctor" << std::endl;

gyoker = nullptr;
*this = std::move(regi);
}
//Mozgató értékadás
LZWBinFa &operator=(LZWBinFa &&regi)
{

std::swap(gyoker, regi.gyoker);
return *this;
}
```

A main függvényben három alkalommal is meghívódik a mozgató konstruktor és vele együtt a mozgató értékadás. Az első esetben a két különálló fa objektumra meghívjuk a swap metódust, ami megcseréli a két objektum erőforrásait, ami mozgatással történik a swapon belül. A második esetben egy vectorba tesszük bele az objektumot, ami ezután már nem lesz használható, mivel std::move(binFa) jobb érték referenciaként lett átadva. Ha v.push_back(binFa) bal érték referenciaként került volna átadásra akkor a másoló konstruktor kerül meghívásra és megmarad az eredeti objektum erőforrása is. A harmadik lehetőség, amikor pedig egy teljesen új objektumba mozgatjuk át a binFa2 erőforrásait.

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

LZWBinFa binFa;
//masolo konstruktor meghivodik
LZWBinFa binFa2 = binFa;
LZWBinFa binFa4 = binFa;
//masolo ertekadas meghivodik
binFa2 = binFa4;
//masolo konstruktor meghivodik
std::vector<LZWBinFa> t;
t.push_back(binFa);

//mozgato konstruktor meghivodik
std::cout << "Swap" << std::endl;
std::swap(binFa2, binFa4);

//mozgato konstruktor meghivodik
std::cout << "Vector" << std::endl;
std::vector<LZWBinFa> v;
v.push_back(std::move(binFa));

//mozgato konstruktor meghivodik
```

```
std::cout << "Új fa" << std::endl;  
LZWBinFa binFa3 = std::move(binFa2);
```

16.3. Hibásan implementált RSA törése

Készítsünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: <https://arato.inf.unideb.hu/batfai.r> (71-73 fólia) által készített titkos szövegen.

Ebben a feladatban egy hibásan implementált RSA titkosítás törését kellett végrehajtani. Kezdjük is a feladat titkosítási részével. Először is ellenőrizzük, hogy kettő e az argumentumok száma, nyilván az első argumentum lesz a szöveg amit titkosítani szeretnénk, a második argumentum pedig az a fájl, amibe kiírjuk a titkosított szöveget. Amennyiben az argumentumok számával nincs gond, akkor létrehozunk egy KulcsPar objektumot, és egy tisztaszöveg változót, amibe a try blokkon belül be is olvassuk a titkosítandó szöveget, és létrehozunk egy File típust, ami pedig az a fájl lesz, amibe kiírjuk a titkosított szöveget, illetve létrehozunk egy PrintWritert is, amivel pedig majd kiírjuk a szöveget a megfelelő fájlba. Ezek után a tisztaszöveg-et átalakítjuk kisbetűssé, mivel a nagy betűk külön lennének titkosítva, ami megnehezítené a szöveg törését. Ezek után pedig a két for cikluson belül megtörténik a titkosítás, valamint a titkosított szöveg kiírása a megadott fájlba. Ezen kívül van még egy KulcsPar osztálya is a forrásnak, ami pedig a titkosítás matematikai részét végzi.

```
public class Rsa {  
  
    public static void main(String[] args) {  
        if(args.length != 2){  
            System.out.println("usage: java Rsa input output");  
            System.exit(-1);  
        }  
        KulcsPar kulcs = new KulcsPar();  
        String tisztaszoveg;  
  
        try{  
            tisztaszoveg = new String (Files.readAllBytes( Paths.get(args[0])));  
            File ki = new File(args[1]);  
  
            PrintWriter kiir = new PrintWriter(args[1]);  
  
            tisztaszoveg = tisztaszoveg.toLowerCase();  
  
            for( int i = 0; i<tisztaszoveg.length(); i++){  
                String szoveg = tisztaszoveg.substring(i, i+1);  
                byte[] buffer = szoveg.getBytes();  
                java.math.BigInteger[] titkos = new java.math.BigInteger[buffer.length];  
                byte[] output = new byte[buffer.length];  
  
                for( int j = 0; j< titkos.length; j++){
```

```

        titkos[j] = new java.math.BigInteger(new byte[] {buffer[j]});
        titkos[j] = titkos[j].modPow(kulcs.e, kulcs.m);
        output[j] = titkos[j].byteValue();
        kiir.print(titkos[j]);
    }
    kiir.println();
}
}
catch(IOException e){
    System.out.println("hiba " + e);
}
}
}

class KulcsPar {
    java.math.BigInteger d,e,m;
    public KulcsPar() {
        int meretBitekben = 700 * (int) (java.lang.Math.log((double) 10) / java ←
            .lang.Math.log((double) 2));

        java.math.BigInteger p = new java.math.BigInteger(meretBitekben, 100, ←
            new java.util.Random());
        java.math.BigInteger q = new java.math.BigInteger(meretBitekben, 100, ←
            new java.util.Random());

        m = p.multiply(q);
        java.math.BigInteger z = p.subtract(java.math.BigInteger.ONE).multiply( ←
            q.subtract(java.math.BigInteger.ONE));

        do {
            do {
                d = new java.math.BigInteger(meretBitekben, new java.util.Random()) ←
                    ;
            } while (d.equals(java.math.BigInteger.ONE));
        } while (!z.gcd(d).equals(java.math.BigInteger.ONE));
        e = d.modInverse(z);
    }
}

```

A program másik része a titkosított szöveg dekódolása. Ebben a forráskódban van egy KulcsPar osztály, aminek három tagváltozója van. Az első a values, ami az adott karakternek a titkosított értékét tárolja, a második a key, ami azt tárolja, hogy mi az adott dekódolt karakter, a harmadik pedig a freq, ami pedig a karakter előfordulásainak a számát tárolja. Ezen kívül megtalálhatóak még a tagváltozók getterei, illetve setterei, valamint az incFreq() metódus is, ami az előfordulást növeli eggyel.

```

class KulcsPar{
    private String values;
    private char key = '_';
    private int freq = 0;

```

```
public KulcsPar(String str, char k){
    this.values = str;
    this.key = k;
}

public KulcsPar(String str){
    this.values = str;
}

public void setValue(String str){
    this.values = str;
}

public void setKey(char k){
    this.key = k;
}

public String getValue(){
    return this.values;
}

public char getKey(){
    return this.key;
}

public void incFreq(){
    freq += 1;
}

public int getFreq(){
    return freq;
}
}
```

Valamint megtalálható a main is, amiben pedig először is megadjuk a programnak, hogy hol van a titkosított fájl, majd pedig létrehozunk egy lines tömböt, amibe be is olvassuk egy while ciklussal a titkosított fájl sorait. Ezek után létrehozunk egy KulcsPár tömböt, egy volt logikai változót, amiben azt fogjuk tárolni, hogy az adott sor szerepel e már a kulcspár tömbben. Majd pedig két egymásba ágyazott for ciklussal, ha már egy adott sor szerepel a tömbünkben, akkor csak növeljük az előfordulásainak a számát eggyel, ha viszont még nem szerepel a tömbben, akkor példányosítunk egy új kulcspárt a tömbbe, aminek beállítjuk a values tagváltozójának az értékét az adott sor értékére. Ezek után rendezzük a kulcspár tömböt az előfordulások száma alapján csökkenő sorrendbe.

```
public static void main(String[] args) {
    try {
        BufferedReader inputStream = new BufferedReader(new FileReader( ↵
            args[0]));
        int lines = 0;

        String line[] = new String[10000];
```

```
while((line[lines] = inputStream.readLine()) != null) {
    lines++;
}

inputStream.close();

KulcsPar kp[] = new KulcsPar[100];

boolean volt = false;
kp[0] = new KulcsPar(line[0]);
int db = 1;

for(int i = 1; i < lines; i++) {
    volt = false;
    for(int j = 0; j < db; j++) {
        if(kp[j].getValue().equals(line[i])) {
            kp[j].incFreq();
            volt = true;
            break;
        }
    }

    if(volt == false) {
        kp[db] = new KulcsPar(line[i]);
        db++;
    }
}

for(int i = 0; i < db; i++) {
    for(int j = i + 1; j < db; j++) {
        if(kp[i].getFreq() < kp[j].getFreq() ) {
            KulcsPar temp = kp[i];
            kp[i] = kp[j];
            kp[j] = temp;
        }
    }
}
```

Ezek után beolvassuk azt a fájlt, amiben sorrendbe vannak rakva a karakterek gyakoriság alapján. Az én esetemben ez a `betugyakorsag.txt` nevű fájl. Ezeket a karaktereket belehelyezzük egy karakter tömbbe, és egy `while` ciklussal a `KulcsPár` példányoknak a `key` tagváltozóját beállítjuk a megfelelő karakterekre. Végül pedig végigmegyünk a `lines` tömbbön, és az alapján kiíratjuk a `kp` tömbből a megfelelő karaktereket.

16.4. Változó argumentumszámú ctor

Készítsünk olyan példát, amely egy képet tesz az alábbi projekt `Perceptron` osztályának bemenetére és a `Perceptron` ne egy értéket, hanem egy ugyanakkora méretű „képet” adjon vissza. (Lásd még a 4 hét/Per-

ceptron osztály feladatát is.)

A Perceptron egy neurális háló, aminek csomópontjai vannak. A csomópontok a mesterséges neuronok. Két fő csomópont a bemeneti és kimeneti csomópontok. Van egy központi csúcs is, ahol a számolás történik. Az input csúcsokból bejövő kapcsolatok súlyozva vannak, ami alapján történik a számolás a központi csúcsban. A tanulás abból áll, hogy a bejövő kapcsolatok erőssége változik, a sokat használt kapcsolatokban erősödik, a ritkán használtakban gyengül. Egy neurális hálózat esetén pedig több réteg lehet egymás után, amik a perceptronok és minden perceptron az eggyel korábbi szinten lévőkből kapja az inputot és az eggyel későbbi szintre továbbítja.

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

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, size);

    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 *newImage = new double[size];
    *newImage = (*p)(image);
    S
    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 = newImage[i * png_image.get_width() + j];

    png_image.write("newMandel.png");

    delete p;
    delete[] image;
    delete[] newImage;
}
```

Ebben a projektben használjuk a Perceptron osztályt. A mandel program által előállított kép lesz feldolgozva. Először eltároljuk png::image objektumként a paraméterként kapott képet. A size változóban a kép méretét tároljuk el. Ezután létrehozuk a perceptron objektumot, mutatóját megadva a konstruktorának a paramétereit, ami változó paraméter számú. A 3 a neurális háló rétegek számát jelenti, a size a kép mérete, a 256 a neuronok száma az egyes rétegekben, ellentétben az előző perceptron feladatban, most nem 1 hanem 256 kimeneti neuront adunk meg, ezáltal egy ugyanakkora képet kapunk vissza. Lefoglaltuk az új kép méretének, pixeleinek a helyét egy double tömbre mutató mutatóval, ami az eredeti kép összes pixelét

képes tárolni. Ezután két for ciklussal bejárjuk az eredeti képet, pixeleit és az új helyre tesszük az eredeti kép piros komponenseit. Miután megvan a kép, meghívjuk a perceptron osztály () operátorát átadva azt a tömböt amiben az eredeti kép pixelei vannak, ami most nem egy double típusú értéket ad vissza, hanem egy double értékeket tároló tömböt, aminek értékeivel módosítjuk az eredeti kép pixeleit. Két for ciklussal szintén végigjárunk a pixeleken és most az eredeti kép piros komponenseit cseréljük ki azokra az értékekre, amit kaptunk perceptron osztálytól. Végül az elkészült képet kimentjük és felszabadítjuk a memóriát.

DRAFT

17. fejezet

Helló, Gödel!

17.1. 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!

C++ nyelvben az allokátoroknak elég sokatmondó neve van, mivel pontosan az a feladatuk, hogy memóriát allokálljanak az adatszerkezeteknek. Erre van egy beépített `std::allocator` allokátor, de írhatunk sajátot is. Ez a program pont egy saját allokátor működését mutatja be, de lássuk is, hogy hogyan. Kezdjük a CustomAlloc osztállyal, ami az allokálást végzi, illetve minden alkalommal amikor memóriát foglal, valamint tájékoztatja a felhasználót arról, hogy hány darab objektumnak, milyen méretű memóriát allokal, valamint arról is tájékoztat, hogy milyen típusú változónak allokalja az adott memóriát. Azt, hogy az allokátor több különböző fajta típussal is tud dolgozni, a

```
template<typename T>
```

-nek köszönhetjük. A deallocate metódus pedig felszabadítja a lefoglalt helyet.

```
#include <iostream>
#include <cxxabi.h>
#include <vector>

template<typename T>
struct CustomAlloc
{
    using size_type      = size_t;
    using value_type     = T;
    using pointer        = T*;
    using const_pointer  = const T*;
    using reference      = T&;
    using const_reference = const T&;
    using difference_type = ptrdiff_t;

    CustomAlloc() {}
    CustomAlloc ( const CustomAlloc & ) {}
    ~CustomAlloc() {}
};
```

```
pointer allocate(size_type n) {
    int s;
    char *p = abi::__cxa_demangle(typeid(T).name(), 0, 0, &s);
    std::cout << "Allocating "
               << n << " object(s) of "
               << n * sizeof(T)
               << " bytes. "
               << typeid(T).name() << "=" << p
               << std::endl;
    free(p);
    return reinterpret_cast<T*> (new char[n*sizeof(T)]);
}

void deallocate(pointer p, size_type n) {
    std::cout << "Deallocate " << n*sizeof(T) << " bytes." << std::endl;
    delete[] reinterpret_cast<char *> (p);
}

};
```

Ezek után jön a main, amiben a CustomAllokátorunk segítségével helyet foglalunk először is egy intnek, aminek a 3 értéket adjuk, majd pedig egy long-ot, aminek a 3213125211 értéket adjuk, és végül pedig egy Stringet, aminek pedig az "a" értéket adjuk. Ez az alábbi futás közbeni képen is jól látszik. Köszönhetően a nyomkövetőknek a program a tudunkra adja, hogy először is allokalált egy 4 bájt méretű int típusú objektumnak helyet, majd pedig egy 8 bájt méretű long objektumnak, végül pedig egy 32 bájt méretű String objektumnak is. Majd pedig a deallokalált először 32, majd 8, végül pedig 4 bájtot.

```
int main(int argc, char *argv[])
{
    std::vector<int, CustomAlloc<int>> ints;
    ints.push_back(3);

    std::vector<long, CustomAlloc<long>> longs;
    longs.push_back(3213125211);

    std::vector<std::string, CustomAlloc<std::string>> strings;
    strings.push_back("a");
    return 0;
}
```

A kép amiről fentebb beszéltem:

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Gödel$ g++ -o CustomAlloc CustomAlloc.cpp
viktor@viktorubuntu:~/prog2/src/prog2/Gödel$ ./CustomAlloc
Allocating 1 object(s) of 4 bytes. i=int
Allocating 1 object(s) of 8 bytes. l=long
Allocating 1 object(s) of 32 bytes. Nst7__cxx112basic_stringIcSt11char_traitsIcESaIcEEE=std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
Deallocate 32 bytes.
Deallocate 8 bytes.
Deallocate 4 bytes.
viktor@viktorubuntu:~/prog2/src/prog2/Gödel$
```

17.2. STL map érték szerinti rendezése

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

Ebben a feladatban kulcs-érték párokat kellett rendezni. Alapvetően ezt kulcs alapján tennénk meg, viszont itt érték alapján kellett. Lássuk hogyan működik a program, kezdve a Map osztállyal:

```
package stlmap;

public class Map
{
    private String kulcs;
    private int ertekek;

    public Map( String[] tomb)
    {
        kulcs = tomb[0];
        ertekek = Integer.parseInt(tomb[1]);
    }

    @Override
    public String toString() {
        return "kulcs=" + kulcs + ", ertekek=" + ertekek;
    }

    public String getKulcs() {
        return kulcs;
    }

    public int getErtek() {
        return ertekek;
    }

    public void setKulcs(String kulcsra){
        kulcs = kulcsra;
    }

    public void setErtek(int ertekre){
        ertekek = ertekre;
    }
}
```

```
}  
  
}
```

Itt látható, hogy a kulcs-érték párok egy osztályban vannak eltárolva, aminek a két tagváltozója a kulcs és érték. Mivel ezek privát tagváltozók, éppen ezért megtalálhatóak a hozzájuk tartozó setterek és getterek is. Ezeken kívül van még egy `toString()` metódus is, ami kiírja az objektumok tagváltozóinak az értékeit. Érdekes még megemlíteni, hogy a konstruktor egy `String` tömböt kap, aminek az első eleme lesz az adott objektum kulcsa, a második eleme pedig az adott objektum értéke. Az, hogy ez miért van így, arra később fogok kitérni. Ez elég egyszerű, ezért térjünk is át magára a beolvasásra, illetve a rendezésre. Az egész úgy kezdődik, hogy megszámoljuk azt, hogy hány darab kulcs érték párunk van a `feladat.txt` nevű fájlban, majd pedig egy `RandomAccessFile` segítségével beolvassuk azokat a `Map` tömbbe:

```
public static void main(String args[] ){  
  
    RandomAccessFile raf;  
    String sor;  
    Map[] tomb;  
    int db;  
  
    try  
    {  
        raf = new RandomAccessFile("stlmap/feladat.txt", "r");  
        db = 0;  
  
        for( sor = raf.readLine(); sor != null; sor = raf.readLine() )  
        {  
            db++;  
        }  
  
        tomb = new Map[db];  
        db = 0;  
        raf.seek(0);  
  
        for( sor = raf.readLine(); sor != null; sor = raf.readLine() )  
        {  
            tomb[db] = new Map(sor.split(", "));  
            db++;  
        }  
        raf.close();  
    }  
}
```

Négy változóval kezdünk. Az első a `RandomAccessFile`, ami a kulcs-érték párokat tartalmazó fájl elérési útját fogja tárolni, egy `String`, ami a beolvasásnál a fájl egy-egy sorát fogja tárolni, egy `Map[]` `tomb`, amibe a kulcs-érték párok kerülnek, illetve egy `int db`, amivel pedig a fájlban lévő sorok számát, illetve a `tomb` hosszát fogja megmondani. Ezek után inicializáljuk a `RandomAccessFile`-t és a `db` változót is, majd pedig egy `for` ciklussal megszámoljuk a sorok számát. Ezek után inicializáljuk a `Map` tömböt, a `db` változó értékét nullára állítjuk, és visszaugrunk a fájl elejére. Ezek után végigmegyünk a fájl sorain, amikben a kulcs-érték párok vesszővel vannak elválasztva egymástól, éppen ezért a konstruktornak egy kételemű tömböt adunk át, ami az adott sor elválasztva a vesszőnél a `sor.split(", ");` függvény segítségével.

Ezek után kiíratjuk az eredeti tömböt egy for each ciklussal, amit aztán shell rendezéssel rendezünk, majd pedig ismét kiírjuk a rendezett értékeket:

```
System.out.println("eredeti értékek: ");

    for( Map i : tomb )
    {
        System.out.println(i.toString());
    }

    for( int gap = db / 2; gap > 0; gap /=2){
        for( int i = gap; i< db; i++){
            Map temp= tomb[i];
            int j;
            for( j = i; j >= gap && tomb[j - gap].getErtek() > temp ↔
                .getErtek(); j -= gap){
                tomb[j] = tomb[j - gap];
            }

            tomb[j] = temp;
        }
    }
    System.out.println("\nRendezett értékek:");
    for( Map i : tomb){
        System.out.println(i.toString());
    }
}
catch(IOException e){
    System.out.println("Hiba a beolvasas soran: "+e);
}
}
```

A program működés közben pedig a következőképpen néz ki:

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
viktor@viktorubuntu:~/prog2/src/prog2/Gödel$ javac stlmap/*.java
viktor@viktorubuntu:~/prog2/src/prog2/Gödel$ java stlmap/Stlmap
eredeti értékek:
kulcs=a, ertekek=13
kulcs=f, ertekek=57
kulcs=d, ertekek=96
kulcs=z, ertekek=1
kulcs=t, ertekek=5
kulcs=c, ertekek=4
kulcs=k, ertekek=24
kulcs=l, ertekek=45
kulcs=m, ertekek=16
kulcs=p, ertekek=90
kulcs=s, ertekek=8
kulcs=h, ertekek=7
kulcs=i, ertekek=99
kulcs=g, ertekek=42
kulcs=x, ertekek=71
kulcs=y, ertekek=57
kulcs=q, ertekek=13

Rendezett értékek:
kulcs=z, ertekek=1
kulcs=c, ertekek=4
kulcs=t, ertekek=5
kulcs=h, ertekek=7
kulcs=s, ertekek=8
kulcs=a, ertekek=13
kulcs=q, ertekek=13
kulcs=m, ertekek=16
kulcs=k, ertekek=24
kulcs=g, ertekek=42
kulcs=l, ertekek=45
kulcs=f, ertekek=57
kulcs=y, ertekek=57
kulcs=x, ertekek=71
kulcs=p, ertekek=90
kulcs=d, ertekek=96
kulcs=i, ertekek=99
viktor@viktorubuntu:~/prog2/src/prog2/Gödel$
```

17.3. Alternatív Tabella rendezése

Mutassuk be a https://progater.blog.hu/2011/03/11/alternativ_tabella a programban a java.lang Interface Comparable

<T>

szerepét!

Ez a feladat nagyon jó gyakorlati tudást ad a kollekciókhoz. Először is ahhoz, hogy a 2017-2018-as bajnokságot mutassuk be, el kell végeznünk pár módosítást a forráskódokban. Először is a Wiki2Matrix.java forrásfájlban -ami előállítja a saját tabellánk előállításához szükséges mátrixot- a táblázat mátrixot át kell írunk a 2017-18-as bajnokság kereszttáblázatára. Ezt elég egyszerű megcsinálni. A 18-as bajnokság kereszttáblázatát nézve, ha egy cella üres, akkor a mi táblázatunkba arra a helyre 0-t, ha a hazai csapat nyert, akkor 1-et, ha a vendég csapat nyert, akkor hármast, ha pedig döntetlen, akkor kettőt írunk. Ha mindent elvégeztünk, akkor valamennyivel kevesebb sora és oszlopa lesz a 18-as bajnokság táblázatának, és a következőképpen fog kinézni:


```
int[][] tablazat= {
    {0, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 3 },
    {3, 0, 3, 2, 1, 2, 2, 1, 1, 1, 1, 3 },
    {1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    {3, 2, 1, 0, 3, 3, 3, 3, 3, 1, 1, 3 },
    {1, 1, 2, 3, 0, 1, 2, 1, 1, 1, 2, 3 },
    {1, 2, 3, 1, 3, 0, 2, 1, 2, 1, 3, 2 },
    {2, 1, 3, 1, 1, 2, 0, 3, 1, 1, 2, 1 },
    {3, 1, 3, 1, 3, 3, 3, 0, 3, 1, 1, 3 },
    {1, 3, 3, 2, 2, 1, 1, 1, 0, 1, 3, 3 },
    {1, 1, 1, 1, 1, 3, 1, 2, 2, 0, 3, 1 },
    {1, 1, 2, 1, 2, 1, 1, 3, 2, 1, 0, 1 },
    {1, 3, 1, 1, 1, 1, 1, 3, 2, 3, 1, 0 }
};
```

Ha ezzel készen vagyunk, akkor le kell futtatni ezt a programot, és a következő kép alján látható mátrixot kell bemásolni az AltTabella L mátrixába. Majd pedig a csapatNevE, az ep, illetve az csapatNevL tömb elemeit is át kell írni. Ha ezeket megcsináljuk, akkor a github linken látható módon fog kinézni a táblázat. Ezt most ide nem másolnám be, mert nagyon hosszú és ronda lenne.

```
Fájl Szerkesztés Nézet Keresés Terminal Sugo
vltor@vltorubuntu:~/prog2/src/prog2/codel/alt$ javac Wiki2Matrix.java
vltor@vltorubuntu:~/prog2/src/prog2/codel/alt$ java Wiki2Matrix
A x pontot szerez y tol matrix!
0, 2, 1, 2, 1, 1, 2, 2, 1, 1, 1, 0,
0, 0, 0, 2, 1, 2, 1, 1, 2, 1, 1, 1,
1, 2, 0, 1, 2, 2, 2, 2, 2, 1, 2, 1,
0, 2, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0,
1, 1, 1, 1, 0, 2, 1, 2, 2, 1, 2, 0,
2, 2, 0, 2, 0, 0, 2, 2, 1, 2, 0, 1,
1, 2, 0, 2, 2, 2, 0, 1, 1, 1, 1, 1,
0, 1, 0, 2, 0, 0, 1, 0, 0, 2, 2, 1,
1, 0, 0, 2, 1, 2, 1, 2, 0, 2, 1, 1,
1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 2,
2, 1, 1, 1, 2, 2, 2, 0, 2, 2, 0, 1,
2, 1, 1, 2, 2, 2, 1, 1, 2, 0, 1, 0, Sor es oszlopösszegekkel!
0, 2, 1, 2, 1, 1, 2, 2, 1, 1, 1, 0, 14
0, 0, 0, 2, 1, 2, 1, 1, 2, 1, 1, 1, 12
1, 2, 0, 1, 2, 2, 2, 2, 2, 1, 2, 1, 18
0, 2, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 7
1, 1, 1, 1, 0, 2, 1, 2, 2, 1, 2, 0, 14
2, 2, 0, 2, 0, 0, 2, 2, 1, 2, 0, 1, 14
1, 2, 0, 2, 2, 2, 0, 1, 1, 1, 1, 1, 14
0, 1, 0, 2, 0, 0, 1, 0, 0, 2, 2, 1, 9
1, 0, 0, 2, 1, 2, 1, 2, 0, 2, 1, 1, 13
1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 2, 10
2, 1, 1, 1, 2, 2, 2, 0, 2, 2, 0, 1, 16
2, 1, 1, 2, 2, 2, 1, 1, 2, 0, 1, 0, 15
11, 15, 6, 18, 13, 15, 14, 14, 15, 14, 12, 9,
A "link" matrix
{0,0, 0.13333333333333333, 0.16666666666666666, 0.11111111111111111, 0.07692307692307693, 0.06666666666666667, 0.14285714285714285, 0.14285714285714285, 0.06666666666666667, 0.07142857142857142, 0.08333333333333333, 0.0, },
{0,0, 0.0, 0.11111111111111111, 0.07692307692307693, 0.13333333333333333, 0.07142857142857142, 0.07142857142857142, 0.13333333333333333, 0.07142857142857142, 0.08333333333333333, 0.11111111111111111, }
{0.09090909090909091, 0.13333333333333333, 0.0, 0.05555555555555555, 0.15384615384615385, 0.13333333333333333, 0.14285714285714285, 0.14285714285714285, 0.13333333333333333, 0.07142857142857142, 0.16666666666666666, 0.11111111111111111, },
{0.0, 0.13333333333333333, 0.16666666666666666, 0.0, 0.07692307692307693, 0.0, 0.0, 0.06666666666666667, 0.07142857142857142, 0.08333333333333333, 0.0, },
{0.09090909090909091, 0.06666666666666667, 0.16666666666666666, 0.05555555555555555, 0.0, 0.13333333333333333, 0.07142857142857142, 0.14285714285714285, 0.13333333333333333, 0.07142857142857142, 0.16666666666666666, 0.0, },
{0.18181818181818182, 0.13333333333333333, 0.0, 0.11111111111111111, 0.0, 0.0, 0.14285714285714285, 0.14285714285714285, 0.06666666666666667, 0.14285714285714285, 0.0, 0.11111111111111111, },
{0.09090909090909091, 0.13333333333333333, 0.0, 0.11111111111111111, 0.15384615384615385, 0.13333333333333333, 0.0, 0.07142857142857142, 0.06666666666666667, 0.07142857142857142, 0.08333333333333333, 0.11111111111111111, },
{0.0, 0.06666666666666667, 0.0, 0.11111111111111111, 0.0, 0.0, 0.14285714285714285, 0.14285714285714285, 0.16666666666666666, 0.11111111111111111, },
{0.09090909090909091, 0.0, 0.0, 0.11111111111111111, 0.07692307692307693, 0.13333333333333333, 0.07142857142857142, 0.14285714285714285, 0.0, 0.14285714285714285, 0.08333333333333333, 0.11111111111111111, },
{0.09090909090909091, 0.06666666666666667, 0.16666666666666666, 0.05555555555555555, 0.07692307692307693, 0.0, 0.07142857142857142, 0.07142857142857142, 0.06666666666666667, 0.0, 0.0, 0.22222222222222222, },
{0.18181818181818182, 0.06666666666666667, 0.16666666666666666, 0.05555555555555555, 0.15384615384615385, 0.13333333333333333, 0.14285714285714285, 0.0, 0.13333333333333333, 0.14285714285714285, 0.0, 0.11111111111111111, },
{0.18181818181818182, 0.06666666666666667, 0.16666666666666666, 0.11111111111111111, 0.15384615384615385, 0.13333333333333333, 0.07142857142857142, 0.07142857142857142, 0.13333333333333333, 0.0, 0.08333333333333333, 0.0, },vltor@vltorubuntu:~/prog2/src/prog2/codel/alt$
```

Most nézzük meg, hogy mit is csinál a Comparable. Ehhez a következő kódcsipetek a lényegesek számunkra:

```
java.util.List<Csapat> rendezettCsapatok = java.util.Arrays.asList( ←
    csapatok);
java.util.Collections.sort(rendezettCsapatok);
java.util.Collections.reverse(rendezettCsapatok);
java.util.Iterator iterv = rendezettCsapatok.iterator();
```

Illetve:

```
class Csapat implements Comparable<Csapat> {  
  
    protected String nev;  
    protected double ertek;  
  
    public Csapat(String nev, double ertek) {  
        this.nev = nev;  
        this.ertek = ertek;  
    }  
  
    public int compareTo(Csapat csapat) {  
        if (this.ertek < csapat.ertek) {  
            return -1;  
        } else if (this.ertek > csapat.ertek) {  
            return 1;  
        } else {  
            return 0;  
        }  
    }  
}
```

Az, hogy mit is jelent a Comparable és a compareTo, azt a hivatalos [dokumentációban](#) találhatjuk meg. E szerint a compareTo függvény visszatérési értéke minden esetben vagy nulla, vagy egy negatív szám, vagy pedig egy pozitív szám attól függően, hogy az az objektum amit hasonlítunk kisebb, nagyobb, vagy egyenlő e azzal az objektummal, amihez hasonlítjuk. A Collections.sort(rendezettCsapatok) is ezt a metódust fogja használni, viszont overrideolnunk kell. Azaz ha a vizsgált objektum értéke kisebb mint a másik, akkor -1-et fog visszaadni, ha nagyobb, akkor +1-et, ha pedig egyenlőek, akkor nullát. Még kérdés lehet az, hogy miért is van szükségünk a Comparable implementálására. A rövid és tömör válasz az az, hogy a Collections.sort() miatt. A hosszabb választ pedig megtaláljuk a hivatalos [dokumentációban](#), ami tisztán kimondja azt, hogy "All elements in the list must implement the Comparable interface." Azaz a lista minden elemének implementálnia kell a Comparable interfészt. A végeredmény pedig:

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
Csapatok rendezve:

| -
| Ferencváros
| 74
| Újpest
| 0.1071
| -
| Mol Vidi FC
| 61
| Ferencváros
| 0.1071
| -
| Debrecen
| 51
| Mol Vidi FC
| 0.0971
| -
| Honvéd
| 49
| Honvéd
| 0.0916
| -
| Újpest
| 48
| Debrecen
| 0.0861
| -
| Mezőkövesd
| 44
| Mezőkövesd
| 0.0834
| -
| Puskás Akadémia
| 40
| Kisvárdá
| 0.0800
| -
| Paks
| 39
| Puskás Akadémia
| 0.0785
| -
| Kisvárdá
| 38
| Paks
| 0.0782
| -
| DVTK
| 38
| DVTK
| 0.0719
| -
| MTK
| 34
| MTK
```

17.4. GIMP Scheme hack

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

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

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

Ez egy prog1-en már tárgyalt feladat, amiben egy gimp kiegészítőről van szó, amiben a bemenő szövegből egy név-mandala fog készülni. A mandala egy szimmetrikus kör alakú kép, ami a Hindu vallásban nagy szerepet játszik a Hindu istenek ábrázolásában. Először a program meghatározza a szöveg hosszát, a kapott értéket a set! utasítással a text-width változó értékének adja. A következő függvény a betűk méretét határozza meg. A szükséges méretek a a GIMP beépített függvényeivel határozza meg a következőképpen. Alapvetően nem feltétlenül szükséges tudnunk a szöveghosszt a mandala előállításához, de ahhoz, hogy szép legyen a kép, ezt is tudnia kell a programnak. Ezt elég könnyű meghatározni, mivel a gimp-text-get-extents-fontname egy listát fog visszaadni, aminek a szöveghossz a legelső eleme, ami nekünk kell. Azt pedig a car függvénnyel határozzuk meg. A car függvény egy lista fejét adja vissza, jelen esetben a lista első elemét, pont ami nekünk kell.

```
(define (elem x lista)

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

)

(define (text-width text font fontsize)
(let*
  (
    (text-width 1)
  )
  (set! text-width (car (gimp-text-get-extents-fontname text fontsize ←
    PIXELS font)))

  text-width
)
)

(define (text-wh text font fontsize)
(let*
  (
    (text-width 1)
    (text-height 1)
  )
  ;;;
  (set! text-width (car (gimp-text-get-extents-fontname text fontsize ←
    PIXELS font)))
  ;;; ved ki a lista 2. elemét
  (set! text-height (elem 2 (gimp-text-get-extents-fontname text ←
    fontsize PIXELS font)))
  ;;;

  (list text-width text-height)
)
)
```

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

(define (script-fu-bhax-mandala text text2 font fontsize width height color ←
  gradient)
  (let*
    (
      (image (car (gimp-image-new width height 0)))
      (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" 100 ←
        LAYER-MODE-NORMAL-LEGACY)))
      (textfs)
      (text-layer)
      (text-width (text-width text font fontsize))
      ;;;
      (text2-width (car (text-wh text2 font fontsize)))
      (text2-height (elem 2 (text-wh text2 font fontsize)))
      ;;;
      (textfs-width)
      (textfs-height)
      (gradient-layer)
    )
  )
```

Ezek után jön maga a mandala. Először létrejön egy réteg (layer) Amit feltöltünk a felhasználó által megadott adatokkal. Ezek a a szöveg, a szöveg betűtípusa, illetve a szöveg mérete. A felhasználó által megadott szöveget elhelyezzük a réteg közepére, a megadott betűtípussal és betűmérettel. Ezután a réteget vízszintesen tükrözi a program, és ráhelyezzük az eredeti réteg felé ezzel elérve a szimmetriát, majd a program elforgatja az így keletkezett képet először 90 majd 45 és végül 30 fokkal és minden egyes elforgatás után megismétli a tükrözést. Ezek után a szövegréteget felnagyítja a teljes réteg méretére.

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

(set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ←
  ))
(gimp-image-insert-layer image textfs 0 -1)
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (/ ←
  height 2))
(gimp-layer-resize-to-image-size textfs)

(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate-simple text-layer ROTATE-180 TRUE 0 0)
(set! textfs (car (gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ←
  -LAYER)))

(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 2) TRUE 0 0)
(set! textfs (car (gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ←
  -LAYER)))
```

```
(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 4) TRUE 0 0)
(set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ↔
-LAYER)))

(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 6) TRUE 0 0)
(set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ↔
-LAYER)))

(plug-in-autocrop-layer RUN-NONINTERACTIVE image textfs)
(set! textfs-width (+ (car(gimp-drawable-width textfs)) 100))
(set! textfs-height (+ (car(gimp-drawable-height textfs)) 100))

(gimp-layer-resize-to-image-size textfs)
```

Ezután két körnek álcázott ellipszist illeszt a képre, amik a szöveget fogják kézrefogni. Az egyik kör vastagsága 8, a másiké pedig 22. Ezek után megtörténik a színátmenet egy új rétegre, és megjelenítettjük a képet.

```
(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width 2) (/ ↔
textfs-width 2)) 18)
  (- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) (+ ↔
textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

(gimp-context-set-brush-size 22)
(gimp-edit-stroke textfs)

(set! textfs-width (- textfs-width 70))
(set! textfs-height (- textfs-height 70))

(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width 2) ↔
(/ textfs-width 2)) 18)
  (- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) (+ ↔
textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

(gimp-context-set-brush-size 8)
(gimp-edit-stroke textfs)

(set! gradient-layer (car (gimp-layer-new image width height RGB-IMAGE ↔
"gradient" 100 LAYER-MODE-NORMAL-LEGACY)))

(gimp-image-insert-layer image gradient-layer 0 -1)
(gimp-image-select-item image CHANNEL-OP-REPLACE textfs)
```

```
(gimp-context-set-gradient gradient)
(gimp-edit-blend gradient-layer BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY ↵
  GRADIENT-RADIAL 100 0 REPEAT-NONE FALSE TRUE 5 .1 TRUE 500 500 (+ (+ ↵
    500 (/ textfs-width 2)) 8) 500)

(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

; (gimp-selection-none image)
; (gimp-image-flatten image)

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

Ezek után már csak a GIMP-be regisztrálás van hátra.

```
(script-fu-register "script-fu-bhax-mandala"
  "Mandala9"
  "Creates a mandala from a text box."
  "Norbert Bátfai"
  "Copyright 2019, Norbert Bátfai"
  "January 9, 2019"
  ""
  SF-STRING      "Text"      "STRING1"
  SF-FONT        "Font"      "Sans"
  SF-ADJUSTMENT  "Font size" '(100 1 1000 1 10 0 1)
  SF-VALUE       "Width"     "1000"
  SF-VALUE       "Height"    "1000"
  SF-GRADIENT    "Gradient"  "Deep Sea"
)
(script-fu-menu-register "script-fu-bhax-mandala"
  "<Image>/File/Create/BHAX"
)
```

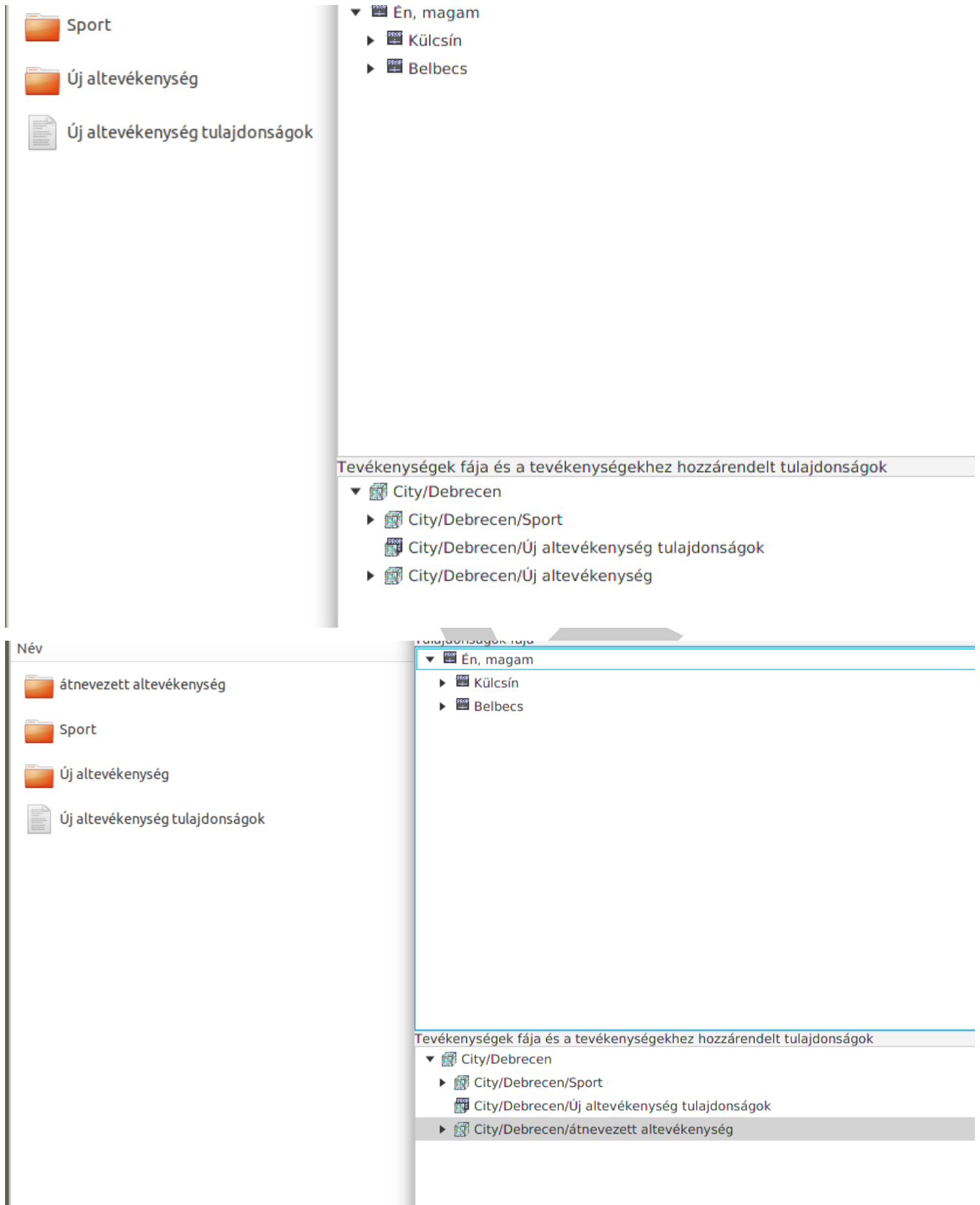
18. fejezet

Helló, !

18.1. 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>

Ebben a feladatban egy meglévő programban kellett hibát keresni, és azt kijavítani. Először is elmondanám azt, hogy ha egy tevékenységre jobb egérgommbal rákattintunk, akkor létre tudunk hozni egy új altevékenységet. Ha pedig erre a tevékenységre kétszer rákattintunk, akkor át tudjuk nevezni az adott tevékenységet. Én egy olyan hibát találtam, hogy ha létrehozunk egy Új altevékenységet, azzal még nincs semmi gond, mivel minden probléma nélkül létrehozza a program. Azonban ha azt a tevékenységet átnevezzük, akkor a program nem átnevezi a tevékenységet, hanem létrehoz egy másik altevékenységet ugyan azzal a névvel. Ezt láthatjuk az első két fényképen:

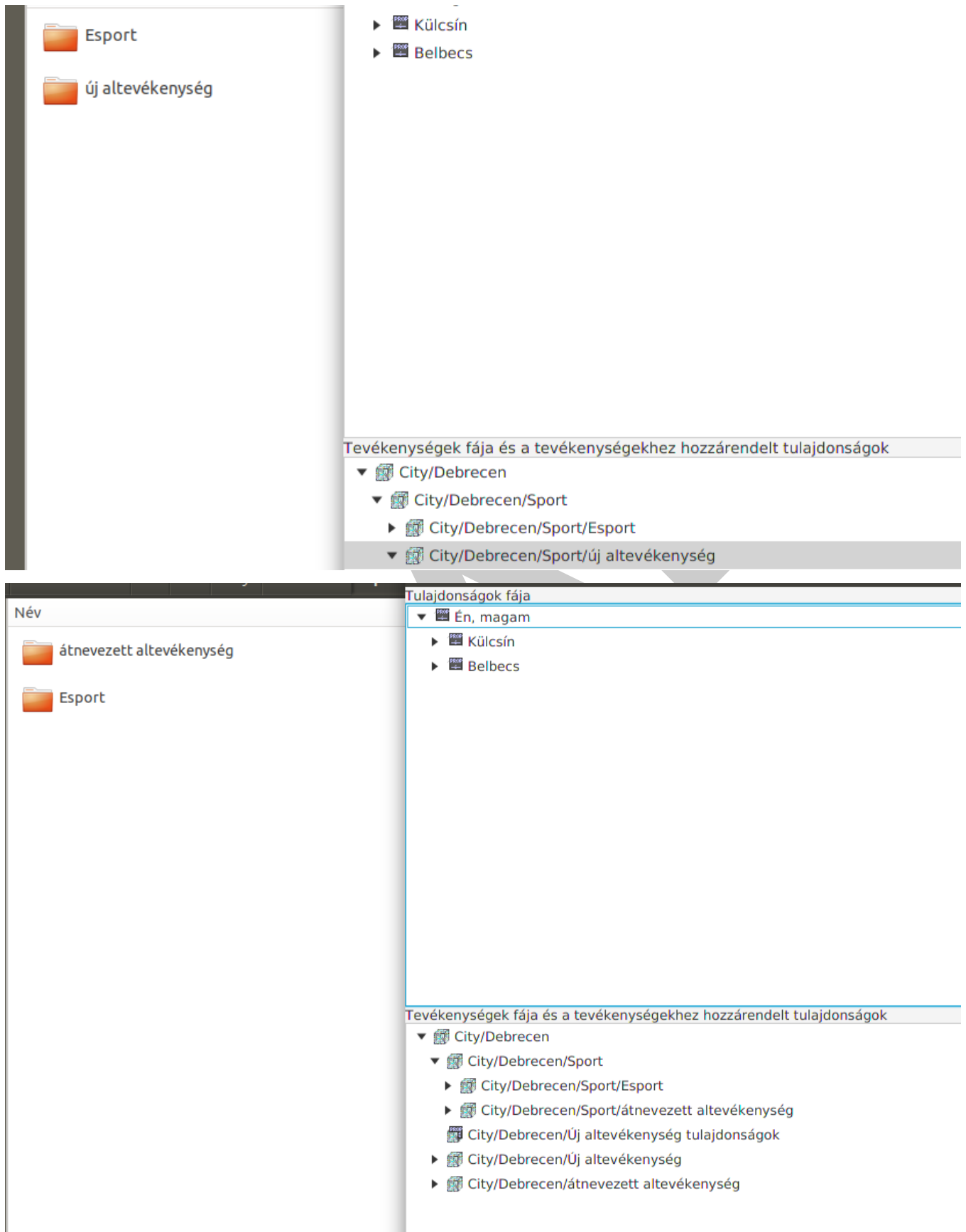


Érzékelhető, hogy ha kellően sok új altevékenységet hoznánk létre és neveznénk át, akkor a végeredmény egy átláthatatlan mappahalom lenne, aminek senki se örül. A hiba javítása elég egyszerű volt. Először is ki kell keresni, hogy hol történik a forráskódban maga az átnevezés. Ez a TextFieldTreeCell osztályon belül

az `editCell()` metóduson belül történik, ami a következőképpen néz ki:

```
private void editCell() {  
  
    if (getItem() == null) {  
        return;  
    }  
  
    String oldText = getItem().toString();  
    textField.setText(oldText);  
  
    textField.setOnKeyReleased((javafx.scene.input.KeyEvent t) -> {  
        if (t.getCode() == javafx.scene.input.KeyCode.ENTER) {  
  
            String newText = textField.getText();  
  
            java.io.File newf = new java.io.File(newText);  
            java.io.File oldf = new java.io.File(oldText);  
            try {  
                if (oldf.isDirectory()) {  
                    //newf.mkdir();  
                    oldf.renameTo(newf);  
                } else {  
                    newf.createNewFile();  
                }  
            } catch (java.io.IOException e) {  
  
                System.err.println(e.getMessage());  
  
            }  
  
            commitEdit(newf);  
        }  
    });  
}
```

Itt a `newf.mkdir();` parancsot kellett átírni arra hogy `oldf.renameTo(newf)` Ez ténylegesen annyit jelent, hogy egy új mappa létrehozása helyett a régi mappát nevezze át az újra. Ezek után már normálisan működik az átnevezés, ami a képeken is látható:



Egy másik hiba az az, hogy ha létrehozunk egy új altevénységet, akkor amíg át nem nevezzük azt, addig nem tudunk létrehozni még egyet, mivel az új altevénység név már létezik. Erről a program csak annyi

tájékoztatást ad a felhasználónak, hogy nem sikerült létrehozni a tevékenységet.

Ezt is viszonylag könnyű javítani. A *TextFieldTreeCell* osztályon belül létre kellett hozni egy számlálót, aminek jelen esetben az *i* nevet adtam, majd pedig amikor meghatározza a program a fájl nevét, akkor a végére még hozzáfűzzük *i*-t. Ezek után egy *while* cikluson belül ellenőrizzük, hogy sikerült-e létrehozni az altevékenységet, azaz a mappát. Ha sikerült akkor megszakítjuk a ciklust és haladunk tovább, ha viszont nem, akkor növeljük *i* értéket 1-el és újrapróbáljuk. Maga a kódcsipet a következőképpen néz ki:

```
public TextFieldTreeCell(javafx.scene.control.TextArea propsEdit) {
    this.propsEdit = propsEdit;
    javafx.scene.control.MenuItem subaMenuItem = new javafx.scene. ↵
        control.MenuItem("Új altevékenység");//"New subactivity");
    addMenu.getItems().add(subaMenuItem);
    subaMenuItem.setOnAction((javafx.event.ActionEvent evt) -> {
        java.io.File file = getTreeItem().getValue();

        boolean sikerulte = false;
        java.io.File f;

        int i = 1;
        while(true){
            f = new java.io.File(file.getPath() + System.getProperty(" ↵
                file.separator") + "Új altevékenység"+ i);

            if (f.mkdir()) {
                javafx.scene.control.TreeItem<java.io.File> newAct
                // = new javafx.scene.control.TreeItem<java.io. ↵
                File>(f, new javafx.scene.image.ImageView(actIcon));
                = new FileTreeItem(f, new javafx.scene.image. ↵
                    ImageView(actIcon));
                getTreeItem().getChildren().add(newAct);
                sikerulte = true;
                break;
            } else {
                i++;
            }
        }

        if(!sikerulte){
            System.err.println("Cannot create " + f.getPath());
        }
    });
}
```

Valamint kép a működésről:

Tevékenységek fája és a tevékenységekhez hozzárendelt tulajdonságok

- ▼ City/Debrecen
 - ▶ City/Debrecen/Sport
 - ▶ City/Debrecen/Új altevékenység3
 - ▶ City/Debrecen/Új altevékenység1
 - ▶ City/Debrecen/Új altevékenység tulajdonságok
 - ▶ City/Debrecen/Új altevékenység
 - ▶ City/Debrecen/Új altevékenység2
 - ▶ City/Debrecen/átnevezett altevékenység
 - ▶ City/Debrecen/Új altevékenység4
 - ▶ City/Debrecen/Új altevékenység5
 - ▶ City/Debrecen/Új altevékenység6

A hibák kijavítása nagyon jó gyakorlati tudást adott az adatfolyamoz kezeléséhez, a streamekhez, és az I/O állománykezeléshez.

18.2. OOCWC Boost ASIO hálózatkezelése

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

Megoldás forrása: <https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll>

Ebben a feladatban a megadott forrásban meg kell magyarázni azt, hogy mi a szerepe az `sscanf()`-nek. Ebben a forráskóban összesen 10-szer fordul elő az `sscanf()`. Ebből most megmutatnék egy párat:

```
{POS}{WS}{INT}{WS}{INT}{WS}{INT}  {
    std::sscanf(yytext, "<pos %d %u %u", &m_id, &from, &to);
    m_cmd = 1001;
}
{CAR}{WS}{INT}                      {
    std::sscanf(yytext, "<car %d", &m_id);
    m_cmd = 1001;
}
{STAT}{WS}{INT}                     {
    std::sscanf(yytext, "<stat %d", &m_id);
    m_cmd = 1003;
}
{GANGSTERS}{WS}{INT}                {
    std::sscanf(yytext, "<gangsters %d", &m_id);
    m_cmd = 1002;
}
```

Ezekről lenne tehát szó. Ahhoz viszont, hogy megértsük, hogy itt mi a feladatuk, először is azt kéne tudnunk, hogy mit is csinál a `sscanf()`. Az `sscanf()` egy fájlkezelő függvény, amelyet arra szoktak használni, hogy a standard input, vagy billentyűzet helyett formázott inputot olvassanak egy Stringből, vagy bufferből. A deklarálása a következőképpen néz ki:

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

Ahol a buffer tartalmazza az olvasandó adatot, a format pedig a beolvasandó adat formája. A formák jelölései mellesleg a következők lehetnek: %c - karakter | %s - String | %d - decimális szám | %i - integer | %u - unsigned decimális szám | %o - oktális integer | %x - hexadecimális integer | %a %e %f %g - lebegőpontos szám | %n - az eddig olvasott karakterek számát adja vissza.

Ezek alapján nézzük meg a forráskódban található legelső `sscanf()`-et, ami a következőképpen néz ki:

```
std::sscanf(yytext, "<pos %d %u %u", &m_id, &from, &to);
```

Akkor az előbbiek alapján ez a függvény a `yytext` bufferből kellene hogy beolvassa az adatokat, amiknek a formája úgy néz ki, hogy először egy "<pos" szöveg van benne, majd egy %d ami egy decimális számot jelöl, valamit két darab %u, amik egy-egy unsigned int-et jelölnek. A decimális szám értékét az `m_id` változónak, az első unsigned int értékét a `from` változónak, a második unsigned int értékét pedig a `to` változónak adja át a függvény. De nézzünk meg egy másikat is, ami pedig a következőképpen néz ki:

```
std::sscanf(yytext, "<init guided %s %d %c>", name, &num, &role);
```

Itt szintén a `yytext`-ből olvassuk az adatokat, amiknek úgy kell kinéznie, hogy először egy "<init guided" szövegnek kell jönnie, aztán pedig egy %s-nek, ami egy Stringet jelöl, majd pedig egy decimális szám, amit ugye a %d jelöl, és végül pedig egy karaktert kell olvasnia, amit a %c jelöl. A String értékét a `name` változónak, a decimális szám értékét a `num` változónak, a karakter értékét pedig a `role` változónak adja át a függvény.

Ezzel pedig már az összes `sscanf()` feladatát meg tudjuk határozni a fentiek alapján.

18.3. SamuCam

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

Ebben a feladatban a webcam kezelését kellett megmutatni a megadott forrásban, ami Qt-t, valamint opencv-t használ. Éppen ezért ha saját magunk is ki szeretnénk próbálni a programot, akkor előbb fel kell telepítenünk a Qt-t, valamint az opencv-t is, amihez rengeteg segítség található az interneten, éppen ezért ebbe én most nem mennék bele, hanem kezdjük is a webkamera használatának az elemzését. Ehhez két forrásfájl lesz nekünk fontos. Az egyik a `main.cpp`, ami kevésbé fontos, a másik pedig a `SamuCam.cpp`, ami sokkal inkább fontosabb, éppen ezért kezdjük is a `main`-nel, amiben csak az alábbi pár sor fontos nekünk:

```
std::string videoStream = parser.value ( webcamipOption ).toString();  
SamuLife samulife ( videoStream, 176, 144 );
```

Itt az történik, hogy a felhasználó által megadott ip címet felhasználja ahhoz, hogy elkezdődhessen a videózás. Ezek után nézzük a `SamuCam.cpp`-t, az első részlete a példányosítás, ami a következőképpen néz ki:

```
SamuCam::SamuCam ( std::string videoStream, int width = 176, int height = 144 )  
: videoStream ( videoStream ), width ( width ), height ( height )  
{
```

```
    openVideoStream();
}

SamuCam::~SamuCam ()
{
}

void SamuCam::openVideoStream()
{
    videoCapture.open ( videoStream );

    videoCapture.set ( CV_CAP_PROP_FRAME_WIDTH, width );
    videoCapture.set ( CV_CAP_PROP_FRAME_HEIGHT, height );
    videoCapture.set ( CV_CAP_PROP_FPS, 10 );
}
```

A `videocapture.open(VideoStream)` utasítás megnyitja a `VideoStream` által eltárolt ip-n keresztül folyó streamet, majd pedig az azt követő három utasítás beállítja a stream szélességét, illetve magasságát, amiknek az értékeit korábban állítottuk be 176-ra illetve 144-re, valamint beállítja a stream fps-ét is 10-re. Ezek után jön a `void SamuCam::run()` metódus, ami steam működését irányítja, és a következőképpen néz ki:

```
void SamuCam::run()
{
    cv::CascadeClassifier faceClassifier;

    std::string faceXML = "lbpcascade_frontalface.xml"; // https://github.com ←  
/Itseez/opencv/tree/master/data/lbpcascades

    if ( !faceClassifier.load ( faceXML ) )
    {
        qDebug() << "error: cannot found" << faceXML.c_str();
        return;
    }

    cv::Mat frame;

    while ( videoCapture.isOpened() )
    {
        QThread::msleep ( 50 );
        while ( videoCapture.read ( frame ) )
        {
            if ( !frame.empty() )
            {
                cv::resize ( frame, frame, cv::Size ( 176, 144 ), 0, 0, cv:: ←  
INTER_CUBIC );
            }
        }
    }
}
```

```
std::vector<cv::Rect> faces;
cv::Mat grayFrame;

cv::cvtColor ( frame, grayFrame, cv::COLOR_BGR2GRAY );
cv::equalizeHist ( grayFrame, grayFrame );

faceClassifier.detectMultiScale ( grayFrame, faces, 1.1, 3, ←
    0, cv::Size ( 60, 60 ) );

if ( faces.size() > 0 )
{
    cv::Mat onlyFace = frame ( faces[0] ).clone();

    QImage* face = new QImage ( onlyFace.data,
                                onlyFace.cols,
                                onlyFace.rows,
                                onlyFace.step,
                                QImage::Format_RGB888 );

    cv::Point x ( faces[0].x-1, faces[0].y-1 );
    cv::Point y ( faces[0].x + faces[0].width+2, faces[0].y + ←
        faces[0].height+2 );
    cv::rectangle ( frame, x, y, cv::Scalar ( 240, 230, 200 ) ←
        );

    emit faceChanged ( face );
}

QImage* webcam = new QImage ( frame.data,
                                frame.cols,
                                frame.rows,
                                frame.step,
                                QImage::Format_RGB888 );

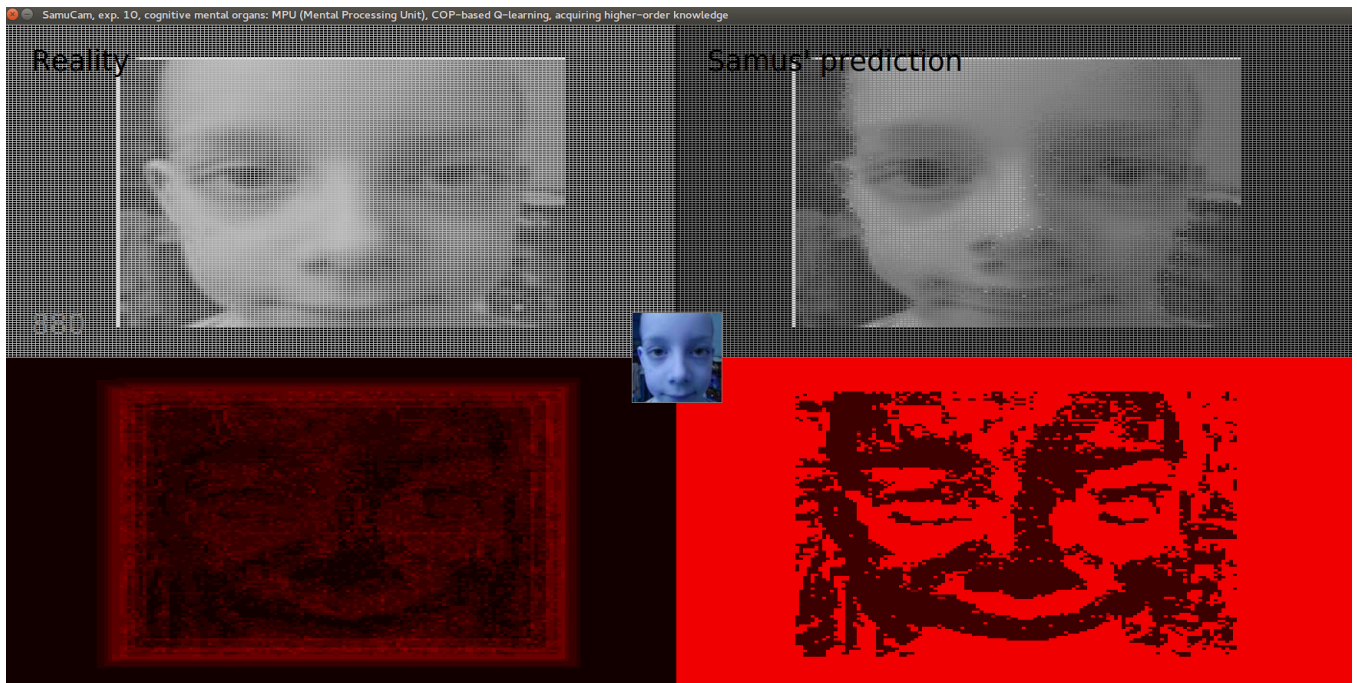
    emit webcamChanged ( webcam );
}

QThread::msleep ( 80 );

}
if ( ! videoCapture.isOpened() )
{
    openVideoStream();
}
}
}
```

Itt először is betölt a program egy CascadeClassifier-t, amely az arcokról készült képeket fogja elemezni. Majd pedig jön két while ciklus, ami a frameket fogja olvasni. A külső while ciklus addig fog menni, amíg megy a stream, a belső pedig addig, amíg jönnek a framek. Ezek után ha az adott frame nem üres, akkor az

adott képkockát átméretezzük 176x144-es méretre. Minden egyes képet eltárol a program a Mat tömbbe, valamint szürkés árnyalatúra állítja az összes eltárolt képkockát. Ezek után a `detectMultiScale()` fogja megkeresni az arcokat a képkockákon. Minden egyes megtalált arc egy téglalapként kerül eltárolásra egy listában. Majd pedig végül az arcokból egy QImage fog készülni. Maga a program pedig működés közben a következőképpen néz ki:



18.4. BrainB

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

Ebben a feladatban a Qt slot-signal mechanizmust kellett bemutatni a megadott forráskódban. Ahhoz, hogy ezt be tudjuk mutatni, először azt kellene megtudni, hogy mi is az. Erre a Qt dokumentációja nagyon jó választ ad, mi szerint a slot-signal mechanizmus az objektumok közötti kommunikációra szolgál. Ez a Qt egyik központi jellemzője. Egy adott esemény bekövetkezésekor egy jel (signal) bocsájtódik ki. A slot pedig egy funkció, amely egy jelre adott válaszként hívódik meg. Egy jelhez több slot is tartozhat, és egy slot több jelre is lehet válasz. Ezt a `connect()` függvénnyel lehet létrehozni aminek a szintaktikája a következőképpen néz ki: `connect(obj1, signal, obj2, slot)`. Ezek alapján már tudjuk, hogy ilyen `connect()` függvényeket kell keresnünk a forrásokban. Valamennyi keresés után a `BrainBWin.cpp` fájlban, azon belül pedig a konstruktorban fogunk találni két ilyet, amelyek a következőképpen néznek ki:

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

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

Ahhoz, hogy ezeket megmagyarázzam, néhány szó a játékról. Az a lényege, hogy négyzetek vannak egy képernyőn, bennük pedig körök. A játék lényege pedig, hogy a bal egérgombot nyomva tartva a Samu

Entropy-n tartjuk az egerünket. Minél jobbak vagyunk, annál több entropy, azaz hős lesz a képernyőn. Egy kép a játékról:



És akkor így már elmondható, hogy mi történik a két előfordulásnál. Az elsőnél, minden esetben, amikor a hősök, azaz Entropy-k helyzete, pozíciója megváltozik, azaz lefut a `heroesChanged()` metódus, akkor a `BrainBThread` fog kibocsájtani egy jelet, amit a `BrainBWin` fog megkapni, és ő pedig frissíteni fogja a hősök helyzetét, azaz le fog futni nála az `updateHeroes()` metódus. A második előfordulásnál pedig, amikor véger ér a játék valami oknál fogva, azaz lefut az `endAndStats()` metódus, akkor a `brainBThread` kibocsájt egy jelet, amit a `BrainBWin` fog észlelni, és nála is le fog futni a saját `endAndStats` metódusa. Még érdekes lehet a signal elküldése, amiket a következő függvényekben találhatunk meg:

```
void draw () {

    cv::Mat src ( h+3*heroRectSize, w+3*heroRectSize, CV_8UC3, cBg );

    for ( Hero & hero : heroes ) {

        cv::Point x ( hero.x-heroRectSize+dispShift, hero.y- ←
            heroRectSize+dispShift );
        cv::Point y ( hero.x+heroRectSize+dispShift, hero.y+ ←
            heroRectSize+dispShift );

        cv::rectangle ( src, x, y, cBorderAndText );

        cv::putText ( src, hero.name, x, cv::FONT_HERSHEY_SIMPLEX, .35, ←
            cBorderAndText, 1 );

        cv::Point xc ( hero.x+dispShift , hero.y+dispShift );
```

```

        cv::circle ( src, xc, 11, cCenter, CV_FILLED, 8, 0 );

        cv::Mat box = src ( cv::Rect ( x, y ) );

        cv::Mat cbox ( 2*heroRectSize, 2*heroRectSize, CV_8UC3, cBoxes ←
        );
        box = cbox*.3 + box*.7;
    }

    cv::Mat comp;

    cv::Point focusx ( heroes[0].x- ( 3*heroRectSize ) /2+dispShift, ←
        heroes[0].y- ( 3*heroRectSize ) /2+dispShift );
    cv::Point focusy ( heroes[0].x+ ( 3*heroRectSize ) /2+dispShift, ←
        heroes[0].y+ ( 3*heroRectSize ) /2+dispShift );
    cv::Mat focus = src ( cv::Rect ( focusx, focusy ) );

    cv::compare ( prev, focus, comp, cv::CMP_NE );

    cv::Mat aRgb;
    cv::extractChannel ( comp, aRgb, 0 );

    bps = cv::countNonZero ( aRgb ) * 10;

    //qDebug() << bps << " bits/sec";

    prev = focus;

    QImage dest ( src.data, src.cols, src.rows, src.step, QImage:: ←
        Format_RGB888 );
    dest=dest.rgbSwapped();
    dest.bits();

    emit heroesChanged ( dest, heroes[0].x, heroes[0].y );
}

```

Valamint:

```

void BrainBThread::run()
{
    while ( time < endTime ) {

        QThread::msleep ( delay );

        if ( !paused ) {

            ++time;

            devel();

        }
        draw();
    }
}

```

```
    }  
    emit endAndStats ( endTime );  
}
```

Mind a két esetben az emit utasítás fog lefutni, ami kibocsátja a jelet, amit a BrainBWin fog megkapni. Ez viszont a slot-signal mechanizmus régi szintaxisa, ami a következőképpen néz ki:

```
connect (   
    sender, SIGNAL( valueChanged( QString, QString ) ),  
    receiver, SLOT( updateValue( QString ) )  
);
```

Azonban ez a Qt-5 ben a következőre változott:

```
connect (   
    sender, &Sender::valueChanged,  
    receiver, &Receiver::updateValue  
);
```

Azaz mostmár egyszerűbb a szintaktika. Először a signalt küldő osztályt kell megadni, majd pedig azt kell megadni, hogy melyik metódus fog lefutni a küldőnél, végül pedig azt, hogy a fogadónál melyik metódusnak kell lefutnia. Ez az új szintaktika ebben a programban a következőképpen néz ki:

```
connect ( brainBThread, &BrainBThread::heroesChanged ,  
          this, &BrainBWin::updateHeroes);  
  
connect ( brainBThread, &BrainBThread::endAndStats,  
          this, &BrainBWin::endAndStats );
```

19. fejezet

Helló, Lauda!

19.1. Port scan

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

Ez a feladat nagyon jó gyakorlati tudást adott a kivételkezelés elméleti részhez. Ebben a feladatban a kivételkezelésre kellett rámutatni a megadott forrásban. Nézzük is meg magát a forráskódot, illetve azt, hogy milyen eredményt ad a program futtatás közben:

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

        for(int i=0; i<1026; ++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!");

            }

    }
}
```

És a kép a futtatásról:

```
1000 nem figyeli
1001 nem figyeli
1002 nem figyeli
1003 nem figyeli
1004 nem figyeli
1005 nem figyeli
1006 nem figyeli
1007 nem figyeli
1008 nem figyeli
1009 nem figyeli
1010 nem figyeli
1011 nem figyeli
1012 nem figyeli
1013 nem figyeli
1014 nem figyeli
1015 nem figyeli
1016 nem figyeli
1017 nem figyeli
1018 nem figyeli
1019 nem figyeli
1020 nem figyeli
1021 nem figyeli
1022 nem figyeli
1023 nem figyeli
1024 nem figyeli
1025 nem figyeli
viktor@viktorubuntu:~/prog2/src/prog2/Lauda$ java KapuSzkenner localhost | grep -v "nem"
631 figyeli
viktor@viktorubuntu:~/prog2/src/prog2/Lauda$
```

Ebből annyit tudunk meg, hogy van egy for ciklus, ami 0-tól 1025-ig fut. Azon belül van egy try-catch szerkezet. A try-ban létrehozunk egy socket objektumot aminek ip címnek a legelső argumentum értékét, portnak pedig i aktuális értékét adjuk. Ilyenkor a program megpróbál a program egy TCP kapcsolatot létrehozni. Ha ez az utasítás nem dob exception-t, akkor kiiratjuk, hogy egy szerver folyamat figyeli a portot, vagy röviden: figyeli, és bezárja a program a socketet. Azonban, ha itt exception-t dob a program, akkor kiírja, hogy nem figyeli. Ezután egy kicsit módosítottam a programot annak érdekében, hogy megtudjam azt, hogy milyen exceptiont dob pontosan a try. A következő eredményt kaptam:

[illegible]

Mint azt láthatjuk, ha nem sikerül kialakítani a kapcsolatot, akkor egy `ConnectException`-t kapunk. Az

oracle [dokumentációja](#) tisztán és érthetően megmagyarázza, hogy a `ConnectException` jelzi azt, hogy hiba történt egy socket egy távoli címhez és porthoz történő csatlakoztatásakor. Általában a kapcsolatot távolról tagadják meg, pl: egy folyamat sem figyeli a cím adott portját. Vagyis, ha sikerül kapcsolatot létrehozni a programnak, akkor tudja, hogy azt a portot figyelik, ha pedig nem, akkor pedig tudja azt, hogy nem figyelik.

19.2. AOP

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

Ez a feladat a fordítást és a kódgenerálást támogató nyelvi elemek elméleti részhez ad egy jó gyakorlati alapot. Ebben a feladatban egy átszővő vonatkozást kellett írni az `LZWBinFa` java átíratába. Mivel az eredeti forráshoz semmit nem kellett hozzáírni, ezért arról nem is beszélnek, hanem térjünk is át rögtön az átszővő vonatkozásra. Ezt a feladatot Aspectj-vel kellett megvalósítani, ami lehetőséget ad arra, hogy anélkül módosítsuk egy program forráskódját, hogy ténylegesen módosítsanánk azt a forráskódot. Itt is pontosan ugyan ez történt. Meg kellett adni kódcsipeteket egy külön fájlban, amiknek megmondhattuk, hogy egy adott metódus előtt vagy után fussanak le. Az Aspectj telepítését linuxon a következő parancs kiadásával lehet megtenni: **apt install aspectj**. Ha ez sikerült, akkor fordítani pedig nem a **javac**, hanem az **ajc** parancs kiadásával kell. Ha jól emlékszem BevProg védési feladat volt, hogy in és post order módon is irassuk ki a fát. Itt most ugyan ezt valósítottam meg aspectj segítségével. Nézzük is a kód első részletét:

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public aspect BinFa{
    int melyseg = 0;

    public pointcut callkiir(LzwBinFa fa, LzwBinFa.Csomopont n, PrintWriter os):call(void LzwBinFa.kiir(LzwBinFa.Csomopont, PrintWriter)) && args(
        n,os) && target(fa) && within(LzwBinFa);

    after(LzwBinFa fa, LzwBinFa.Csomopont n, PrintWriter os):callkiir(fa,n,os){
    }

    public pointcut hivas(LzwBinFa.Csomopont n, PrintWriter os): call(void LzwBinFa.kiir(LzwBinFa.Csomopont, PrintWriter)) && args(n,os);

    after(LzwBinFa.Csomopont n, PrintWriter os) : hivas(n, os){

        try{
            kiirPre(n, new PrintWriter("preorder.txt"));
        }
        catch(FileNotFoundException e) {
            System.out.println(e);
        }
    }
}
```

```
melyseg = 0;

try{
    kiirPost(n,new PrintWriter("postorder.txt"));
}
catch(FileNotFoundException e){
    System.out.println(e);
}
}
```

Először is importáljuk a `PrintWritert` és a `FileNotFoundException`-t, hiszen ezeket használni fogjuk. Majd pedig megmondjuk a programnak, hogy az után, hogy ha meghívódik az eredeti programban az `LzwBinFa.kiir` metódus, akkor annak a metódusnak az argumentumait elkérjük, és után lefutott, próbálja meg lefuttatni a `kiirPre()` valamint a `kiirPost()` függvényeket. Nyilvánvalóan itt kaphatunk `FileNotFoundException` exceptiont, éppen ezért ezt try-catch blokkba írjuk, és kezeljük.

```
public void kiirPost(LzwBinFa.Csomopont elem, java.io.PrintWriter os) {

    if(elem != null) {
        ++melyseg;

        kiirPost(elem.nullasGyermek(), os);
        kiirPost(elem.egyesGyermek(), os);

        for(int i = 0; i < melyseg; i++){
            os.print("---");
        }
        os.print(elem.getBetu());
        os.print("(");
        os.print(melyseg -1);
        os.println(")");

        --melyseg;
    }
}

public void kiirPre(LzwBinFa.Csomopont elem, java.io.PrintWriter os) {

    if(elem != null) {
        ++melyseg;

        for(int i = 0; i < melyseg; i++){
            os.print("---");
        }
        os.print(elem.getBetu());
        os.print("(");
        os.print(melyseg -1);
        os.println(")");
    }
}
```



```

kiirPre(elem.nullasGyermeke(), os);
kiirPre(elem.egyGyermeke(), os);
--melyseg;
}
}

```

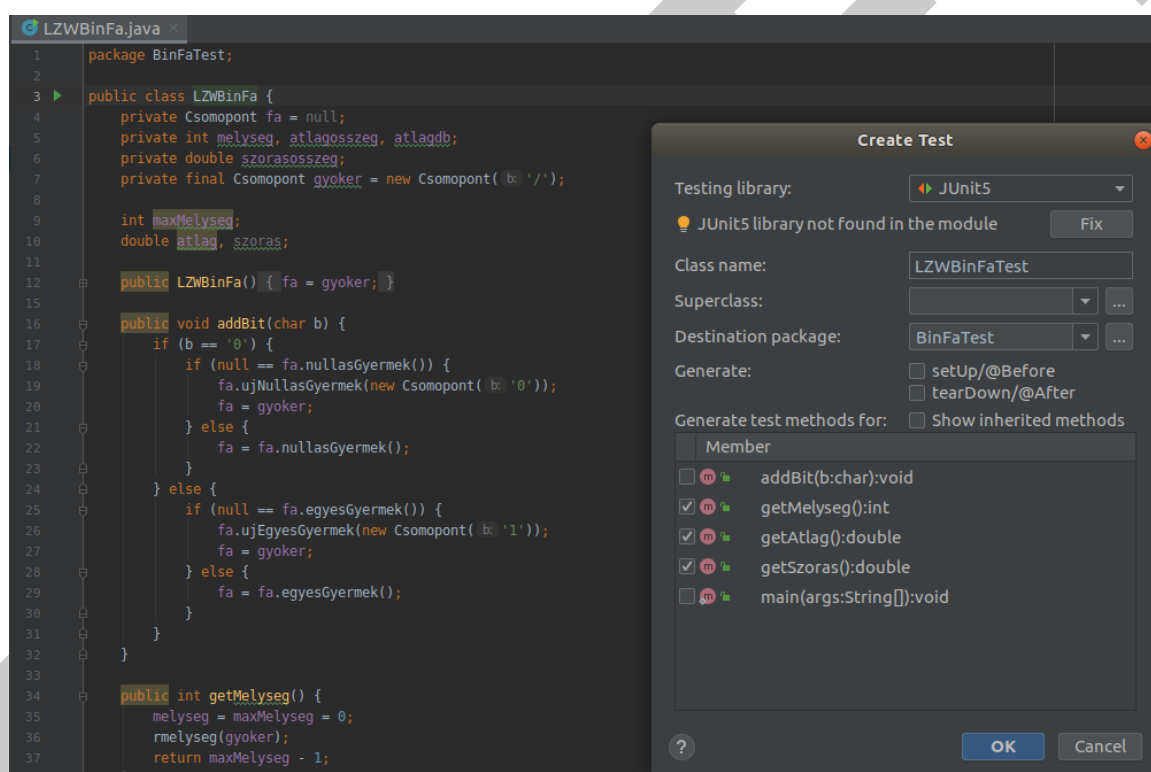
A maradék része a kódnak pedig a két függvény. Az eredeti programban a fát in order módon járta be a program, azaz először az adott elem bal oldali gyerekeit dolgozta fel, aztán az adott elemet, és végül pedig az adott elem jobb oldali gyerekeit. Ezzel szemben itt az első függvény a Post order bejárás, ahol jól láthatóan először az adott elem bal oldali gyerekeit dolgozta fel a függvény, aztán az adott elem jobb oldali gyerekeit, és végül pedig magát az elemet. A második függvény pedig a pre order bejárás, ahol először az adott elemet dolgozta fel a függvény, majd pedig az adott elem bal oldali gyerekeit, és végül pedig az adott elem jobb oldali gyerekeit. Ez a két függvény vég számon tartja és kiírja a fa mélységét is. Végül pedig kép az eredményről:

Traversal Type	Output Sequence (Depth in parentheses)	mélység	átlag	szórás
Inorder	---/(0) -----0(1) -----0(2) -----0(3) -----0(4) -----0(5) -----0(6) -----1(6) -----1(7) -----1(5) -----1(6) -----0(7) -----1(4) -----0(5) -----0(6) -----1(5) -----1(3) -----1(4) -----0(5) -----0(6) -----0(7) -----1(2) -----0(3) -----1(3) -----0(4) -----0(5) -----1(1) -----0(2) -----0(3) -----0(4) -----1(5) -----1(4) -----1(2) -----0(3) -----0(4)	7	5.363636363636363	1.3618169680781094
Postorder	-----0(6) -----1(7) -----1(6) -----0(5) -----0(7) -----1(6) -----1(5) -----0(4) -----0(6) -----0(5) -----1(5) -----1(4) -----0(3) -----0(7) -----0(6) -----0(5) -----1(4) -----1(3) -----0(2) -----0(3) -----0(5) -----0(4) -----1(3) -----1(2) -----0(1) -----1(5) -----0(4) -----1(4) -----0(3) -----0(2) -----0(4) -----0(3) -----1(2) -----1(1) ---/(0)	7	5.363636363636363	1.3618169680781094
Ki (Preorder)	-----1(2) -----0(3) -----0(4) -----1(1) -----0(2) -----1(4) -----0(3) -----1(5) -----0(4) ---/(0) -----1(3) -----0(4) -----0(5) -----1(2) -----0(3) -----0(1) -----1(4) -----0(5) -----0(6) -----0(7) -----1(3) -----0(2) -----1(5) -----1(4) -----0(5) -----0(6) -----1(6) -----0(7) -----1(5) -----0(4) -----1(7) -----1(6) -----0(5) -----0(6)	7	5.363636363636363	1.3618169680781094

19.3. Junit teszt

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

Ebben a feladatban egy a Binfához megadott bemenethez kellett Junit tesztet írni, és remélhetőleg az eredmény megegyezik a papíron kiszámolt eredménnyel. A megadott bemenet az volt hogy 01111001001001000111. Erre kézzel kiszámolva 4-es mélység, 2.75-ös átlag, és 0.9574-es értékű szórás jött ki, ahogy az a feladat leírásában lévő linket megnyitva is látszik. Szóval ha minden jól megy, akkor a JUnit tesztünk is ugyan ezeket az eredményeket fogja kiszámolni. Én ennek a feladatnak a megoldásához IntelliJ Idea-t használtam. Mivel az eredeti LZWBinFa kódján megint nem kellett változtatni, ezért azt most sem taglalnám, hanem térjünk rá egyből a tesztre. Szerencsére IntelliJ-ben lehet tesztet generálni. Ezt azonban ugyan úgy kell elképzelni, mint amikor UML-ből generálunk kódot. Azaz csak egy sablont fogunk kapni. Egyszerűen csak a tesztelni kívánt osztály nevére jobb egérrel kattintunk, és aztán pedig a generate test-re. Éppen ezért ez a feladat a fordítást és a kódgenerálást támogató nyelvi elemek elméleti részhez ad egy jó gyakorlati alapot.



Itt kijelölhetjük, hogy mely metódusokat szeretnénk tesztelni. Majd pedig ha leokézzuk, akkor az előbb említett sablont kapjuk ami a következőképpen néz ki:

Ezt ha egy kicsit kibővítgetjük akkor a következő kódot fogjuk kapni eredményül:

```
package junit;
import static org.junit.jupiter.api.Assertions.*;

class LZWBinFaTest {

    private LZWBinFa binfa = new LZWBinFa();
    private static final String testStr = "01111001001001000111";

    @org.junit.jupiter.api.Test
    void getMelyseg() {
        for (char i : testStr.toCharArray())
            binfa.addBit(i);
        double melyseg = binfa.getMelyseg();
        System.out.println("getMelyseg() teszt -- expecting: 4, got: " + ←
            melyseg);
        assertEquals(4, melyseg);
    }

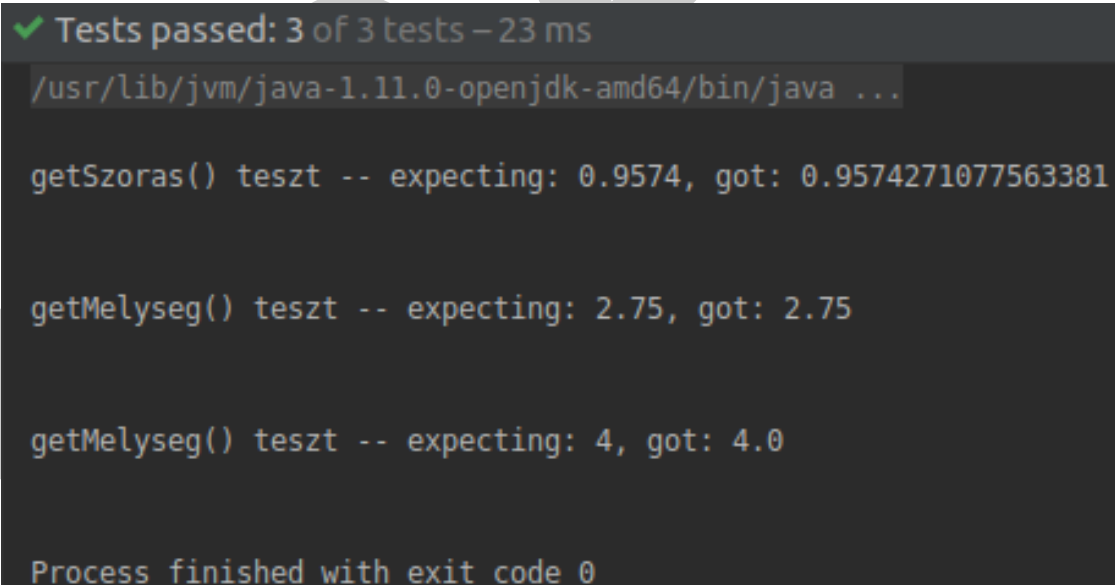
    @org.junit.jupiter.api.Test
    void getAtlag() {
        for (char i : testStr.toCharArray())
            binfa.addBit(i);
        double atlag = binfa.getAtlag();
```

```
        System.out.println("getMelyseg() teszt -- expecting: 2.75, got: " + ↵
            atlag);
        assertEquals(2.75, atlag);
    }

    @org.junit.jupiter.api.Test
    void getSzoras() {
        for (char i : testStr.toCharArray())
            binfa.addBit(i);
        double szoras = binfa.getSzoras();
        System.out.println("getSzoras() teszt -- expecting: 0.9574, got: " ↵
            + szoras);
        assertEquals(0.9574, szoras, 0.001);
    }
}
```

Menjünk sorba. Először is létrehozunk egy LZWBinFa objektumot, illetve létrehozuk a stringet, amivel teszteljük. Ezek után jönnek a metódusok. Először is a `void getMelyseg()` metódus, amiben egy for-each ciklussal belepakoljuk a stringünk karaktereit a binfába, majd pedig meghatározzuk a keletkezett fa mélységét, kiírjuk azt a felhasználó számára, és az `assertEquals()` függvénnyel összehasonlítjuk a kapott eredményt a várt eredménnyel. Pontosan ugyan ez történik a további két metódusban is. Belepakoljuk a string karaktereit a fába, meghatározzuk az átlagot/szórást, és összehasonlítjuk a kapott eredményt a várt eredménnyel.

Végül pedig kép az eredményről:



```
✓ Tests passed: 3 of 3 tests – 23 ms
/usr/lib/jvm/java-1.11.0-openjdk-amd64/bin/java ...

getSzoras() teszt -- expecting: 0.9574, got: 0.9574271077563381

getMelyseg() teszt -- expecting: 2.75, got: 2.75

getMelyseg() teszt -- expecting: 4, got: 4.0

Process finished with exit code 0
```

20. fejezet

Helló, Calvin!

20.1. MNIST

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

Ebben a feladatban a Tensorflow segítségével kellett megtanítani egy mesterséges intelligenciának azt, hogy felismerjen egy kézzel írott, 28x28 pixeles képen lévő számot. Ehhez az MNIST könyvtárat használja tanulásként. A Bátfai Norbert által megadott forrást kicsit át kellett alakítani, mert azóta, hogy az elkészült, rengeteget változott a tensorflow. Legtöbbszöt függvényhívásokat kellett átírni, vagyis inkább hozzáadni a függvényhíváshoz azt, hogy *compat.v1*. és már működtek is. Azonban volt pár sor, amit másképp kellett átírni. Ezek a következők: Először is a 46. sorban a függvényhívást a következőképpen kellett átírni:

```
img = tf.image.decode_png(file, channels=1)
```

Ez annyit jelent, hogy a saját képünknek, amit fel kellene, hogy ismerjen a program, a grayscale-es változata lesz felhasználva. Arra, hogy ez miért kell, két okunk van. Az első az az, hogy az mnist képei is grayscale képek, a másik (a fontosabb) ok pedig az, hogy ha ezt a változtatást nem tesszük meg, akkor hibát fogunk kapni. Éppen ezért ez egy erősen ajánlott változtatás. A következő változtatást pedig a 72. sorban kellett elvégezni, ahol is a paramétereket kellett nevesíteni, amit a következőképpen lehet megoldani:

```
cross_entropy = tf.reduce_mean(tf.nn. ↵  
    softmax_cross_entropy_with_logits(logits=y, labels=y_))
```

Valamint az utolsó kis kódcsipet amit nem át, hanem hozzáírtam a forráshoz, az a következő sor:

```
tf.io.write_graph(sess.graph, "models/", "mnist.pbtxt")
```

Ez annyit jelent, hogy a models mappába hozzon létre egy mnist.pbtxt nevű fájlt, ami a programnak a gráfja. Ezt később Tensorboard-al meg fogjuk tudni nyitni és elemezni. Ezek után nézzük magát a forráskódot:

```
tf.compat.v1.disable_eager_execution()  
x = tf.compat.v1.placeholder(tf.float32, [None, 784])  
W = tf.Variable(tf.zeros([784, 10]))  
b = tf.Variable(tf.zeros([10]))  
y = tf.matmul(x, W) + b  
y_ = tf.compat.v1.placeholder(tf.float32, [None, 10])
```

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits( ↵
    logits=y, labels=y_))
train_step = tf.compat.v1.train.GradientDescentOptimizer(0.5).minimize( ↵
    cross_entropy)

tf.compat.v1.initialize_all_variables().run()
print("-- A halozat tanitasa")
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
    if i % 100 == 0:
        print(i/100, "%")
print("-----")

# Test trained model
print("-- A halozat tesztelese")
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print("-- Pontossag: ", sess.run(accuracy, feed_dict={x: mnist.test. ↵
    images,
                                                    y_: mnist.test.labels}))
print("-----")
```

Először is eltároljuk magát a képet, ami ugye 28x28, azaz összesen 784 pixel. Majd pedig elkezdjük a for ciklussal tanítani a programunkat. Ezer képpel tanítjuk, és minden századik képnél kiírjuk azt, hogy hol járunk. Ezek után ellenőrizzük a pontosságát, és azt is a felhasználó tudomására hozzuk. Majd pedig megpróbálunk felismertetni először egy mnist képet, majd pedig a saját magunk által rajzolt nyolcast.

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

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

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

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

print("-- A saját kezi 8-asom felismerese, mutatom a szamot, a ↵
    tovabblepeshez csukd be az ablakat")

img = readimg()
image = img.eval()
image = image.reshape(28*28)

matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm ↵
```

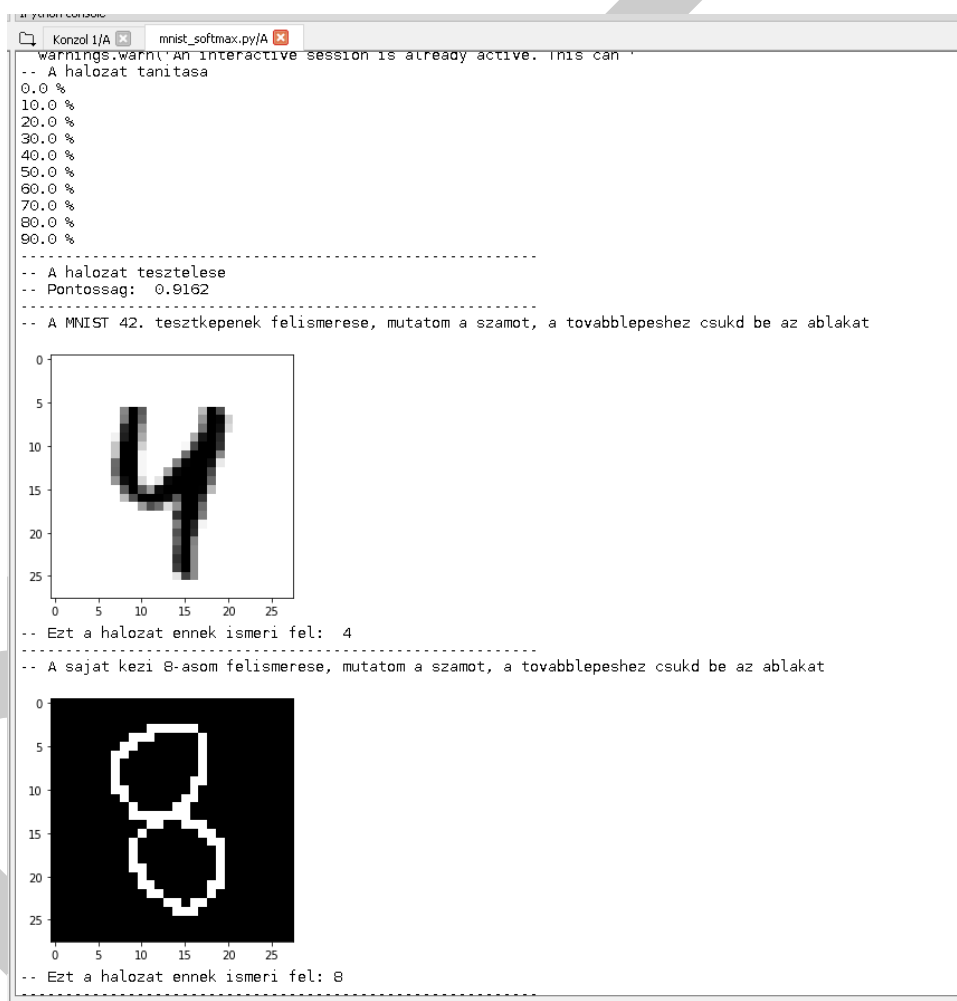
```
.binary)
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()

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

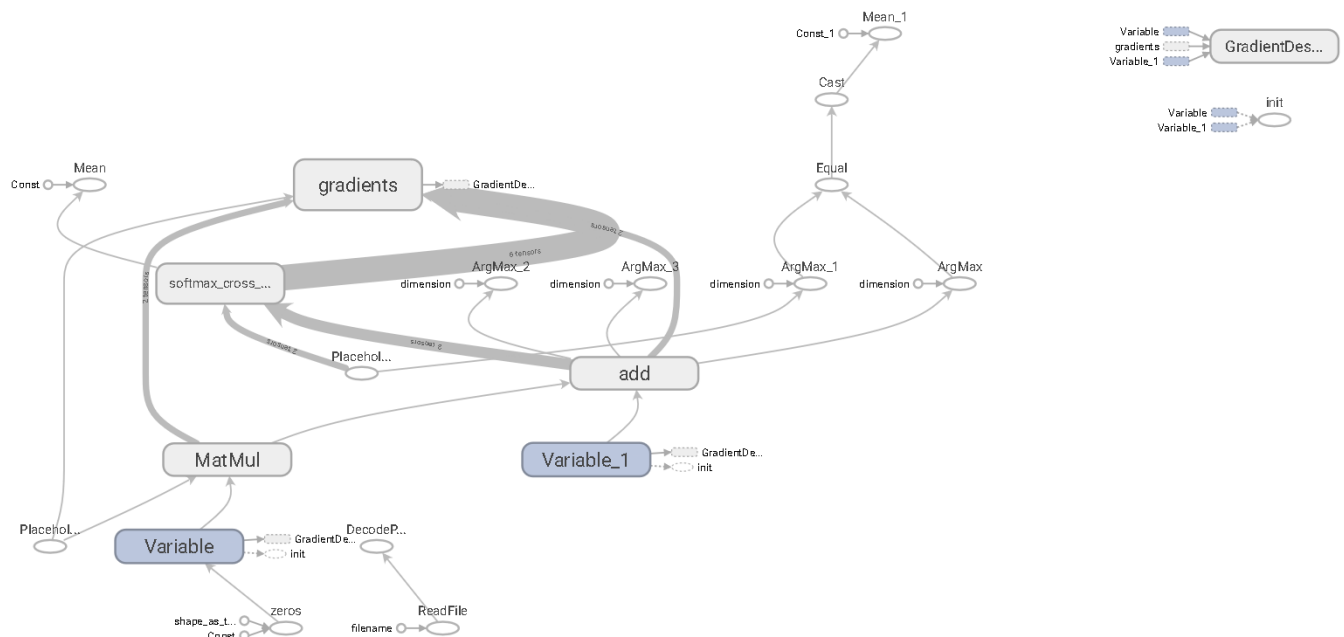
print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")

tf.io.write_graph(sess.graph, "models/", "mnist.pbtxt")
```

Mind a két kép esetében elmentjük az adott képet, aztán pedig kirajzoltatjuk a `matplotlib.pyplot.show()` függvénnyel. Ezek után pedig meghatározza a program, hogy ő minek gondolja az adott számot, és végül pedig kiírjuk a konzolra az eredményt. Az eredmény pedig:



Ezek után már csak a gráf megjelenítése van hátra Tensorboard-ban:



20.2. Deep MNIST

Mint az előző, de a mély változattal.

Ebben a feladatban az előző Tensorflow-os feladatot kellett ismét megoldani, azonban ezúttal már deep mnist-es változattal. Az alap forráskód megtalálható a Tensorflow github repójában, én is azt vettem alapul. Viszont pár dolgot át kellett írni, azonban mindenhol ugyan azt. A következő sorokban: 76, 101, 111, 117, 121, 124, 135, 144, 145, 147, 148, és 179. Ezekben a sorokban a tf és a függvény neve közé azt kellett írni, hogy `.compat.v1` és már működött is az alap program. Azonban az alap program nem tartalmazza a saját magunk által rajzolt kép beolvasását és felismerését. De szerencsénkre ezt egy az egyben kimásolhatjuk az előző forrásból:

```
img = reading()
image = img.eval()
image = image.reshape(28*28)
matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm.binary)
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()
print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
```

Erről már tudjuk, hogy azt jelenti, hogy beolvassuk a saját képünket, átalakítjuk, és elmentjük 8.png néven. Ezek után pedig megjelenítjük az elmentett képet, és kiíratjuk azt, hogy mit gondol a képről a program. Most pedig nézzük a forráskód többi részét.

```
with tf.name_scope('reshape'):
    tf.compat.v1.disable_eager_execution()
mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

x = tf.compat.v1.placeholder(tf.float32, [None, 784])
```



```
y_ = tf.compat.v1.placeholder(tf.float32, [None, 10])

y_conv, keep_prob = deepnn(x)

with tf.name_scope('loss'):
    cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_,
                                                             logits=y_conv)
cross_entropy = tf.reduce_mean(cross_entropy)

with tf.name_scope('adam_optimizer'):
    train_step = tf.compat.v1.train.AdamOptimizer(1e-4).minimize(↵
        cross_entropy)

with tf.name_scope('accuracy'):
    correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
    correct_prediction = tf.cast(correct_prediction, tf.float32)
    accuracy = tf.reduce_mean(correct_prediction)

graph_location = tempfile.mkdtemp()
print('Saving graph to: %s' % graph_location)
train_writer = tf.compat.v1.summary.FileWriter(graph_location)
train_writer.add_graph(tf.compat.v1.get_default_graph())
```

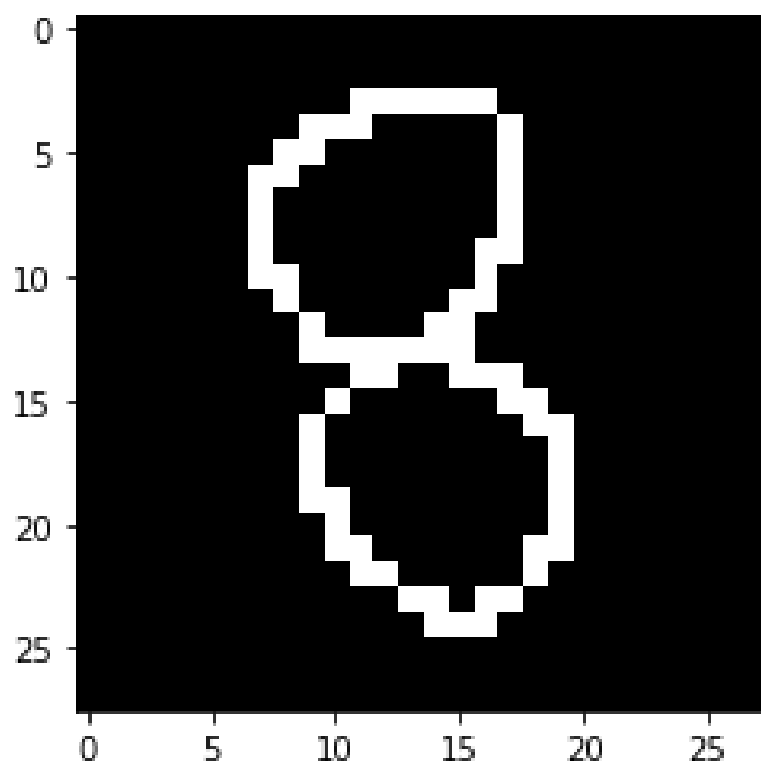
Először is létrehozza modellt a program, és mivel csak a grayscale-es képekre vagyunk kíváncsiak, éppen ezért átalakítjuk őket úgy hogy számunkra megfeleljenek.

```
with tf.compat.v1.Session() as sess:
    sess.run(tf.compat.v1.global_variables_initializer())
    for i in range(20000):
        batch = mnist.train.next_batch(50)
        if i % 100 == 0:
            train_accuracy = accuracy.eval(feed_dict={
                x: batch[0], y_: batch[1], keep_prob: 1.0})
            print('step %d, training accuracy %g' % (i, train_accuracy))
            train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

    print('test accuracy %g' % accuracy.eval(feed_dict={
        x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

Ezek után jön a tanítás része a dolognak. Húszezer képpel tanítjuk a programot, és 100 képenként kiíratjuk azt, hogy éppen hol járunk, illetve azt is, hogy mekkora pontossággal ismeri fel a képeket. Itt egy elég kis számmal fog kezdődni, de ahogy egyre több képet dolgoz fel, úgy fogja egyre jobban felismerni a képeket. Amint kész a tanítás, megmutatjuk neki a saját képünket, és reménykedünk abban, hogy felismeri. Szerencsére az én esetemben igen lett a válasz. Azt még érdemes hozzátenni, hogy a tanítás sokkal több időt vesz igénybe, mint az előző feladatnál. Nekem körülbelül egy órába telt a folyamat. És végül pedig egy kép az eredményről:

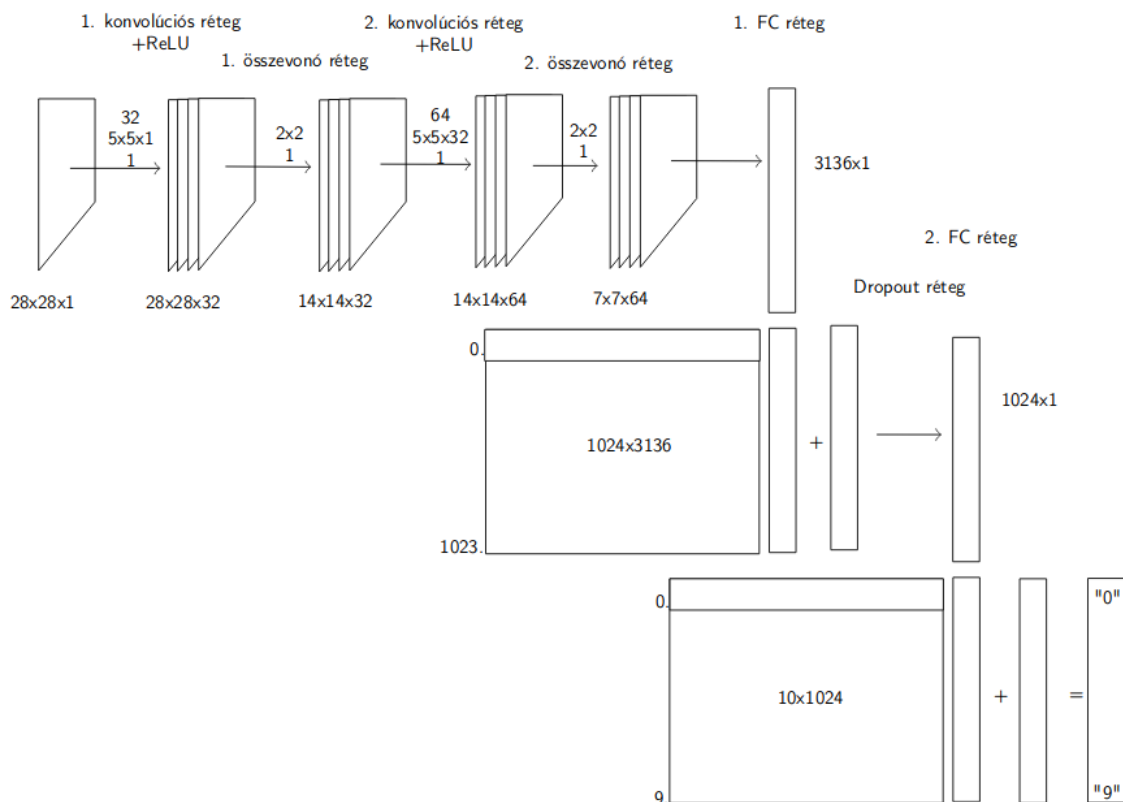
```
step 18000, training accuracy 0.95
step 18100, training accuracy 0.98
step 18200, training accuracy 0.96
step 18300, training accuracy 0.95
step 18400, training accuracy 0.97
step 18500, training accuracy 0.96
step 18600, training accuracy 0.98
step 18700, training accuracy 0.98
step 18800, training accuracy 0.97
step 18900, training accuracy 0.97
step 19000, training accuracy 0.96
step 19100, training accuracy 0.98
step 19200, training accuracy 0.95
step 19300, training accuracy 0.97
step 19400, training accuracy 0.97
step 19500, training accuracy 0.96
step 19600, training accuracy 0.95
step 19700, training accuracy 0.98
step 19800, training accuracy 0.96
step 19900, training accuracy 0.95
test accuracy 0.95
```



```
-- Ezt a halozat ennek ismeri fel: 8
```

A program működését jól szemlélteti az alábbi ábra:

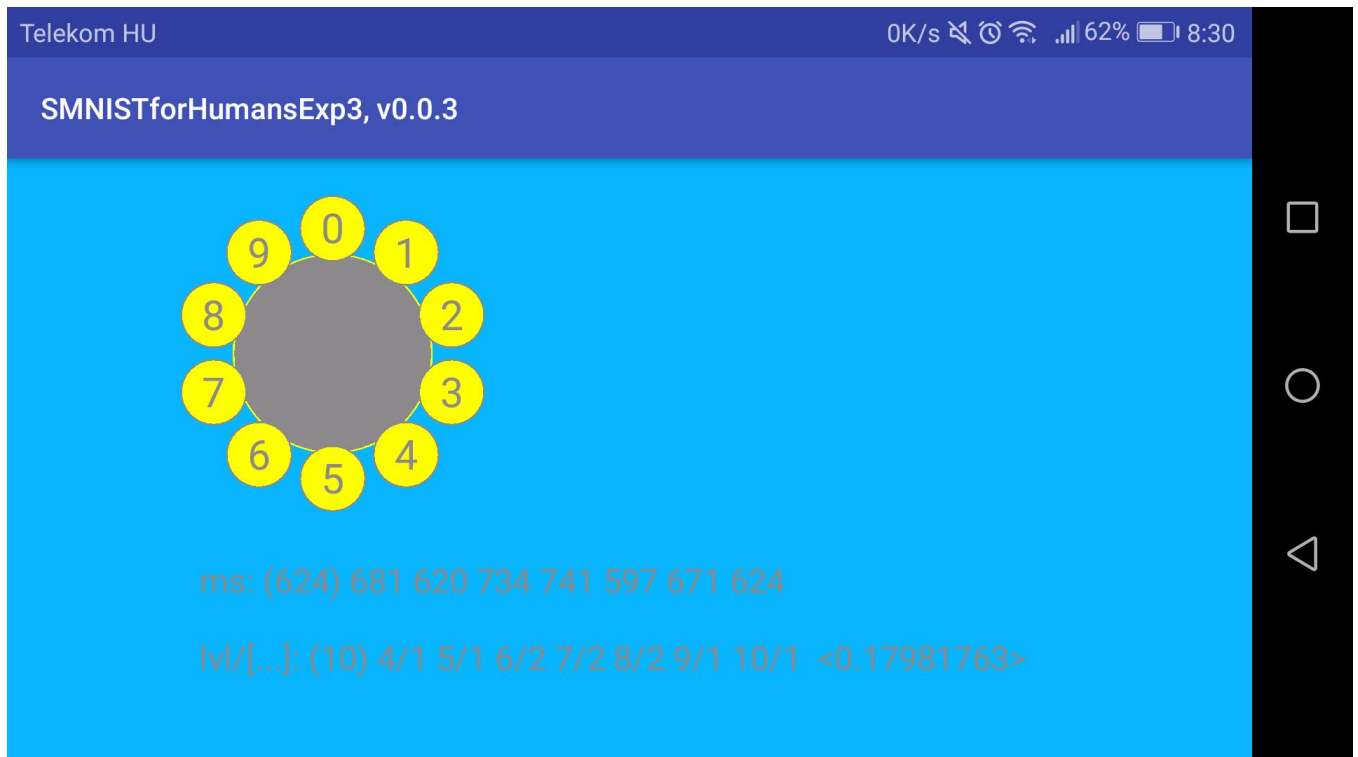
Egy másik TensorFlow "Helló, Világ!": deep MNIST



20.3. SMNIST for Machines

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

Ebben a feladatban az SMNIST for humans program továbbfejlesztett változatát kellett kipróbálni. Az SMNIST for humans még prog1-es téma volt, és az a lényege, hogy egy programban található egy kör, és a körben pedig 0 és 9 darab közötti véletlenszerűen változó számú pontok találhatóak. Minden egyes váltásnál fel kell ismernie a felhasználónak azt, hogy hány darab négyzet van a körben, és az ennek megfelelő számjegyre bökní. Egy kép erről a programról:



Egy átlag ember körülbelül a hatodik szintig képes eljutni. Azonban egy gép valószínűleg sokkal ügyesebb egy embernél. Éppen ezért jött létre az SMNIST for Machines. Ami nagyon hasonló a for humans verzióhoz, azonban ez kifejezetten számítógépek számára jött létre. A mi feladatunk az volt, hogy próbáljunk ki egy ilyen SMNIST for Machines modellt. A futtatáshoz szükségünk lesz a mxnet-re, amit egy egyszerű parancs kiadásával fel lehet telepíteni: **pip install mxnet**. Valamint szükségünk lesz még pár fájlra is, de ezeknek a beszerzési lehetőségét a forrás tárgyalása közben fogom elmondani, amit nézzük is meg.

```
import numpy as np
import gzip
import os, sys
import mxnet as mx
import logging
logging.getLogger().setLevel(logging.INFO)

# Fix the seed
mx.random.seed(43)

# Set the compute context
ctx = mx.cpu()

# Load the data
os.listdir()
os.chdir('D:\smnist')

with gzip.open('t10k-images-idx3-ubyte.gz', 'r') as f:
    test_data = np.frombuffer(f.read(), np.uint8, offset=16)

with gzip.open('t10k-labels-idx1-ubyte.gz', 'r') as f:
    test_labels = np.frombuffer(f.read(), np.uint8, offset = 8)
```

```
with gzip.open('train-images-idx3-ubyte.gz', 'r') as f:
    train_data = np.frombuffer(f.read(), np.uint8, offset=16)

with gzip.open('train-labels-idx1-ubyte.gz', 'r') as f:
    train_labels = np.frombuffer(f.read(), np.uint8, offset = 8)
```

Először is beimportálunk pár dolgot, amire szükség lesz. Az első a numpy, amire a tesztképek átalakításához lesz szükség, a második a gzip, amivel a fájlokat fogjuk megnyitni, a harmadik az os, amivel abba a mappába fog navigálni a program, amelybe a szükséges fájlok találhatóak. Negyedik az mxnet, aminek a tanítás lesz a fő feladata, és végül pedig a logging, amivel logolni fog a program. Majd pedig rögtön látunk egy `setLevel()` függvényt, amivel azt lehet beállítani, hogy milyen szintig ignorálja a program a logging üzeneteket. Ezek után beállítjuk, hogy a cpu fogja elvégezni a munkát, és megnyitjuk a szükséges fájlokat. Ezeket a fájlokat [Erről](#) a linkről lehet beszerezni, és ezek tartalmazzák a tanításhoz szükséges képeket.

```
#Reshape and prepare data
train_data = np.reshape(train_data, (-1,1,28,28))
test_data = np.reshape(test_data, (-1,1,28,28))
batch_size = 100
train_iter = mx.io.NDArrayIter(train_data, train_labels, batch_size, ↵
    shuffle=True)
val_iter = mx.io.NDArrayIter(test_data, test_labels, batch_size)

data = mx.sym.var('data')
# first conv layer
conv1 = mx.sym.Convolution(data=data, kernel=(5,5), num_filter=32)
tanh1 = mx.sym.Activation(data=conv1, act_type="relu")
pool1 = mx.sym.Pooling(data=tanh1, pool_type="max", kernel=(2,2), stride ↵
    =(2,2))

#second conv layer
conv2 = mx.sym.Convolution(data=pool1, kernel=(5,5), num_filter=64)
tanh2 = mx.sym.Activation(data=conv2, act_type="relu")
pool2 = mx.sym.Pooling(data=tanh2, pool_type="max", kernel=(2,2), stride ↵
    =(2,2))

# first fullc layer
flatten = mx.sym.flatten(data=pool2)
fc1 = mx.symbol.FullyConnected(data=flatten, num_hidden=1024)
tanh3 = mx.sym.Activation(data=fc1, act_type="relu")
dropout = mx.sym.Dropout(data=tanh3, p=0.5)

# second fullc
fc2 = mx.sym.FullyConnected(data=dropout, num_hidden=10)

# softmax loss
lenet = mx.sym.SoftmaxOutput(data=fc2, name='softmax')

lenet_model = mx.mod.Module(symbol=lenet, context=ctx)
```

```
lenet_model.fit(train_iter,
                optimizer='sgd',
                optimizer_params={'learning_rate':0.001},
                eval_metric='acc',
                batch_end_callback = mx.callback.Speedometer(batch_size,  ←
                    100),
                num_epoch=50)

test_iter = mx.io.NDArrayIter(test_data, test_labels, batch_size)
# predict accuracy for lenet
acc = mx.metric.Accuracy()
lenet_model.score(test_iter, acc)
print(acc)
```

Ezek után átalakítjuk a képeket 28x28-as grayscale-es képekre, és létrehozuk a szükséges rétegeket. Majd pedig megkezdődik a gép tanítása. Minden századik kép után kiíratjuk azt, hogy hol járunk, hogy milyen gyorsan dolgozik a gép, és a gép pontosságát, illetve minden ötszázadik kép után pedig kiíratunk egy összesített pontosságot. Végül pedig egy kép a futtatásról:

```
INFO:root:Epoch[0] Batch [0-100] Speed: 573.24 samples/sec accuracy=0.124059
INFO:root:Epoch[0] Batch [100-200] Speed: 574.36 samples/sec accuracy=0.239300
INFO:root:Epoch[0] Batch [200-300] Speed: 577.09 samples/sec accuracy=0.413800
INFO:root:Epoch[0] Batch [300-400] Speed: 566.84 samples/sec accuracy=0.671100
INFO:root:Epoch[0] Batch [400-500] Speed: 557.60 samples/sec accuracy=0.814800
INFO:root:Epoch[0] Train-accuracy=0.519700
INFO:root:Epoch[0] Time cost=105.073
INFO:root:Epoch[1] Batch [0-100] Speed: 577.10 samples/sec accuracy=0.884752
INFO:root:Epoch[1] Batch [100-200] Speed: 577.59 samples/sec accuracy=0.897600
INFO:root:Epoch[1] Batch [200-300] Speed: 580.75 samples/sec accuracy=0.910000
INFO:root:Epoch[1] Batch [300-400] Speed: 581.30 samples/sec accuracy=0.916500
INFO:root:Epoch[1] Batch [400-500] Speed: 581.80 samples/sec accuracy=0.926300
INFO:root:Epoch[1] Train-accuracy=0.910533
INFO:root:Epoch[1] Time cost=103.533
INFO:root:Epoch[2] Batch [0-100] Speed: 555.08 samples/sec accuracy=0.934554
INFO:root:Epoch[2] Batch [100-200] Speed: 580.75 samples/sec accuracy=0.943100
INFO:root:Epoch[2] Batch [200-300] Speed: 580.72 samples/sec accuracy=0.941900
INFO:root:Epoch[2] Batch [300-400] Speed: 580.27 samples/sec accuracy=0.944100
INFO:root:Epoch[2] Batch [400-500] Speed: 579.73 samples/sec accuracy=0.950800
INFO:root:Epoch[2] Train-accuracy=0.944150
INFO:root:Epoch[2] Time cost=104.219
INFO:root:Epoch[3] Batch [0-100] Speed: 578.14 samples/sec accuracy=0.953960
INFO:root:Epoch[3] Batch [100-200] Speed: 578.95 samples/sec accuracy=0.954900
INFO:root:Epoch[3] Batch [200-300] Speed: 579.37 samples/sec accuracy=0.955600
INFO:root:Epoch[3] Batch [300-400] Speed: 580.06 samples/sec accuracy=0.960700
INFO:root:Epoch[3] Batch [400-500] Speed: 577.39 samples/sec accuracy=0.957500
INFO:root:Epoch[3] Train-accuracy=0.957017
INFO:root:Epoch[3] Time cost=103.608
INFO:root:Epoch[4] Batch [0-100] Speed: 577.58 samples/sec accuracy=0.966931
INFO:root:Epoch[4] Batch [100-200] Speed: 578.08 samples/sec accuracy=0.960900
INFO:root:Epoch[4] Batch [200-300] Speed: 578.24 samples/sec accuracy=0.963600
INFO:root:Epoch[4] Batch [300-400] Speed: 579.58 samples/sec accuracy=0.963900
INFO:root:Epoch[4] Batch [400-500] Speed: 577.38 samples/sec accuracy=0.965600
INFO:root:Epoch[4] Train-accuracy=0.964600
INFO:root:Epoch[4] Time cost=103.840
INFO:root:Epoch[5] Batch [0-100] Speed: 576.99 samples/sec accuracy=0.965743
INFO:root:Epoch[5] Batch [100-200] Speed: 580.76 samples/sec accuracy=0.967900
INFO:root:Epoch[5] Batch [200-300] Speed: 579.64 samples/sec accuracy=0.968100
INFO:root:Epoch[5] Batch [300-400] Speed: 580.74 samples/sec accuracy=0.969200
INFO:root:Epoch[5] Batch [400-500] Speed: 579.33 samples/sec accuracy=0.970500
INFO:root:Epoch[5] Train-accuracy=0.968583
INFO:root:Epoch[5] Time cost=103.534
INFO:root:Epoch[6] Batch [0-100] Speed: 580.14 samples/sec accuracy=0.972871
INFO:root:Epoch[6] Batch [100-200] Speed: 578.85 samples/sec accuracy=0.969500
INFO:root:Epoch[6] Batch [200-300] Speed: 572.85 samples/sec accuracy=0.970300
INFO:root:Epoch[6] Batch [300-400] Speed: 541.86 samples/sec accuracy=0.972900
INFO:root:Epoch[6] Batch [400-500] Speed: 577.08 samples/sec accuracy=0.973800
INFO:root:Epoch[6] Train-accuracy=0.971800
INFO:root:Epoch[6] Time cost=105.175
```

IV. rész

Irodalomjegyzék

DRAFT

20.4. Általános

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

20.5. C

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

20.6. C++

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

20.7. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf , 2004.

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az 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.