

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

| | <i>TITLE :</i> | | |
|---------------|---|--------------------|------------------|
| | Univerzális programozás | | |
| <i>ACTION</i> | <i>NAME</i> | <i>DATE</i> | <i>SIGNATURE</i> |
| WRITTEN BY | Bátfai, Norbert ÁCs Tálinger, Mark-Imre | 2020. november 24. | |

REVISION HISTORY

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

Ajánlás

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

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

DRAFT

Tartalomjegyzék

| | |
|--|----------|
| I. Bevezetés | 1 |
| 1. Vízió | 2 |
| 1.1. Mi a programozás? | 2 |
| 1.2. Milyen doksikat olvassak el? | 2 |
| 1.3. Milyen filmeket nézzek meg? | 2 |
| II. Tematikus feladatok | 3 |
| 2. Helló, Turing! | 5 |
| 2.1. Végtelen ciklus | 5 |
| 2.2. Lefagyott, nem fagyott, akkor most mi van? | 7 |
| 2.3. Változók értékének felcserélése | 9 |
| 2.4. Labdapattogás | 10 |
| 2.5. Szóhossz és a Linus Torvalds féle BogoMIPS | 12 |
| 2.6. Helló, Google! | 13 |
| 2.7. 100 éves a Brun téTEL | 14 |
| 2.8. A Monty Hall probléma | 15 |
| 3. Helló, Chomsky! | 17 |
| 3.1. Decimálisból unárisba átváltó Turing gép | 17 |
| 3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen | 18 |
| 3.3. Hivatalos nyelv | 19 |
| 3.4. Saját lexikális elemző | 20 |
| 3.5. l33t.l | 21 |
| 3.6. A források olvasása | 22 |
| 3.7. Logikus | 23 |
| 3.8. Deklaráció | 24 |

| | |
|--|-----------|
| 4. Helló, Caesar! | 26 |
| 4.1. int *** háromszögmátrix | 26 |
| 4.2. C EXOR titkosító | 28 |
| 4.3. Java EXOR titkosító | 29 |
| 4.4. C EXOR törő | 30 |
| 4.5. Neurális OR, AND és EXOR kapu | 32 |
| 4.6. Hiba-visszaterjesztéses perceptron | 35 |
| 5. Helló, Mandelbrot! | 37 |
| 5.1. A Mandelbrot halmaz | 37 |
| 5.2. A Mandelbrot halmaz a std::complex osztállyal | 41 |
| 5.3. Biomorfok | 44 |
| 5.4. A Mandelbrot halmaz CUDA megvalósítása | 46 |
| 5.5. Mandelbrot nagyító és utazó C++ nyelven | 49 |
| 5.6. Mandelbrot nagyító és utazó Java nyelven | 50 |
| 6. Helló, Welch! | 53 |
| 6.1. Első osztályom | 53 |
| 6.2. LZW | 56 |
| 6.3. Fabejárás | 60 |
| 6.4. Tag a gyökér | 61 |
| 6.5. Mutató a gyökér | 64 |
| 6.6. Mozgató szemantika | 65 |
| 7. Helló, Conway! | 67 |
| 7.1. Hangyszimulációk | 67 |
| 7.2. Java életjáték | 69 |
| 7.3. Qt C++ életjáték | 75 |
| 7.4. BrainB Benchmark | 76 |
| 8. Helló, Schwarzenegger! | 79 |
| 8.1. Szoftmax Py MNIST | 79 |
| 8.2. Mély MNIST | 81 |
| 9. Helló, Chaitin! | 82 |
| 9.1. Iteratív és rekurzív faktoriális Lisp-ben | 82 |
| 9.2. Gimp Scheme Script-fu: króm effekt | 82 |
| 9.3. Gimp Scheme Script-fu: név mandala | 83 |

| | |
|--|------------|
| 10. Helló, Gutenberg! | 84 |
| 10.1. Programozási alapfogalmak | 84 |
| 10.2. Programozás bevezetés | 86 |
| 10.3. Programozás | 87 |
| III. Második felvonás | 89 |
| 11. Helló, Arroway! | 91 |
| 11.1. OO szemlélet | 91 |
| 11.2. „Gagyi” | 94 |
| 11.3. Yoda | 94 |
| 11.4. EPAM: Objektum példányosítás programozási mintákkal | 95 |
| 12. Helló, Liskov! | 97 |
| 12.1. Liskov helyettesítés sértése | 97 |
| 12.2. Szülő-gyerek | 99 |
| 12.3. EPAM: Liskov féle helyettesíthetőség elve, öröklődés | 100 |
| 12.4. EPAM: Interfész, Osztály, Absztrakt Osztály | 101 |
| 13. Helló, Mandelbrot! | 104 |
| 13.1. Forward engineering UML osztálydiagram | 104 |
| 13.2. EPAM: Neptun tantárgyfelvétel modellezése UML-ben | 107 |
| 13.3. EPAM: Neptun tantárgyfelvétel UML diagram implementálása | 107 |
| 13.4. EPAM: OO modellezés | 111 |
| 14. Helló, Chomsky! | 113 |
| 14.1. Encoding | 113 |
| 14.2. EPAM: Order of everything | 114 |
| 14.3. EPAM: Bináris keresés és Buborék rendezés implementálása | 115 |
| 14.4. EPAM: Saját HashMap implementáció | 117 |
| 15. Helló, Stroustrup! | 121 |
| 15.1. JDK osztályok | 121 |
| 15.2. EPAM: It's gone. Or is it? | 122 |
| 15.3. EPAM: Kind of equal | 123 |
| 15.4. EPAM: Java GC | 124 |

| | |
|---|------------|
| 16. Helló, Gödel! | 126 |
| 16.1. STL map érték szerinti rendezése | 126 |
| 16.2. EPAM: Mátrix szorzás Stream API-val | 127 |
| 16.3. EPAM: LinkedList vs ArrayList | 130 |
| 16.4. EPAM: Refactoring | 130 |
| 17. Helló, ! | 132 |
| 17.1. OOCWC Boost ASIO hálózatkezelése | 132 |
| 17.2. EPAM: XML feldolgozás | 133 |
| 17.3. EPAM: ASCII Art | 135 |
| 17.4. EPAM: Titkos üzenet, száll a gépben! | 137 |
| 18. Helló, Lauda! | 140 |
| 18.1. Port scan | 140 |
| 18.2. EPAM: DI | 141 |
| 18.3. EPAM: JSON szerializáció | 146 |
| 18.4. EPAM: Kivételkezelés | 150 |
| 19. Helló, Berners-Lee! | 152 |
| 19.1. Java és C++ OOP programozási nyelvek: | 152 |
| 19.2. Bevezetés a mobilprogramozásba | 158 |
| IV. Irodalomjegyzék | 161 |
| 19.3. Általános | 162 |
| 19.4. C | 162 |
| 19.5. C++ | 162 |
| 19.6. Lisp | 162 |

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

DRAFT

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat: Egy végtelen ciklust rengetek modón meg lehet határoni, a legegyszerűbb módja a `for(;;)`. A mellett, hogy ez egyszerű, mások is egyértelműen érteni fogják, hogy oda végtelen ciklust szerettünk volna rakni. A fordító a for-os és while-os ciklusból ugyanazt az assembly kódot fordítja, a for használata csak áltlahatóság és egyértelműség céljából ajánlot.

A végtelen ciklusunk nem csak a meghatározásban, hanem a proceszor terhelésében is különböző, úgyan is leterhelhetjük azt, egyáltalán nem is zavarjuk meg vagy csak pár magot dolgoztatunk meg.

0%-ban terhelhetjük a végtelen ciklusunkban a sleep függvényt is szerepelhetjük. Fontos megjegyezni, hogy a függvény használatához szükségünk van az unistd.h-ra.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    while(1)
    {
        sleep(100);
    }
}
```

| mark@latitude: ~/Asztal/bhax-master/thematic_tutorials/bhax_textbook | | | | | | | | | | | | | |
|---|------|----|-----|---------|--------|-------|---|------|------|---------|-----------------|--|--|
| Fájl Szerkesztés Nézet Keresés Terminál Súgó | | | | | | | | | | | | | |
| top - 13:13:47 up 30 min, 1 user, load average: 1,17, 0,90, 0,82 | | | | | | | | | | | | | |
| Tasks: 251 total, 1 running, 197 sleeping, 0 stopped, 0 zombie | | | | | | | | | | | | | |
| %Cpu(s): 16,0 us, 1,7 sy, 0,0 ni, 82,1 id, 0,2 wa, 0,0 hi, 0,0 si, 0,0 st | | | | | | | | | | | | | |
| KiB Mem : 3931832 total, 169308 free, 1875040 used, 1887484 buff/cache | | | | | | | | | | | | | |
| KiB Swap: 2097148 total, 2056164 free, 40984 used, 1507988 avail Mem | | | | | | | | | | | | | |
| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND | | |
| 7021 | root | 20 | 0 | 303660 | 10964 | 9532 | S | 0,0 | 0,3 | 0:00.02 | cups-browsed | | |
| 7043 | root | 0 | -20 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.00 | kworker/3:2H-kb | | |
| 7076 | root | 0 | -20 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.00 | kworker/4:2H-kb | | |
| 7109 | root | 0 | -20 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.17 | kworker/1:2H-kb | | |
| 7261 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.46 | kworker/0:0-eve | | |
| 7372 | root | 0 | -20 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.00 | kworker/2:2H-kb | | |
| 8147 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.48 | kworker/0:3-eve | | |
| 13082 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.35 | kworker/1:6-eve | | |
| 14227 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.14 | kworker/2:9-eve | | |
| 19311 | mark | 20 | 0 | 728540 | 95456 | 32956 | S | 0,0 | 2,4 | 0:24.88 | gedit | | |
| 19721 | root | 20 | 0 | 23468 | 2368 | 1756 | S | 0,0 | 0,1 | 0:00.02 | mount.ntfs | | |
| 19811 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.34 | kworker/u8:2-i9 | | |
| 19834 | mark | 20 | 0 | 1066772 | 60008 | 39892 | S | 0,0 | 1,5 | 0:03.98 | nautilus | | |
| 19879 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.00 | kworker/0:1-cgr | | |
| 19881 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.10 | kworker/2:1-mm_ | | |
| 19928 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.20 | kworker/1:1-eve | | |
| 19930 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.05 | kworker/3:3-eve | | |
| 19955 | mark | 20 | 0 | 447680 | 10684 | 9304 | S | 0,0 | 0,3 | 0:00.03 | zeitgeist-damo | | |
| 19962 | mark | 20 | 0 | 330916 | 16396 | 13944 | S | 0,0 | 0,4 | 0:00.06 | zeitgeist-fts | | |
| 20001 | mark | 20 | 0 | 31540 | 5532 | 3824 | S | 0,0 | 0,1 | 0:00.06 | bash | | |
| 20197 | mark | 20 | 0 | 2600508 | 107224 | 83696 | S | 0,0 | 2,7 | 0:00.80 | WebExtensions | | |
| 20240 | mark | 20 | 0 | 2576096 | 81124 | 64272 | S | 0,0 | 2,1 | 0:00.30 | Web Content | | |
| 20513 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.06 | kworker/u8:1-ev | | |
| 24559 | mark | 20 | 0 | 4372 | 752 | 688 | S | 0,0 | 0,0 | 0:00.00 | a.out | | |
| 24594 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.00 | kworker/u8:3-ev | | |
| 24595 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.04 | kworker/u8:4-i9 | | |

Látható, hogy az a.out az az a programunk, nem terheli a proceszort.

Egy magot 100%-ban dolgoztatni nagyon egyszerű, mivel semmi egyére nincs szükségünk, mint leírni a végtelen ciklust.

```
#include <stdio.h>
int main()
{
    for(;;);
}
```

| mark@latitude: ~/Asztal/bhax-master/thematic_tutorials/bhax_textbook | | | | | | | | | | | | | |
|---|------|-----|-----|---------|--------|--------|---|-------|------|---------|-----------------|--|--|
| Fájl Szerkesztés Nézet Keresés Terminál Súgó | | | | | | | | | | | | | |
| top - 13:15:42 up 32 min, 1 user, load average: 0,94, 0,89, 0,83 | | | | | | | | | | | | | |
| Tasks: 252 total, 3 running, 195 sleeping, 0 stopped, 0 zombie | | | | | | | | | | | | | |
| %Cpu(s): 33,8 us, 0,8 sy, 0,0 ni, 65,3 id, 0,0 wa, 0,0 ht, 0,1 si, 0,0 st | | | | | | | | | | | | | |
| KiB Mem : 3931832 total, 225156 free, 1884920 used, 1821756 buff/cache | | | | | | | | | | | | | |
| KiB Swap: 2097148 total, 2041596 free, 55552 used, 1508780 avail Mem | | | | | | | | | | | | | |
| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND | | |
| 24697 | mark | 20 | 0 | 4372 | 752 | 688 | R | 100,0 | 0,0 | 0:22.56 | a.out | | |
| 20144 | mark | 20 | 0 | 3095664 | 522428 | 155000 | R | 32,5 | 13,3 | 2:08.74 | Web Content | | |
| 20088 | mark | 20 | 0 | 2963596 | 304660 | 145912 | S | 7,0 | 7,7 | 0:53.10 | firefox | | |
| 20197 | mark | 20 | 0 | 2600508 | 106260 | 83696 | S | 0,7 | 2,7 | 0:00.93 | WebExtensions | | |
| 24590 | mark | 20 | 0 | 52876 | 4180 | 3532 | R | 0,7 | 0,1 | 0:07.19 | top | | |
| 1724 | mark | 20 | 0 | 517944 | 91068 | 71228 | S | 0,3 | 2,3 | 1:08.49 | Xorg | | |
| 1873 | mark | 20 | 0 | 3919036 | 432860 | 118984 | S | 0,3 | 11,0 | 1:57.54 | gnome-shell | | |
| 2103 | mark | 20 | 0 | 222732 | 8280 | 7976 | S | 0,3 | 0,2 | 0:01.55 | ibus-engine-sim | | |
| 1 | root | 20 | 0 | 225748 | 9128 | 6330 | S | 0,0 | 0,2 | 0:04.15 | systemd | | |
| 2 | root | 20 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.00 | kthreadd | | |
| 3 | root | 0 | -20 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.00 | rcu_gp | | |
| 4 | root | 0 | -20 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.00 | rcu_par_gp | | |
| 9 | root | 0 | -20 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.00 | mm_percpu_wq | | |
| 16 | root | 20 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.07 | ksoftirqd/0 | | |
| 11 | root | 20 | 0 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:01.79 | rcu_sched | | |
| 12 | root | rt | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.01 | migration/0 | | |
| 13 | root | -51 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.00 | idle_inject/0 | | |
| 14 | root | 20 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.00 | cpuhp/0 | | |
| 15 | root | 20 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.00 | cpuhp/1 | | |
| 16 | root | -51 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.00 | idle_inject/1 | | |
| 17 | root | rt | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.01 | migration/1 | | |
| 18 | root | 20 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.05 | ksoftirqd/1 | | |
| 20 | root | 0 | -20 | 0 | 0 | 0 | I | 0,0 | 0,0 | 0:00.07 | kworker/1:0H-kb | | |
| 21 | root | 20 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.00 | cpuhp/2 | | |
| 22 | root | -51 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.00 | idle_inject/2 | | |
| 23 | root | rt | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.01 | migration/2 | | |
| 24 | root | 20 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.06 | ksoftirqd/2 | | |
| 27 | root | 20 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.00 | cpuhp/3 | | |
| 28 | root | -51 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.00 | idle_inject/3 | | |
| 29 | root | rt | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.01 | migration/3 | | |
| 30 | root | 20 | 0 | 0 | 0 | 0 | S | 0,0 | 0,0 | 0:00.03 | ksoftirqd/3 | | |

Látható, hogy az a.out az az a programunk, csak az egyik magot terheli.

Ha azt szeretnénk, hogy a végtelen ciklusunk minden magot dolgoztasson kicsit többet kell gépelnünk. Hozzá kell adjuk a #pragma omp parallel sort, amivel azt érjük el, hogy a folyamatott több magon futasssa. használatához szükségünk van az omp-hra. (A fordításnál -fopenmp kapcsolóval kell bővítenünk a parancsot.)

```
#include <stdio.h>
#include <omp.h>
int main()
{
    #pragma omp parallel
    for(;;)
}
```

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
top - 13:18:28 up 34 min, 1 user, load average: 2,36, 1,39, 1,02
Tasks: 252 total, 2 running, 196 sleeping, 0 stopped, 0 zombie
%Cpu(s): 98,8 us, 1,2 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 ht, 0,1 si, 0,0 st
Kb Mem : 3931852 total, 223444 free, 1888736 used, 1819652 buff/cache
Kb Swap: 2097148 total, 2041596 free, 55552 used. 1509048 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
24743 mark 20 0 35572 980 888 R 356,8 0,0 1:09,42 a.out
20144 mark 20 0 3098224 531468 154680 S 35,0 13,5 3:18,58 Web Content
20088 mark 20 0 3022676 297792 145392 S 6,9 7,6 1:10,81 firefox
863 avahi 20 0 47348 3520 2980 S 0,3 0,1 0:01,47 avahi-daemon
1873 mark 20 0 3918856 433416 118872 S 0,3 11,0 2:11,06 gnome-shell
7261 root 20 0 0 0 I 0,3 0,0 0:00,64 kworker/u8:0-19
24590 mark 20 0 52876 4180 3532 R 0,3 0,1 0:07,90 top
1 root 20 0 225748 9128 6336 S 0,0 0,2 0:04,30 systemd
2 root 20 0 0 0 S 0,0 0,0 0:00,00 kthreadd
3 root 0 -20 0 0 I 0,0 0,0 0:00,00 rcu_gp
4 root 0 -20 0 0 I 0,0 0,0 0:00,00 rcu_par_gp
9 root 0 -20 0 0 I 0,0 0,0 0:00,00 mm_percpu_wq
16 root 20 0 0 0 S 0,0 0,0 0:00,07 ksoftirqd/0
11 root 20 0 0 0 I 0,0 0,0 0:01,91 rcu_sched
12 root rt 0 0 0 S 0,0 0,0 0:00,01 migration/0
13 root -51 0 0 0 S 0,0 0,0 0:00,00 idle_inject/0
14 root 20 0 0 0 S 0,0 0,0 0:00,00 cpuhp/0
15 root 20 0 0 0 S 0,0 0,0 0:00,00 cpuhp/1
16 root -51 0 0 0 S 0,0 0,0 0:00,00 idle_inject/1
17 root rt 0 0 0 S 0,0 0,0 0:00,01 migration/1
18 root 20 0 0 0 S 0,0 0,0 0:00,05 kssoftirqd/1
20 root 0 -20 0 0 I 0,0 0,0 0:00,07 kworker/1:0H-kb
21 root 20 0 0 0 S 0,0 0,0 0:00,00 cpuhp/2
22 root -51 0 0 0 S 0,0 0,0 0:00,00 idle_inject/2
23 root rt 0 0 0 S 0,0 0,0 0:00,01 migration/2
24 root 20 0 0 0 S 0,0 0,0 0:00,07 kssoftirqd/2
27 root 20 0 0 0 S 0,0 0,0 0:00,00 cpuhp/3
28 root -51 0 0 0 S 0,0 0,0 0:00,00 idle_inject/3
29 root rt 0 0 0 S 0,0 0,0 0:00,01 migration/3
30 root 20 0 0 0 S 0,0 0,0 0:00,03 kssoftirqd/3
33 root 20 0 0 0 S 0,0 0,0 0:00,00 kdevtmpfs
```

Látható, hogy az a.out az az a programunk, minden magot terhel. Igaz a képen nem 100%-ban úgyan is az üres végétlen ciklus előtt veszi azokat a programokat, amelyek ténylegesen "dolgoznak".

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

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

Megoldás videó:

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

```
Program T100
{

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

main(Input Q)
```

```
{  
    Lefagy (Q)  
}  
}
```

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

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

akár önmagára

```
T100 (T100)  
false
```

ezt a kimenetet adja.

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

```
Program T1000  
{  
  
    boolean Lefagy (Program P)  
    {  
        if (P-ben van végtelen ciklus)  
            return true;  
        else  
            return false;  
    }  
  
    boolean Lefagy2 (Program P)  
    {  
        if (Lefagy (P))  
            return true;  
        else  
            for (;;) ;  
    }  
  
    main (Input Q)  
    {  
        Lefagy2 (Q)  
    }  
}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true

- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

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

Tanulságok, tapasztalatok, magyarázat: Ha úgy vesszük, hogy a T100 és T1000 létező program és T1000 ben meghívjuk saját magát. A t100 alapján ha a programunkba van végtelen ciklus, akkor igaz értéket ad a Lefagy program a Lefagy2 programnak ,így tehát az is igaz értéket fog adni, viszont ha a Lefagy false értéket ad vissza akkor a Lefagy2 belép egy végtelen ciklusba, vagyis a program le fog fagyni. Ebből következik, hogy olyan program, mint a T100 nem működik mivel, ha egy olyan program érkezik bele amiben van végtelen ciklus, akkor a program beáll mert a ciklus nem áll meg.

2.3. Változók értékének felcserélése

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

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

Napjainkba a számítógép fejlettsége és gyorsasága miatt, már egyszerűen megcsinálhatjuk egy segédváltozóval vagy exort-tal, de régen nagyon sokat számított az erőforrások jó felhasználása, elosztása. Ezekkel a megoldásokkal sokkal könnyebb volt a számítógépeknek számolni, ha különbséggel vagy szorzással cserélük fel a változókat. Az utóbbi kettőt nézzük most meg.

Változócsere különbséggel:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a=1;
    int b=2;
    printf("%s\n%d %d\n", "kulonbseggel:", a,b);
    a=a-b;
    b=a+b;
    a=b-a;
    printf("%d %d\n", a,b);
}
```

A printf() függvény a kiíratáshoz használjuk. Az első argumentum a kíratás formátuma, a többi pedig a változók kiíratása. A „%d” azt jelenti, hogy egy egész típusú változót fogunk kiíratni, még a „\n” a sortörést.

Változócsere szorzattal:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a=1;
    int b=2;
```

```
    printf("%s\n%d %d\n", "szorzassal:", a, b);
    a=a*b;
    b=a/b;
    a=a/b;
    printf("%d %d\n", a, b);
}
```

A megoldás itt annyiban különbözik, hogy nem „+” és „–”, -t használunk hanem „*” és „/” -t.

2.4. Labdapattogás

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

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

```
#include <stdio.h>
#include <curses.h>
#include <unistd.h>
int
main ( void )
{
    WINDOW *ablak;
    ablak = initscr ();
    int x = 0;
    int y = 0;
    int xnov = 1;
    int ynov = 1;
    int mx;
    int my;
    for ( ; ) {
        getmaxyx ( ablak, my , mx );
        mvprintw ( y, x, "O" );
        refresh ();
        usleep ( 100000 );
        x = x + xnov;
        y = y + ynov;
        if ( x>=mx-1 ) {
            xnov = xnov * -1;
        }
        if ( x<=0 ) {
            xnov = xnov * -1;
        }
        if ( y<=0 ) {
            ynov = ynov * -1;
        }
        if ( y>=my-1 ) {
            ynov = ynov * -1;
        }
    }
}
```

```
        }
    }
    return 0;
}
```

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

A fejlécben azonál feltűnik újdonság ként a curses.h header, amit azért használunk, hogy elérjük a képernyő kezelő függvényeket.

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

Így hozzuk létre a kimenetet. Az initscr () függvény curses módba lépteti a terminált.

A deklarált x és y -on lesz a kezdő értékünk. Az xnov és ynov pedig a lépésközt mutatja. (lépésenként a koordináta rendszeren xnov, ynov-al való elmozdulást). Az mx és my lesznek a határértékek, hogy a program csak az ablakon belül mozogjon és ne lépjön kibőlőle. Ez az a határ ahol a labda visszapattan.

A végtelen ciklus következetében, a labda addig pattog, amíg a program fút. A ciklusban az első függvény a getmaxyx (), ez határozza meg, hogy mekkora az ablakunk mérete. A refresh() függvénytel frissítjük az ablakot, ezzel látjuk a labda mozgását, a frissítés nélkül a labda végig egy helyben állna a kijelzőn. Közöttük a mvprintw() függvény az x és y tengelyen megrazolja a „ „ között lévő szöveget, számot vagy karaktert, esetünkben az O-t. Az usleep függvény azt szabályozza mennyi ideig altassa a ciklust még újra indul, azaz milyen gyorsan mozogjon a labda.

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

Megnöveljük az értékeket, minden ciklus lefutásnál (mozog a "labda").

A következő négy if-el pedig azt vizsgáljuk, hogy a labda az ablak szélén van-e. Ha igen akkor -1 -el szorozzuk, ezáltal a labda irányt változtat. A fordításnál -Incourses kapcsolót kell használnunk.

Most nézzük meg a programot if-ek nélkül.

Forrás:https://progpater.blog.hu/2011/02/13/megtalaltam_neo_t

```
#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <unistd.h>
int
main (void)
{
    int xj = 0, xk = 0, yj = 0, yk = 0;
    int mx = 80 * 2, my = 24 * 2;
    WINDOW *ablak;
    ablak = initscr ();
    noecho ();
    cbreak ();
    nodelay (ablak, true);
    for (;;) {

```

```
xj = (xj - 1) % mx;
xk = (xk + 1) % mx;
yj = (yj - 1) % my;
yk = (yk + 1) % my;
clear ();
mvprintw (0, 0,
    " ←
    -----");
mvprintw (24, 0,
    " ←
    -----");
mvprintw (abs ((yj + (my - yk)) / 2),
          abs ((xj + (mx - xk)) / 2), "X");
refresh ();
usleep (150000);
}
return 0;
}
```

Magyarázat: Az if-ek helyet megoldáshoz most szükségünk van matematikai számításokra, ehez deklarálunk egész tipusú változókat. A számításokat egy végtelen ciklusban számoljuk és mvprintw-vel íratjuk ki a képernyőre. A clear()-el minden egyes számítás előtt letisztítjuk az ablakot, ez azt csinálja, amit az előzőnél a refresh(). Az első kettő mvprintw-vel a felső és alsó határokat rajzoljuk ki, ebben a dobozban fog pattogni a labda. A harmadikkal pedig a "Labdát". Az Usleep függvény itt is a pattogás sebességét határozza meg.

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:

```
#include <stdio.h>
int main()
{
    int a=1;
    int bit=0;
    do
        bit++;
    while (a<<=1);
    printf("%d %s\n",bit,"bites a szohossz a gepon");
}
```

Ez a program a géünk szó hosszát fogja kiírni, azaz az int méretét. A feladatot a BogoMIPS ben használt while ciklus feltétellel írjuk meg (A BogoMIPS a processzorunk sebességét lemérő program amit Linus Torvalds írt meg).

A main függvényben az első sorban deklaráljuk a változót, amivel megvizsgáljuk a gépünk szóhosszát(int méretét). A "bit" változó fogja a lépéseket számlálni. A programot dowhile ciklussal(háltlesztelős) írjuk meg, mivel a sima while nem számítaná bele az első lépést. Ha még is while-val szeretnénk megírni, a kiíratás előtt a végeredmény növelnünk kell eggyel. A ciklus addig fut amíg az "a" nem lesz egyenlő nullával.

Mi a bitshift operátor? Ugye vesszük az egyet. Ennek a bináris kódja a 0001, a bitshift operátor egy 0 -val eltolja, azaz 0010 kapjuk, ez a 2 szám, a count növekedik tehát az értéke 1 lesz. A ciklus újra lefut és eltolja még egyszer a számot egy 0-val, így 0100 kapunk ami a négy. Ez addig fut, még a gépünk szó hosszán (az int méretén) kívül nem tolja az 1-est. Ekkor a változónk értéké 0 lesz. A ciklus befejeződik, és kiíratjuk hányat lépett. Ez a szám adja meg, hogy hány bites a szóhossz.

2.6. Helló, Google!

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

A PageRank egy keresőmotor amit a Google használ. A programot két fiatal írta meg 1998-ban. Nevét az egyik kitalálója után kapta.

A következőben, egy 4 lapból álló PageRank-at fogunk megnézni. A lapok PageRank-ét az alapján nézzük, hogy hány oldal osztotta meg a saját honlapján az oldal hiperlinkjét.

```
#include <stdio.h>
#include <math.h>
void
kiir (double tomb[], int db)
{
    int i;
    for (i = 0; i < db; ++i)
        printf ("%f\n", tomb[i]);
}
double
tavolsag (double PR[], double PRv[], int n)
{
    double osszeg = 0.0;
    int i;
    for (i = 0; i < n; ++i)
        osszeg += (PRv[i] - PR[i]) * (PRv[i] - PR[i]);
    return sqrt(osszeg);
}
int
main (void)
{
    double L[4][4] = {
        {0.0, 0.0, 1.0 / 3.0, 0.0},
        {1.0, 1.0 / 2.0, 1.0 / 3.0, 1.0},
        {0.0, 1.0 / 2.0, 0.0, 0.0},
        {0.0, 0.0, 1.0 / 3.0, 0.0}
    };
    double PR[4] = { 0.0, 0.0, 0.0, 0.0 };
```

```
double PRv[4] = { 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0 };
int i, j;
for (;;)
{
    for (i = 0; i < 4; ++i)
    {
        PR[i] = 0.0;
        for (j = 0; j < 4; ++j)
            PR[i] += (L[i][j] * PRv[j]);
    }
    if (tavolsag (PR, PRv, 4) < 0.00000001)
        break;
    for (i = 0; i < 4; ++i)
        PRv[i] = PR[i];
}
kiir (PR, 4);
return 0;
}
```

Forrás:https://progpater.blog.hu/2011/02/13/bearazzuk_a_masodik_labot

Kezdjük az új headerrel, ez a math.h. Ez tartalmazza a matematikai számításokhoz szükséges függvényeket. A main() fügvénnyben először is létrehozunk egy mátrixot, ami a lapok összeköttetését adja meg. Ha az érték 0 akkor a lap nincs összekötve az adott lappal és persze önmagával sincs. Ahol 1/2 vagy 1/3 az érték az azt jelzi, hogy az oldal hány oldallal van összekötve, például az 1/2: Az oldal 2 oldallal van összekötve és abbal az egyik kapcsolatot jelzi (az 1).

A PR tömb fogja a PageRank értéket tárolni. A PRv tömb pedig a mátrixal való számításokhoz kell. A következő lépés egy végtelen ciklus. Ez majd a számítások végén a break parancsal megszakítjuk, ha a megadott feltétel teljesül. A for ciklusban van maga a PageRank számítása ami majd a tavolság függvényt is meghívja, ami egy részszámolást tartalmaz. A végtelen cikluson belül lévő ciklusok azért mennek 4-ig, mert 4 oldalt vizsgálunk. A ciklusból való kilépés a "break" parancsal történik, ha a tavolság függvényben kapott eredmény kisebb mint 0.00000001. A végén a kiir függvény megkapja a PR értékeit és az oldalak számát és kiíratja azokat.

2.7. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

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

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

A téTEL Viggo Brun bizonyította 1919-ben. Ezért is nevezték el róla. A téTEL kimondja hogy az ikerprímek reciprok összege a Brun konstanthoz konvergál, ami egy véges érték. Ikerprímeknek nevezzük azokat a prím számokat, melyek különbségének abszolút értéke kettő. Például a 3 és 5 vagy a 11 és 13.

Brun téTEL R szimulációban:

```
library(matlab)
stp <- function(x) {
```

```
primes = primes(x)
diff = primes[2:length(primes)]-primes[1:length(primes)-1]
idx = which(diff==2)
t1primes = primes[idx]
t2primes = primes[idx]+2
rt1plust2 = 1/t1primes+1/t2primes
return(sum(rt1plust2))
}
x=seq(13, 1000000, by=10000)
y=sapply(x, FUN = stp)
plot(x,y,type="b")
```

A számoláshoz először is kell egy matlab könyvtár. A program fő része az stp függvény. Ez a függvény megkapja x-et. X egy határ szám lesz, ami megmondja meddig kell a prímeket megkeresni. Ehez a primes függvényt használjuk. A primes(x) kiírja x-ig az összes prímet. A diff vektorban eltároljuk a primes vektorban tárolt egymás melletti prímek különbségét. A számítást úgy végezzük, hogy a 2 prímtől indulva kivonjuk a prímből az előtte lévő prímet. Az idx el vizsgáljuk meg, hogy mely prímek különbsége 2 és ezek hol vannak (a helyüket a which függvény adja meg). A t1primes vektorban elhelyezzük ezeket a prímeket. A t2primes vektorba pedig ami ezeknél a prímeknél kettővel nagyobb (azaz ikerprímek). rt1plust2 vektorban végezzük a recikroképzést és a pár reciprokát összeadjuk. A returnban pedig a sum függvénnnyel vissza adjuk ezek summázott összegét. Végezetül a plot függvénnnyel lerajzoljuk grafikusan.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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

A Monty Hall-paradoxon egy valószínűségi paradoxon. A lényege, hogy 3 ajóból kell választani, kettő mögött nincs semmi és a haramdik mögött egy értékes nyeremény. Az egészet az bonyolítja, hogy az ajtó kiválasztása után, a műsorvezető kinyítja az egyik ajtót ami nem nyert és felajánlja a váltást. A kérdés az, hogy megéri-e változtatni. A józan paraszti ész azt mondja, hogy nem. Matematikailag azonban az jön ki, hogy igen megéri. Lássuk a programt, ami leszimulálja a döntéseket.

```
kiserletek_szama=10000
kiserlet = sample(1:3, kiserletek_szama, replace=T)
jatekos = sample(1:3, kiserletek_szama, replace=T)
musorvezeto=vector(length = kiserletek_szama)
for (i in 1:kiserletek_szama) {
  if(kiserlet[i]==jatekos[i]) {

    mibol=setdiff(c(1,2,3), kiserlet[i])

  }else{

    mibol=setdiff(c(1,2,3), c(kiserlet[i], jatekos[i]))
```

```
        }
        musorvezeto[i] = mibol[sample(1:length(mibol),1)]
    }
nemvaltoztatesnyer= which(kiserlet==jatekos)
valtoztat=vector(length = kiserletek_szama)
for (i in 1:kiserletek_szama) {
    holvált = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i]))
    valtoztat[i] = holvált[sample(1:length(holvált),1)]

}
valtoztatesnyer = which(kiserlet==valtoztat)
sprintf("Kiserletek szama: %i", kiserletek_szama)
length(nemvaltoztatesnyer)
length(valtoztatesnyer)
length(nemvaltoztatesnyer)/length(valtoztatesnyer)
length(nemvaltoztatesnyer)+length(valtoztatesnyer)
```

Most a kísérletet 10000-szer fogjuk szimulálni. A kísérlet vektorban az első és a harmadik "ajtó" közül választunk 10000-szer. A replace=T-vel tesszük lehetővé, hogy egy eredmény többször is kijöhessen. A játékos választásait a játékos vektornál ugyan így meghatározzuk. A sample() fügvénnyel végezzük a kiválasztást. A musorvezeto vektort a length függvényel a kísérletek számával tesszük egyenlővé. Következik a for ciklus ami 1-től a kísérletek számáig fut (10000). A ciklusban egy feltétel vizsgálat következik. Az if-fel megvizsgáljuk, hogy a játékos álltal választott ajtó megegyezik e a kísérletben szereplő ajtóval. Ha a feltétel igaz egy mibol vektorba beletesszük azokat az ajtokat, amiket a játékos nem választott. Az else ágon, ha a feltétel nem igaz ,akkor azt az ajtót eltároljuk, amit nem választott és amelyik a nyereményt rejt. A musorvezeto vektorban pedig azt az ajtót amit ki fog nyitni. A nemvaltoztat és nyer vektorban azok az esetek vannak, amikor a játékos azt az ajtót választotta előre, ami mögött az ajtó van és nem változtat a döntésén. A valtoztat vektorban pedig azt, mikor megváltozatja a döntését és így nyer. Ezt egy forciklussal vizsgáljuk. A legvégén kiíratjuk az eredményeket, hogy melyik esetben hányszor nyert.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

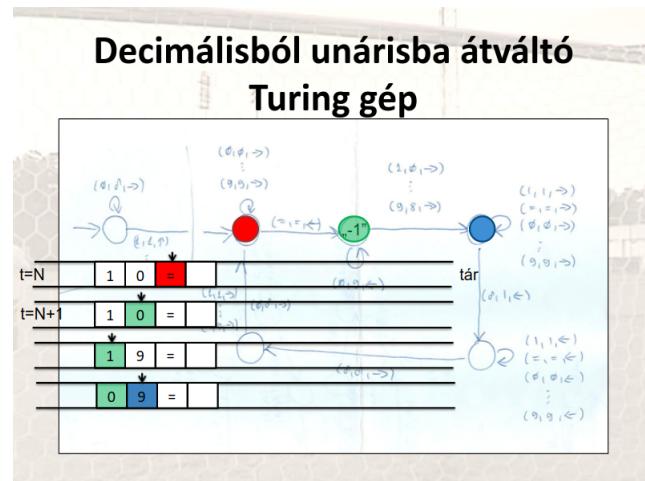
Forrás:<https://slideplayer.hu/slide/2108567/>

Magát a gép fogalmát 1936-ban Alan Turing alkotta meg. A gép decimális számrendszerből unáris számrendszerbe írja át a számot. Az unáris számrendszer másnéven egyes számrendszer, lényege, hogy 1 eseket írunk csak. Ha az 1 számot akarjuk unárisba átváltani, az értéke egy, ha a 2-öt akkor az értéke 11, a tíz pedig 1111111111. A program c++ ban a következő:

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    int tiz=0, szaz=0;
    cout<<"Decimalis szam:\n";
    cin>>a;
    cout<<"A szam unarisban:\n";
    for (int i=0; i<a; i++) {
        cout<<"1";
        ++tiz;
        if (tiz==10) {cout<<" "; tiz=0;}
        if (szaz==100){cout<<"\n";szaz=0;}
    }
    return 0;
}
```

A kód egyszerű. Bekérünk egy decimális számot "a"-ba, és egy forciklus segítségével addig irunk minden egy 1-est amíg i(ami kezdetben 0 és minden egyel növeljük) kisebb mint a. A kimenetet az olvashatóság szempontjából tizesével szóközzel, százasával pedig sortöréssel választja el a program az egyes kimeneti részeket.

Állapot gráf:



Magyarázat az Állapotmenet grafikájának:

A gép beolvassa a memorászalag számjegyeit, (Az ábrán a szám a 10) ha elér az " $=$ " ig, az előtte lévő számmal kezd el dolgozni, még az 0 nem lesz. Az első elem egy 0, de mivel a következő nem nulla, hanem 1 ezért ebből kivon 1-et, azaz hátulról a második elemet 0-ra állítja. A kezdő elem ami 0 volt, az pedig 9 lesz, és ebből minden kivon egyet még 0 nem lesz, (8,7,6,5,4,3,2,1,0), minden kivonásnál kiírat sorban 1 est, így annyi 1 lesz, mint a decimális szám értéke. (Ha 100 lenne a szám akkor az 100 után 099 lenne aztán 098,097....089,088 és így megy 000 ig, és a kimeneten 100db 1 es lesz.)

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

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

A generatív nyelvek kidolgozását Noam Chomsky nevéhez fűzzük. A nyelveket osztályba rendezzük. Van-nak erősebb és gyengébb osztályok. Az erősebb osztály képes létrehozni gyengébb osztályt.

Négy darab alapon fekszik a generatív nyelvtan:

- 1.Terminális szimbólumok. Azaz a konstansok.
- 2.Nem terminális jelek. Ezek a változók.
- 3.Kezdőszimbólum. Egy kijelölt szimbólum.
- 4.Helyettesítési szabályok. Ezzel szavakat értelmezzük majd.

Forrás:<https://slideplayer.hu/slide/2108567/>

1.nyelv

S, X, Y
a, b, c

Az S, X, Y lesznek a változóink. Az a,b,c pedig a konstansok

S \rightarrow abc, S \rightarrow aXbc, Xb \rightarrow bX, Xc \rightarrow Ybcc, bY \rightarrow Yb, aY \rightarrow aaX, aY \rightarrow aa
S (S \rightarrow aXbc)
aXbc (Xb \rightarrow bX)
abXc (Xc \rightarrow Ybcc)
abYbcc (bY \rightarrow Yb)

```
aYbbcc (aY->aa)
aabbcc
S (S->aXbc)
aXbc (Xb->bX)
abXc (Xc->Ybcc)
abYbcc (bY->Yb)
aYbbcc (aY->aaX)
aaXbbcc (Xb->bX)
aabXbcc (Xb->bX)
aabbbXcc (Xc->Ybcc)
aabbbYbcc (bY->Yb)
aabYbbccc (bY->Yb)
aaYbbbccc (aY->aa)
aaabbccc
```

Azt láthatjuk, hogy egészen addig alkalmazzuk a helyettesítési szabályokat még csak konstansaink lesznek. Azaz minden alsóbb osztályt hozunk létre.

2. Itt a változók az A.B.C és a konstansok a,b,c.

```
A, B, C legyenek változók
a, b, c legyenek konstansok
A->aAB, A->aC, CB->bCc, cB->Bc, C->bc
A (A->aAB)
aAB (A->aC)
aaCB (CB->bCc)
aabCc (C->bc)
aabbcc
de lehet így is:
A (A->aAB)
aAB (A->aAB)
aaABB (A->aAB)
aaaABBB (A->aC)
aaaaACBBB (CB->bCc)
aaaabCcBB (cB->Bc)
aaaabCBcB (cB->Bc)
aaaabCBBc (CB->bCc)
aaaabbCcBc (cB->Bc)
aaaabbCBcc (CB->bCc)
aaaabbbCccc (C->bc)
aaaabbbbcccc
```

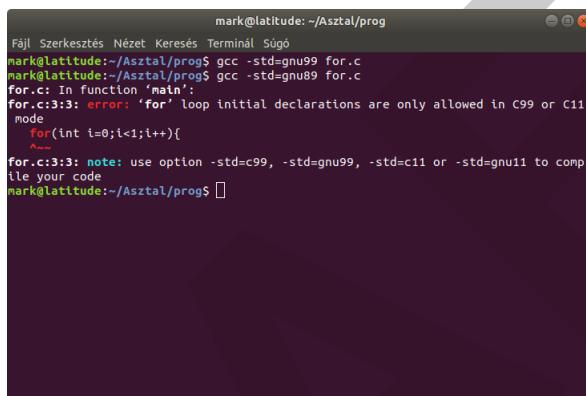
3.3. Hivatkozási nyelv

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

Ahogy a beszélt nyelv, úgy a programozási nyelv is fejlődik. Ennek a bemutatására az alábbi programot fogjuk használni:

```
#include <stdio.h>
int main(){
    for(int i=0;i<1;i++) {
        printf("Lefut");
    }
}
```

Itt ami lényeges, nem a kódban lesz, hanem a fordításnál. Megvizsgáljuk, hogy a C89 es nyelvtan és a C99-es szerint hogyan fordítja le a programot a fordító. Ha a C89 es nyelvtannal fordítom: "gcc -std=gnu89 fajlnev.c -o fajlnev". A program hibát fog írni a for ciklusnál. Most ha a fordításnál átírjuk "gcc -std=gnu99 fajlnev.c -o fajlnev"-re (Azaz a fordító a 99 nyelvtan lesz) ,akkor láthatjuk, hogy lemegy a fordítás és a program működik. A kódon belül, a for ciklusban deklaráltuk az int i-t.



Magyarázat: Az okot a kódon belül, a for ciklusban kell keresni, ugyanis az "i" -t a forcikluson belül deklaráltuk. A C89 nyelvtanban ez még nem volt megengedett, így a fordító hibát írt, de a C99-ben már igen, ezért nem jelez hibát.

3.4. Saját lexikális elemző

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

A program a bemeneten megjelenő valós számokat összeszámolja.

A lexikális elemző kódja:

```
%{
#include <string.h>
int szamok=0;
}%
%[0-9]+ { ++szamok; }
int
main()
{
    yylex();
```

```
printf("%d szam",szamok);  
return 0;  
}
```

A szamok változóval számoljuk hányszor fordul elő szám a bemenetben. A programot a % - jelekkel osztjuk fel részekre.

```
[0-9]+ {++szamok; }
```

Ez a sor adja azt, hogy 0-9 karaktert talál akkor növelje a "szamok" változót. A printf-vel pedig csak kiíratjuk, hogy hány szám volt a bemenetben(ez az elemzés). A yylex() a lexikális elemző

a fordítás a következő:

```
flex program.l
```

ez készít egy "lex.cc.y" fájlt. ezt az alábbi módon futtatjuk.

```
cc lex.yy.c -o program_neve -lfl
```

A futtatáshoz pedig hozzá kell csatolni a vizsgált szöveget.

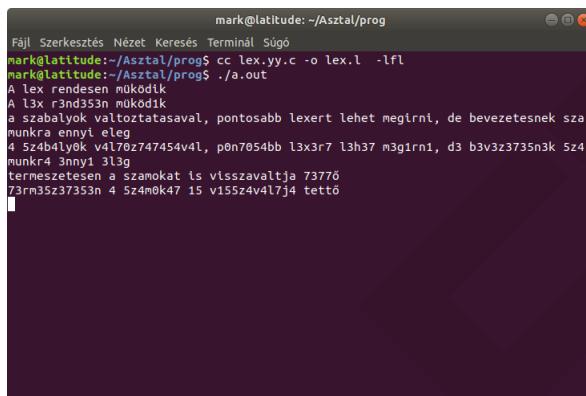
3.5. I33t.I

Lexelj össze egy I33t cipher!

```
% {  
#include <string.h>  
int szamok=0;  
}  
%%  
"0" {printf("o");}  
"1" {printf("i");}  
"3" {printf("e");}  
"4" {printf("a");}  
"5" {printf("s");}  
"7" {printf("t");}  
"o" {printf("0");}  
"i" {printf("1");}  
"e" {printf("3");}  
"a" {printf("4");}  
"s" {printf("5");}  
"t" {printf("7");}  
%%  
int  
main()  
{  
yylex();  
printf("%d szam",szamok);  
return 0;  
}
```

Ez a program lefordítja a l33t nyelven írt titkos szöveget vagy a rendes szöveget írja át a l33t nyelvre. Nem ismer fel minden leet szöveget, úgyan is a program minden számhoz, csak egy betűt rendel és tudjuk egy karaktert több féle képen is lehet leetelni.

A program működése az előzővel majdnem megegyezik, csak annyiban tér el, hogy valós számok helyett, itt most a megadott számokat keresi a bemenetben és azok a számok helyett a l33t nyelvben való megfelelő betűket írja a helyére. Ha pedig a l33t nyelvre akarjuk fordítani, akkor a betűket vizsgálja és a megfelelő számot írja be.



```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
mark@latitude:~/Asztal/prog$ cc lex.yy.c -o lex.l -lfl
mark@latitude:~/Asztal/prog$ ./a.out
A lex rendesen működik
A l3x r3nd353n működik
a szabalyok változtatásaval, pontosabb lexert lehet megírni, de bevezetésnek sza
munkra eynyíl eleg
4 5z4b1y0k v4l70z747454v4l, p0n7054bb l3x3r7 l3h37 m3g1rn1, d3 b3v3z3735n3k 5z4
munkr4 3nnyíl 3l3g
termesztesen a szamokat is visszaváltja 73776
73rm35z3735n 4 5z4m0k47 15 v155z4v4l7j4 tettő
```

3.6. A források olvasása

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

```
if(signal(SIGINT, jelkezelő)==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 megy ránézésre, 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)

Ötször lefútt a for ciklus. Az i értéke a ciklus lefutása előtt változik.

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

Ötször lefútt a for ciklus, mint az előzőnél. Az i értéke a ciklus lefutása után változik.

iv.

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

Ötször lefútt a for ciklus, a vektor elemeinek 0-tól 4-ig veszik fel az értékeit.

v.

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

Egy for ciklus ami addig fut le amíg eléri n-t és d indexértékét s indexértékéhez hasonlítja.

vi.

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

Egy kiíratás, ami az értékeket két függvényből várja.

vii.

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

Egy kiíratás, ami kiír egy értéket egy függvényből és egy változót.

viii.

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

Egy kiíratás, ami kiír egy értéket egy függvényből és egy változót, a függvény most az a memória-címével fog dolgozni.

Tanulságok, tapasztalatok, magyarázat: Figyeljünk az operátorok használatára és az értékváltoztatások sorrendjére, úgyan is ezek figyelmenkívül hagyása később nehézséget okozhat számunkra, például egy nem várt buggal esetleg helytelen érték visszatérítésével.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

Tanulságok, tapasztalatok, magyarázat: 1. minden x-re létezik egy nála nagyobb y ami prím. 2. minden x létezik egy annál nagyobb ikerprímszám. 3. Van olyan y aminél minden x prím kisebb. 4. Van olyan y aminél bármely nagyobb szám nem prímszám.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész

```
int a;
```

- egészre mutató mutató

```
int *b;
```

- egész referencia

```
int &r;
```

- egések tömbje

```
int t[5];
```

- egések tömbjének referencia (nem az első elemé)

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

- egészre mutató mutatók tömbje

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

- egészre mutató mutatót visszaadó függvény

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

- egészre mutató mutatót visszaadó függvényre mutató mutató

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

- egészet 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);
```

Mit vezetnek be a programba a következő nevek?

- ```
int a;
```

Egy egész típusú változót.

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

Egy egész típusú mutatót ami a-ra mutat.

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

a változónak a referenciajára.

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

Egy 5 elemű egész típusú tömböt.

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

Egészek tömbjének referenciaját.

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

5 elemű int-re mutató mutatók tömbjét.

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

Egy függvényt ami int-re mutató mutatót ad vissza.

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

Egy int-re mutató mutatót visszaadó függvényre mutató mutatót.(pl. az előző függvényre)

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

int-et visszaadó, két intet kapó függvényre mutató mutatót visszaadó egészet kapó függvényt.

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

int-et visszaadó, két intet kapó függvényre mutató mutatót visszaadó egészet kapó függvényre mutató mutatót.

4. fejezet

Helló, Caesar!

4.1. int *** háromszögmátrix

A következő programban egy alsó háromszögmátrixot hozunk létre.

Forrás:https://gitlab.com/nbatfai/bhax/blob/master/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/-Caesar/tm.c

Védési videó: https://www.youtube.com/watch?v=T_QqRdhgaP4

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int nr=5;
    double **tm;
    if ((tm=(double **)malloc(nr*sizeof(double)))==NULL)
    {
        return -1;
    }
    for(int i=0; i<nr; i++)
    {
        if((tm[i]=(double *) malloc ((i+1) * sizeof(double)))==NULL)
        {
            return -1;
        }
    }
    for(int i=0; i<nr; i++)
        for(int j=0; j<i+1; j++)
            tm[i][j]=i*(i+1)/2+j;
    for(int i=0; i<nr; i++)
    {
        for(int j=0; j<i+1; j++)
            printf("%f,", tm[i][j]);
        printf("\n");
    }
    tm[3][0]=42.0;
```

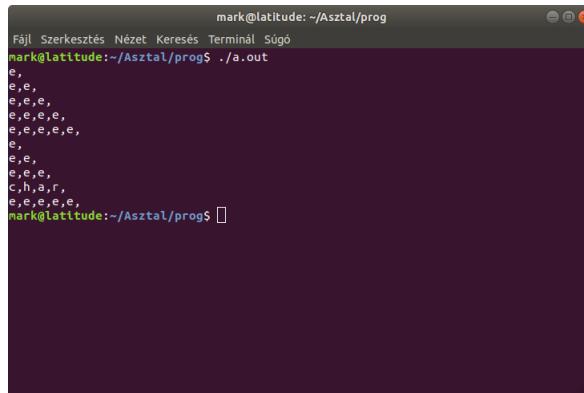
```
(* (tm+3) ) [1]=43.0;
* (tm[ 3]+2)=44.0;
* (* (tm+3)+3)=45.0;
for(int i=0; i<nr; i++)
{
    for(int j=0; j<i+1; j++)
        printf("%f,",tm[i][j]);
    printf("\n");
}
for(int i=0; i<nr; i++)
    free(tm[i]);
free(tm);
return 0;
}
```

Magyarázat: Szokás szerint includoljuk a szükséges fejlécet. A fő függvény első sorában adjuk meg, hogy hány sorunk legyen "int nr=5", ezzel egy 5 soros háromszögmátrixot hozzunk létre. Az első ifben megtaláljuk a malloc függvényt, ami dinamikus memória foglaló, ezzel nr számú double ** mutatót foglalunk le. Ha null értéket ad vissza az azt jelzi, hogy nincs elég hely a foglaláshoz. A következő for lefoglalja a mátrix sorait, az első sornak egy double * mutatót foglal le, a másodiknak 2-t és így tovább, hogy meg legyen az alsó háromszögmátrix forma. A 3. for ciklussal megadjuk a mátrix elemeit. Az "i" a matrix sorai, a "j" pedig a benne lévő mutatók. A "tm[i][j]=i*(i+1)/2+j;" sorral érjük el azt, hogy az elemek minden eggyel nőjenek. A 4. for ciklus a kiíratás. A kiíratás után a harmadik sort megváltoztatjuk. Az érdekes ebben a program részben az, hogy a sor elemeire négy féle képen hívhatunk, mind a négy felírás ekvivalens. A legvégén pedig a free()-vel felszabadítjuk a lefoglalt memóriát, ezzel megelőzve a memória folyást.

Ha helyesen dolgoztunk ezt kell megkapjuk.

```
mark@latitude:~/Asztal/prog$ gedit harom.cp
mark@latitude:~/Asztal/prog$ g++ harom.cp
mark@latitude:~/Asztal/prog$ ./a.out
43.000000,
44.000000,
45.000000,
46.000000,47.000000,48.000000,
49.000000,50.000000,51.000000,52.000000,
```

Érdekesség, hogy a double kicsérélésével, bármilyen tipust használhatunk. A lenti képen a programban double helyet chart használtam.



4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

A feladat lényege a szöveg titkosítás Exor-ral(XOR). Az XOR a "kizáró vagy". A szöveget az alábbi módon titkosítjuk: az eredeti szöveg bájtjaihoz rendelünk titkosító kulcs bájtokat. Aztán XOR műveletet végzunk rajta ami úgy működik, hogy ha a bitek azonosak (1,1;0,0) akkor 0 ad vissza értéknek, ha pedig különbözök (1,0;0,1) akkor 1 et ad vissza. minden bitpáron elvégezve ezt, megkapunk egy titkosított szöveget.

Forrás:https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux/ch05s02.htm

Kód:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#define MAX_KULCS 100
#define BUFFER_MERET 256
int main (int argc, char **argv)
{
    char kulcs[MAX_KULCS];
    char buffer[BUFFER_MERET];
    int kulcs_index = 0;
    int olvasott_bajtok = 0;
    int kulcs_meret = strlen (argv[1]);
    strncpy (kulcs, argv[1], MAX_KULCS);
    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);
    }
}
```

Először is deklaráljuk a szükséges fejléceket. Ezt követően két állandó változót határozunk meg a #define parancsal. Ezeknek az értéke nem változik, azaz konstans értékek lesznek. Az első állandó a MAX_KULCS az értéke 100. A második a BUFFER_MERET 256, ez a beolvasásnál fog kelleni. A fő függvényben egy-egy char típusú tömb méreteivé tesszük a két állandót. Következőkben két változót hozunk be, a kulcs_index, ami a kulcsunk aktuális elemét tárolja, és az olvasott_bajtok, ami a beolvasott bajtok összegét tárolja. A kulcs_merete változóban a kulcs méretét adjuk meg a "strlen()" függvény segítségével. A kulcsot mi adjuk meg az egyik argumentumként. Az strcpy függvény a kulcs kezeléséhez szükséges. A while ciklusban beolvassuk a buffer tömbe a bemenetet. A ciklus addig fut, ameddig van mit beolvasni. A read függvényel lépünk ki a ciklusból. A while cikluson belül a forciklusban végig megyünk az összes bajton és végre hajtjuk a titkosítást.

A futtatás a következő: A fordítás: gcc fajlnev.c -o fajlnev miután lefut, utána futtatjuk: ./fajlnev 56789012 (ez a kulcs) titkosítando.txt (ide írjuk a titkosítandó txt fajl nevét, relíciós jelek között) > titkos.szoveg (titkosított fajlneve). A titkos szöveget a more titkos.szoveg parancsal nézhetjük meg.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Az előző feladatot fogjuk megírni Java-ban. A Java egy objektumorientált programozási nyelv, azaz a nyelv objektumokból és osztályokból áll. A Sun Microsystems informatikai cég alkotta meg. Maga a nyelv a C és a C++ nyelvekhez hasonló, azonban sokkal egyszerűbb (az említett objektumorientáltság miatt). A kezdéshez beszéljünk kicsit az osztályokról a "Class"-okról melyek egy függvények csoportja. Van public és private része, a publikus függvényeket a programból bármi meghívhatja, míg a private függvényeket vagy változókat csak az osztályon belüli vagy barát függvények hívhatják meg.

```
public class ExorTitkosito
{
    public ExorTitkosito(String kulcsSzoveg,
                         java.io.InputStream bejovoCsatorna,
                         java.io.OutputStream kimenocsatorna)
                         throws java.io.IOException
    {
        byte [] kulcs = kulcsSzoveg.getBytes();
        byte [] buffer = new byte[256];
        int kulcsIndex = 0;
        int olvasottBajtok = 0;
        while((olvasottBajtok = bejovoCsatorna.read(buffer)) != -1)
        {
            for(int i=0; i<olvasottBajtok; ++i)
            {
                buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);
                kulcsIndex = (kulcsIndex+1) % kulcs.length;
            }
            kimenocsatorna.write(buffer, 0, olvasottBajtok);
        }
    }
    public static void main(String[] args)
```

```
{  
    try  
    {  
        new ExorTitkosito(args[0], System.in, System.out);  
    }  
    catch(java.io.IOException e)  
    {  
        e.printStackTrace();  
    }  
}
```

Az ExorTitkosito() függvény, kapja meg a bekért argumentumokat. Ha rosszul kapja meg, a throw() hibát adja vissza. A függvény belsőjében történik a titkosítás XOR-val. Ez ugyan úgy működik, mint a fenti C kódban. Ami érdekes lehet számunka az a byte típus, ez 8-bit. Byte típusú lessz a kulcs és a buffer tömb is, ezek tárolják a kulcsot és a beolvastott szöveget.

Vizsgáljuk meg a main függvényt. A Java nyelvben a main az osztály egyik függvénye (eltér a C++ -tol, ahol a main egy különálló fő függvény az osztálytól.) Az alábbi sor "public static void main(String[] args)" a függvény fejléce. A "public" mutatja, hogy publikus, azaz elérhető. A "static"-al jelöljük, hogy része az osztálynak. A void típust meg már ismerjük az előzőkből. A main-be képesek vagyunk argumenetumokat bekérni a terminálból. Ezzen belül láthatjuk a try() és a catch() függvényt, ezekkel a függvényekkel C++ -ban, A try() a hiba üzenetet küldi még a catch() ezt elkapja és kiírja nekünk.

A fordításhoz java fordító kell. Ehez most a "javac"-t fogjuk használni. Ha ez nincs fent a számítógépünkön, akkor a gép jelezni fogja, hogyan kell telepítenünk. Fordítani és futtatni az alábbi módon fogjuk:

```
//Fordítás:  
javac ExorTitkosító.java  
//Futtatás:  
java ExorTitkosító titkosítandó.szöveg > titkosított.szöveg
```

4.4. C EXOR törő

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

Az alábbi feladatban a 3.2 feladatban lévő titkosítóhoz írunk egy programot ami feltöri a titkosított szöveget. A program alapműködése ugyan azon az elven alapszik, mint a 3.2 mivel ugyan úgy XOR-val alakítjuk vissza a szöveget. A lényeg, hogy a kulcsot amivel titkosítottunk azt ismerjük, mert ezzel a kulcsal tudjuk feltörni. Úgy működik, hogy a titkosított bájtokat össze exortáljuk a kulcsal, és így újra az eredeti bájtokat kapjuk. A feladatban a 3.2 ben titkosított azöveget és a kulcsot fogjuk használni, ugyanis erre épül a program.

Kód:

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

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
int tiszta_lehet(const char titkos[], int titkos_meret)
{
    return strcasestr(titkos, "hogy") && strcasestr(titkos, "nem") && ←
        strcasestr(titkos, "az") && strcasestr(titkos, "ha");
}
}
void exor(const char kulcs[], int kulcs_meret, char titkos[], int ←
titkos_meret)
{
    int kulcs_index=0;
    for(int i=0; i<titkos_meret; ++i)
    {
        titkos[i]=titkos[i]^kulcs[kulcs_index];
        kulcs_index=(kulcs_index+1)%kulcs_meret;
    }
}
int exor_tores(const char kulcs[], int kulcs_meret, char titkos[], int ←
titkos_meret)
{
    exor(kulcs, kulcs_meret, titkos, titkos_meret);
    return tiszta_lehet(titkos, titkos_meret);
}
int main(void)
{
    char kulcs[KULCS_MERET];
    char titkos[MAX_TITKOS];
    char *p=titkos;
    int olvasott_bajtok;
    while((olvasott_bajtok=read(0, (void *) p, (p-titkos+OLVASAS_BUFFER< ←
MAX_TITKOS)? OLVASAS_BUFFER:titkos+MAX_TITKOS-p)))
    p+=olvasott_bajtok;
    for(int i=0; i<MAX_TITKOS-(p-titkos);++i)
        titkos[p-titkos+i]='\0';
    for(int ii='0';ii<='9';++ii)
        for(int ji='0';ji<='9';++ji)
            for(int ki='0';ki<='9';++ki)
                for(int li='0';li<='9';++li)
                    for(int mi='0';mi<='9';++mi)
                        for(int ni='0';ni<='9';++ni)
                            for(int oi='0';oi<='9';++oi)
                                for(int pi='0';pi<='9';++pi)
    {
        kulcs[0]=ii;
        kulcs[1]=ji;
        kulcs[2]=ki;
        kulcs[3]=li;
        kulcs[4]=mi;
```

```
kulcs[5]=ni;
kulcs[6]=oi;
kulcs[7]=pi;
if(exor_tores(kulcs,KULCS_MERET,titkos,p-titkos))
printf("Kulcs: [%c%c%c%c%c%c%c] \nTiszta szoveg: [%s]\n",ii,ji,ki,
      li,mi,ni,oi,pi,titkos);
exor(kulcs,KULCS_MERET,titkos,p-titkos);
}
return 0;
}
```

Forrás:https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux/ch05s02.htm

Elsőnek is definiáljuk az állandókat és a fejléceket. Az állandók közül most is a buffer a beolvasáshoz szükséges, a kulcs mérete mégint a kulcsot tartalmazó tömbhöz kell ami az előzőleg használt kód miatt 8. A fő függvény előtt találunk függvényeket. Az átlagos szóhossz és a tiszta lehet függvény a törés gyorsaságát segítik elő. Az átlagos szóhossz megadja az szóhossz atlagat még a tiszta lehet pedig a gyakori magyar szavakat figyeli. A void exor () függvény megkap egy kulcsot, a méretét, a titkos szövegnek a tömbjét és annak a méretét. Itt a forciklusban a kulcsot össze exortálja a titkos szöveggel. Az exor_tores függvény meghívja az exor függvényt is vissza adja a tiszta szöveget. A fő függvényben láthatjuk deklarációk után a titkos szöveg beolvasását. Utána a program megnézi az összes lehetséges permutációt és a megoldást kírja a kimenetre, ezzel a kódval a 3.2 programot használva fel tudjuk törni a szöveget.

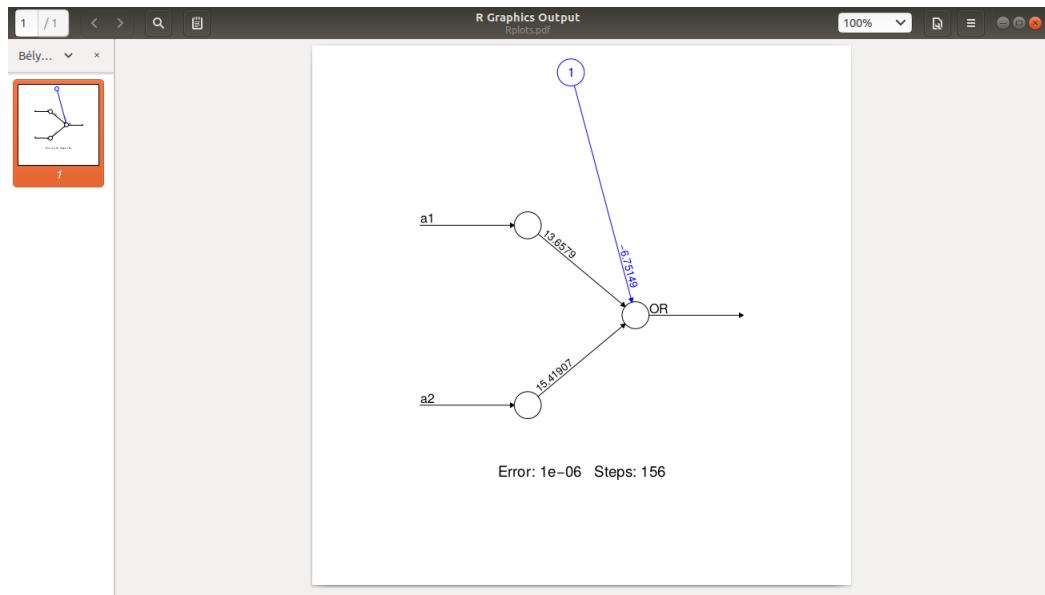
Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html#exor_titkosito

4.5. Neurális OR, AND és EXOR kapu

A feladatban egy Neurális hálozatot fogunk írni R nyelvben. A nevét a neuron-ról kapta, az az az idegsejtéről. Ezekből épül fel az idegrendszer. Ez egy ingerlékeny sejt, ami ingerület fel és leadásával továbbít információt, amit fel is dolgoz.

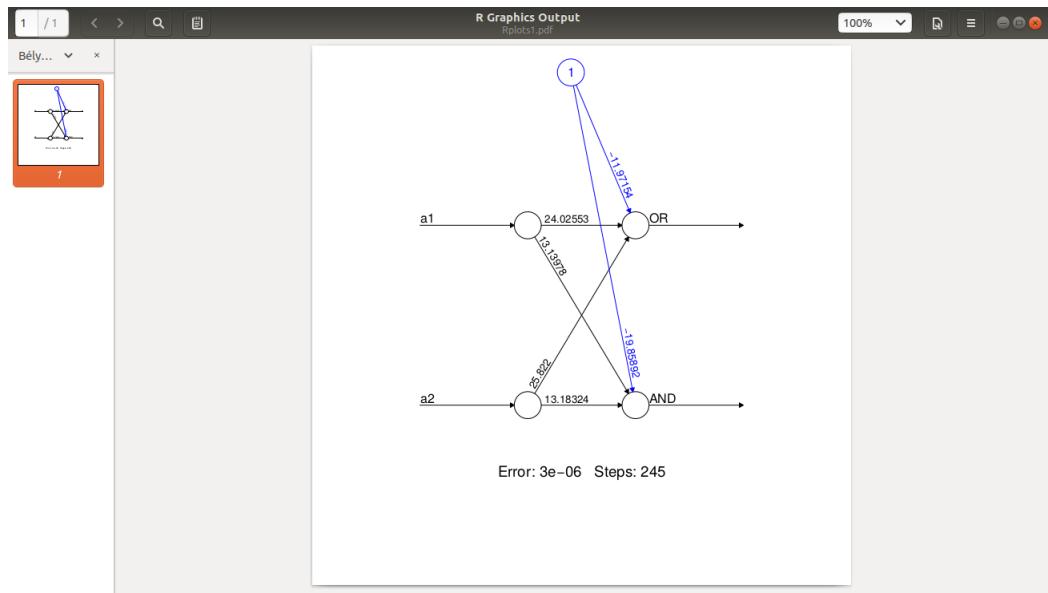
Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/NN_R

```
library(neuralnet)
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
OR      <- c(0,1,1,1)
or.data <- data.frame(a1, a2, OR)
nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE,   ←
                   stepmax = 1e+07, threshold = 0.000001)
plot(nn.or)
compute(nn.or, or.data[,1:2])
```



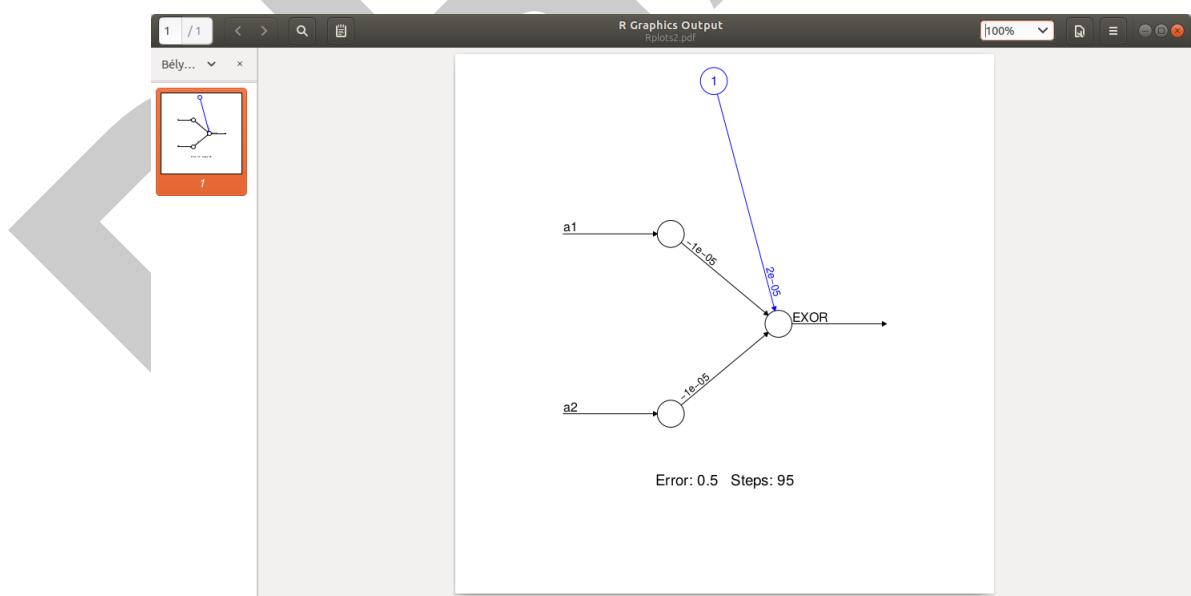
A program elején meghívjuk a neuralnet könyvtárat ami tartalmazza a nekünk szükséges függvényeket. A bemenet az a1 és az a2 lesz. A gép most a logikai vagyot, azaz az OR -t fogja megtanulni. Ha a1 és a2 bemenet 0-t ad. Az OR értéke is 0 lesz, minden más esetben az értéke 1. Ezeket az or.data-ban tárolja el a program. Úgy mond "megtanulja". Az nn.or értékét pedig a neuralnet() függvényel határozzuk meg. A függvény első argumentumában a megtanuladnó érték van, aza hogy az OR értéke 0 legyen vagy 1. A második argumentumban adjuk meg az or.data ami alapján tanulja meg a program. A harmadik argumentumban rejtett neutronok száma van. A stepmax a lépésszámot adja. A plot függvényvel kirajzolunk (lásd a képen) a tanulás folyamatának egyik esetét.

```
library(neuralnet)
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
OR      <- c(0,1,1,1)
AND     <- c(0,0,0,1)
orand.data <- data.frame(a1, a2, OR, AND)
nn.orand <- neuralnet(OR+AND~a1+a2, orand.data, hidden=0, linear.output= FALSE, stepmax = 1e+07, threshold = 0.000001)
plot(nn.orand)
compute(nn.orand, orand.data[,1:2])
```



A programunk azzal bővül, hogy megtanítjuk a programnak az OR-t és az AND-et fogja megtanulni a program. A különbség az előzőtől annyi, hogy az AND csak akkor kap 1 értéket, ha a1 és a2 értéke is 1, különben az AND értéke 0. A tanulás folyamat ugyan olyan mint az előző. A tanulás módját az orand.data-ba mentjük.

```
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
EXOR   <- c(0,1,1,0)
exor.data <- data.frame(a1, a2, EXOR)
nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=0, linear.output=FALSE, ←
    stepmax = 1e+07, threshold = 0.000001)
plot(nn.exor)
compute(nn.exor, exor.data[,1:2])
```

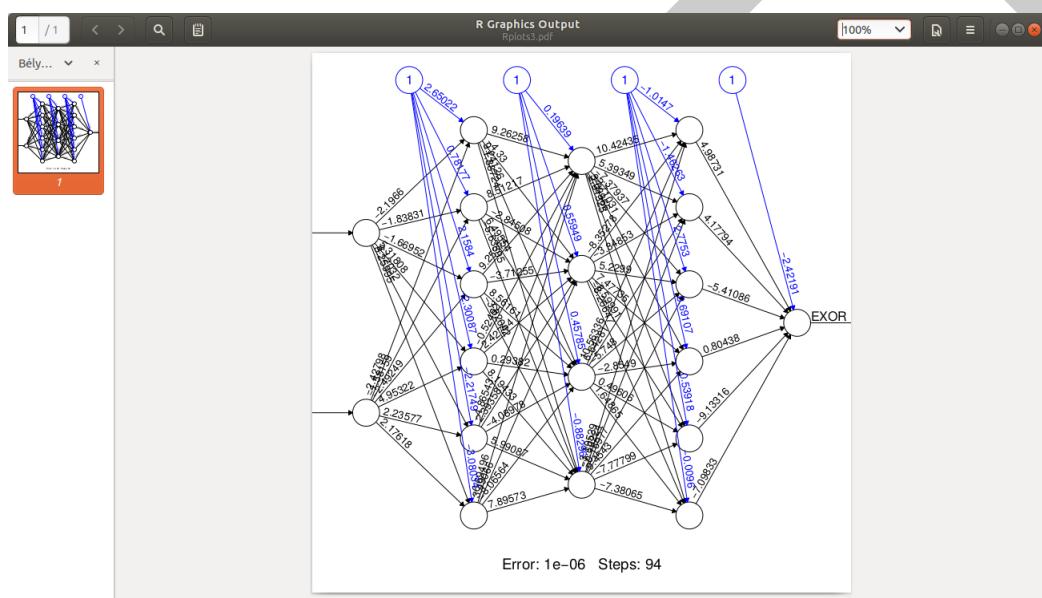


Itt pedig az EXORT tanítatjuk meg a programmal. Az EXOR-nál az EXOR értéke akkor 1, ha az a1 és a2 értéke 1,0 vagy 0,1 . Ha mind akét érték 0,0 vagy 1,1 akkor az EXOR értéke 0 lesz. Ezt a tanulási mintát az exor.data-ban mentjük el. És a tanulás pont úgyanúgy van mint a fentiekben. A képen láthatjuk, hogy

a program nem tanulta meg amit kell, ugyanis az eredmények hibásak. A kulcs abban van, hogy a rejtett neutronok értéke 0. A következőben nézzük meg a megoldását.

```
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
EXOR    <- c(0,1,1,0)
exor.data <- data.frame(a1, a2, EXOR)
nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=c(6, 4, 6), linear. ←
  output=FALSE, stepmax = 1e+07, threshold = 0.000001)
plot(nn.exor)
compute(nn.exor, exor.data[,1:2])
```

Itt anyiban változtattunk, hogy a rejtett neutronoknak létrehoztunk 3 réteget, a rétegek értékei 6,4,6. Ahogy a képen is látszik, az eredmény így helyes.



4.6. Hiba-visszaterjesztéses perceptron

C++

A perceptron a mesterséges intelligenciának olyan, mint az agynak a neuron. A program képes feldolgozni és megtanulni a bemenetet, ami 0,1 ből áll.

Forrás:<https://youtu.be/XpBnR31BRJY>

Kód:

```
#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, 1);
double* image = new double[size];
for(int i {0}; i<png_image.get_width(); ++i)
    for(int j {0}; j<png_image.get_height(); ++j)
        image[i*png_image.get_width()+j] = png_image[i][j].red;
double value = (*p) (image);
std::cout << value << std::endl;
delete p;
delete [] image;
}
```

A kód magyarázata: két headere van szükségünk az "mlp.hpp" és a "png++/png.hpp" -re, ezek a megjelenítés miatt kellenek nekünk és ebbe van a perceptron elve is. A fő függvényünk elején lefoglaljuk a tárhelyet a képnak és megadjuk a méreteit. Következik a perceptron létrehozása és a megfelelő értékek hozzá adása. A "double* image = new double[size];" sorral a végén létrehozunk egy size méretű képet és utánna feltöljük a megadott képpel. A delete parancsokkal töröljük a perceptronról és a képet.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Mandelbrot halmaz egy halmaz a komplex számsíkon. Nevét Benoit Mandelrol kapta, aki megfogalmazta a fraktálok fogalmát (A fraktálok komplex alakzatok).

Forrás:<https://sourceforge.net/p/udprog/code/ci/master/tree/source/kezdo/elsocpp/mandelbrot/mandelbrot.cpp>

Kód:

```
#include <stdio.h>
#include <stdlib.h>
#include <png.h>
#include <sys/times.h>
#include <libpng16/png.h>

#define SIZE 600
#define ITERATION_LIMIT 32000

void mandel (int buffer[SIZE][SIZE]) {

    clock_t delta = clock ();
    struct tms tmsbuf1, tmsbuf2;
    times (&tmsbuf1);

    float a = -2.0, b = .7, c = -1.35, d = 1.35;
    int width = SIZE, height = SIZE, iterationLimit = ITERATION_LIMIT;

    float dx = (b - a) / width;
    float dy = (d - c) / height;
    float reC, imC, reZ, imZ, newReZ, newImZ;

    int iteration = 0;

    for (int j = 0; j < height; ++j)
    {
        for (int k = 0; k < width; ++k)
```

```
{  
  
    reC = a + k * dx;  
    imC = d - j * dy;  
  
    reZ = 0;  
    imZ = 0;  
    iteration = 0;  
  
    while (reZ * reZ + imZ * imZ < 4 && iteration < iterationLimit)  
    {  
        newReZ = reZ * reZ - imZ * imZ + reC;  
        newImZ = 2 * reZ * imZ + imC;  
        reZ = newReZ;  
        imZ = newImZ;  
  
        ++iteration;  
    }  
  
    buffer[j][k] = iteration;  
}  
}  
  
times (&tmsbuf2);  
printf("%ld\n", tmsbuf2.tms_utime - tmsbuf1.tms_utime  
      + tmsbuf2.tms_stime - tmsbuf1.tms_stime);  
  
delta = clock () - delta;  
printf("%f sec\n", (float) delta / CLOCKS_PER_SEC);  
  
}  
  
int main (int argc, char *argv[])  
{  
  
    if (argc != 2)  
    {  
        printf("Hasznalat: ./mandelpng fajlnev\n");  
        return -1;  
    }  
    FILE *fp = fopen(argv[1], "wb");  
    if (!fp) return -1;  
  
    png_structp png_ptr = png_create_write_struct (PNG_LIBPNG_VER_STRING, ←  
          NULL, NULL, NULL);  
  
    if (!png_ptr)  
        return -1;  
    png_infop info_ptr = png_create_info_struct(png_ptr);
```

```
if (!info_ptr)
{
    png_destroy_write_struct (&png_ptr, (png_infopp) NULL);
    return -1;
}
if (setjmp(png_jmpbuf(png_ptr)))
{
    png_destroy_write_struct (&png_ptr, &info_ptr);
    fclose(fp);
    return -1;
}
png_init_io(png_ptr, fp);

png_set_IHDR(png_ptr, info_ptr, SIZE, SIZE,
            8, PNG_COLOR_TYPE_RGB, PNG_INTERLACE_NONE,
            PNG_COMPRESSION_TYPE_BASE, PNG_FILTER_TYPE_BASE);

png_text title_text;
title_text.compression = PNG_TEXT_COMPRESSION_NONE;
title_text.key = "Title";
title_text.text = "Mandelbrot halmaz";
png_set_text(png_ptr, info_ptr, &title_text, 1);

png_write_info(png_ptr, info_ptr);

png_bytеп row = (png_bytеп) malloc(3 * SIZE * sizeof(png_byte));
int buffer[SIZE][SIZE];
mandel(buffer);

for (int j = 0; j < SIZE; ++j)
{
    for (int k = 0; k < SIZE; ++k)
    {
        row[k*3] = (255 - (255 * buffer[j][k]) / ITERATION_LIMIT);
        row[k*3+1] = (255 - (255 * buffer[j][k]) / ITERATION_LIMIT);
        row[k*3+2] = (255 - (255 * buffer[j][k]) / ITERATION_LIMIT);
        row[k*3+3] = (255 - (255 * buffer[j][k]) / ITERATION_LIMIT);
    }
    png_write_row(png_ptr, row);
}
png_write_end(png_ptr, NULL);

printf("%s mentve\n", argv[1]);
```

}

A png.h headerre van szükségünk ahoz hogy png-t tudjunk kezelni. Ez alapból nincs meg a gépen, ezért először is le kell töltenünk az internetről egy fájlt ami tartalmazza a headert. Miután ezt letöltöttük, még telepíteni kell a libpng könyvtárat az alábbi módon: "sudo apt-get install libpng++-dev".

Az include-ról kicsit lejebb, bővebben kifejtve beszélek majd. A kódot állandók definiálásával kezdjük, ilyen lesz a kép maximum szélessége, magassága. Az első függvény fogja nekünk legenerálni a képet. A "png" csomagot használjuk ehez. Létrehozunk egy üres pngt ami 500x500 pixel ((500X500 as mátrix)). A forcikluson belül rgb színkóddal határozzuk meg a színes pixeleket. és a "image.write" a képet kiküldjük a kimenetrre egy adott névvel. ez a függvény a fő függvény legalján lesz meghívva. A következő egy struktúra amiben 2 double tipusú változót deklarálunk , ez a komplex számoknak a struktúrája. Ezután a fő függvényben létrehozunk egy tömböt ami 500x500 elemű. Ezekhez az állandókat használjuk. 3 egész tipusú deklarálása után 2 double változót deklarálunk a "dx" és "dy" amivel a pixeleket fogunk meghatározni. A következő sorban lefoglaljuk a helyet c, z, zuj változóknak, utánna elvégezzük a számításokat és beletesszük azokat a tömbe és meghívjuk a függvényt amivel generáljuk.

```
for (int j = 0; j < height; ++j)
{
    for (int k = 0; k < width; ++k)
    {

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

        reZ = 0;
        imZ = 0;
        iteration = 0;

        while (reZ * reZ + imZ * imZ < 4 && iteration < iterationLimit)
        {
            newReZ = reZ * reZ - imZ * imZ + reC;
            newImZ = 2 * reZ * imZ + imC;
            reZ = newReZ;
            imZ = newImZ;

            ++iteration;
        }

        buffer[j][k] = iteration;
    }
}
```

A fenti részlet számolja ki a komplex számot. Erre azért van szükség, mert nem használjuk a komplex osztályt. A lenti képen találhatjuk meg a kimeneti fájlt, amit úgy kapunk meg, ha program megírása után így kompiláljuk: gcc -o mandelhalmaz mandelhalmaz.c -lpng. A futatáshoz pedig ./mandelhalmaz mandelh2.png parancsot használjuk, ahol a második rész a kimeneti fájl neve.



5.2. A Mandelbrot halmaz a std::complex osztályval

Itt a feladat ugyan az mint az előző pontban. A különbség, hogy itt most használjuk a complex header-t. Ahogy említve volt az előző feladatban is, a komplex osztály segítségével egy teljes struktúrát sporolhatunk meg.

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

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

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

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

    if ( argc == 9 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
        a = atof ( argv[5] );
        b = atof ( argv[6] );
        c = atof ( argv[7] );
        d = atof ( argv[8] );
    }
    else
```

```
{  
    std::cout << "Használat: ./complex fajlnev szelesség magasság n a b c ←  
        d" << std::endl;  
    return -1;  
}  
  
png::image < png::rgb_pixel > kep ( szelesség, magasság );  
  
double dx = ( b - a ) / szelesség;  
double dy = ( d - c ) / magasság;  
double reC, imC, reZ, imZ;  
int iteráció = 0;  
  
std::cout << "Számítás\n";  
  
for ( int j = 0; j < magasság; ++j )  
{  
  
    for ( int k = 0; k < szelesség; ++k )  
    {  
        reC = a + k * dx;  
        imC = d - j * dy;  
        std::complex<double> c ( reC, imC );  
  
        std::complex<double> z_n ( 0, 0 );  
        iteráció = 0;  
  
        while ( std::abs ( z_n ) < 4 && iteráció < iterációsHatar )  
        {  
            z_n = z_n * z_n + c;  
  
            ++iteráció;  
        }  
  
        kep.set_pixel ( k, j,  
                        png::rgb_pixel ( iteráció%255, (iteráció*iteráció ←  
                            )%255, 0 ) );  
    }  
  
    int százalek = ( double ) j / ( double ) magasság * 100.0;  
    std::cout << "\r" << százalek << "%" << std::flush;  
}  
  
kep.write ( argv[1] );  
std::cout << "\r" << argv[1] << " mentve." << std::endl;  
}
```

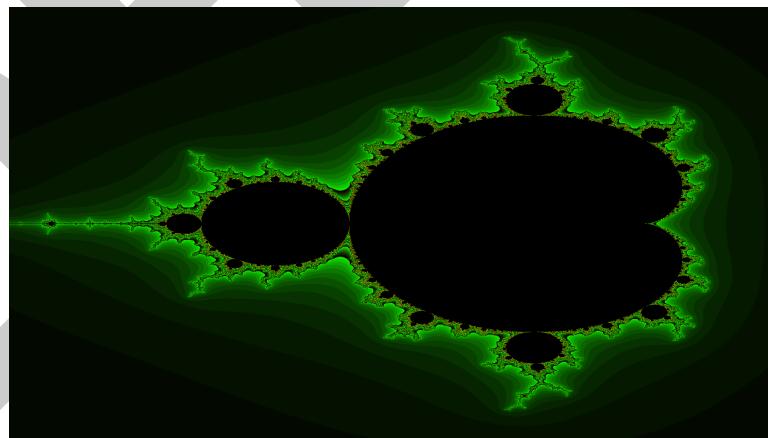
A fő függvényben dekralálunk 2 változót, ha argumentumként jól adjuk meg ezeket, akkor ezeket átadja a változóknak, ha nem jól adjuk meg, akkor kiírjuk, hogy kell helyesen használni. Ezek után megadjuk a szé-

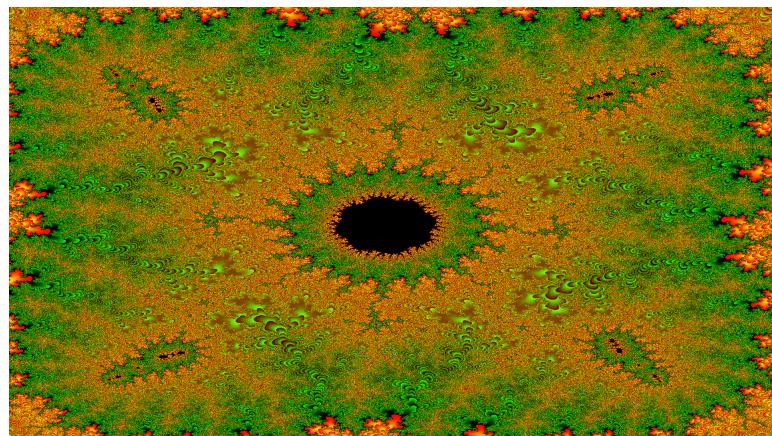
lességet és a magasságot, ami ebbe az esetbe FullHD és az iterácoós határt. Továbbá létrehozunk változókat, amik a kép elkészítéséhez kellenek majd. Az if fügvény vizsgálja meg, az argumentum érvényességét, illetve itt adja át az előbb említett értékeket. Az else ág a rossz esetén, a segítséget írja ki. Ezek után lefoglaljuk a helyet a képnak. A dx, dy-hez hozzá rendeljük a megfelelő változókat. A forciklusban végig megyünk minden elemen és megadjuk a c változó értékét. Ekkor használjuk a complex-et, while ciklusban végezzük a számításokat, utánna rgb kóddal a pixeleket kiszinezzük.

A futtatáshoz szükségünk lesz a -lpng kapcsolóra.

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

A fenti kód kommentárba helyezésével, elérhetjük, hogy a program ne kommunikáljon a felhasználóval, azaz az értékeket csak a forráskódban lehet változtatni és az előre meghatározott számokkal fog dolgozni. Ezzel egyszerűsíthető a program a kezdő felhasználók számára. A következő két képen láthatjuk majd a program végeredményét az alapértelmezett beépített értékekkel, illetve egy képet, ami felhasználó által megadott értékkel lett elkészítve.





5.3. Biomorfok

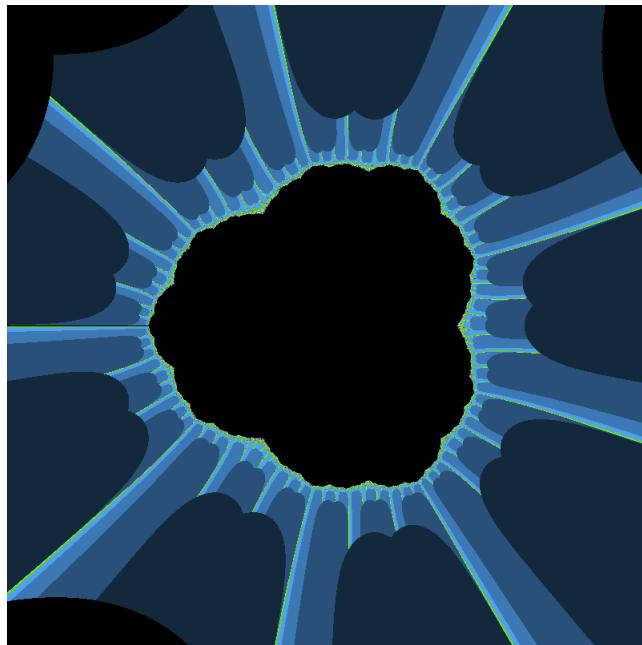
A biomorf program a mandelbrot programkódját vesszi alapul. A mandelbrot halmaz tarttarlmazza az összes ilyen halmazt. A program ugyanúgy bekéri a megfelelő bemeneteket, ha nem jó akkor kiírja. Ha jó, akkor a megfelelő változók megkapják a megfelelő értékeket. Ezután történik a kép létrehozása. Ugyan úgy megkapja a dx és dy az értéket. Aztán pedig a komplex számokat gozzuk létre. Megint végig megy a program minden ponton és ahol kell használjuk az rgb kódos színezést. A legvégén pedig kiküldjük a képet a kimenetre.

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

Kód:

```
#include <iostream>
#include "png++/png.hpp"
#include <complex>
int
main ( int argc, char *argv[] )
{
    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double xmin = -1.9;
    double xmax = 0.7;
    double ymin = -1.3;
    double ymax = 1.3;
    double reC = .285, imC = 0;
    double R = 10.0;
    if ( argc == 12 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
        xmin = atof ( argv[5] );
        xmax = atof ( argv[6] );
        ymin = atof ( argv[7] );
        ymax = atof ( argv[8] );
```

```
reC = atof ( argv[9] );
imC = atof ( argv[10] );
R = atof ( argv[11] );
}
else
{
    std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c ←
        d reC imC R" << std::endl;
    return -1;
}
png::image<png::rgb_pixel> kep ( szelesseg, magassag );
double dx = ( xmax - xmin ) / szelesseg;
double dy = ( ymax - ymin ) / magassag;
std::complex<double> cc ( reC, imC );
std::cout << "Szamitas\n";
for ( int y = 0; y < magassag; ++y )
{
    for ( int x = 0; x < szelesseg; ++x )
    {
        double rez = xmin + x * dx;
        double imZ = ymax - y * dy;
        std::complex<double> z_n ( rez, imZ );
        int iteracio = 0;
        for (int i=0; i < iteraciosHatar; ++i)
        {
            z_n = std::pow(z_n, 3) + cc;
            if (std::real ( z_n ) > R || std::imag ( z_n ) > R)
            {
                iteracio = i;
                break;
            }
        }
        kep.set_pixel ( x, y,
                        png::rgb_pixel ( (iteracio*20)%255, (iteracio ←
                            *40)%255, (iteracio*60)%255 ) );
    }
    int szazalek = ( double ) y / ( double ) magassag * 100.0;
    std::cout << "\r" << szazalek << "%" << std::flush;
}
kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}
```



5.4. A Mandelbrot halmaz CUDA megvalósítása

Továbbra is a mandelbrot halmazzal foglalkozunk, de mi is az a cuda? A CUDA az Nvidia videókártyáknak egy párhuzamos számításokat segítő technológia. Használható C és C++ nyelveknél is. Ezen technika segítségével fogjuk felgyorsítani a kép létrehozását. Ehez szükségünk lesz egy Nvidia videókártyára ami rendelkezik CUDA-val. Továbbá telepítenünk kell. A kód kiterjesztése ".cu"

Megoldás forrása:https://progpater.blog.hu/2011/03/27/a_parhuzamossag_gyonyorkodtet

Kód:

```
#include <pngpp/image.hpp>
#include <pngpp/rgb_pixel.hpp>
#include <sys/times.h>
#include <iostream>
#define MERET 600
#define ITER_HAT 32000
__device__ int
mandel (int k, int j)
{
    float a = -2.0, b = .7, c = -1.35, d = 1.35;
    int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, reZ, imZ, ujrez, ujimZ;
    int iteracio = 0;
    reC = a + k * dx;
    imC = d - j * dy;
    reZ = 0.0;
    imZ = 0.0;
    iteracio = 0;
```

```
while (reZ * reZ + imZ * imZ < 4 && iteracio < iteracionsHatar)
{
    // z_{n+1} = z_n * z_n + c
    ujreZ = reZ * reZ - imZ * imZ + reC;
    ujimZ = 2 * reZ * imZ + imC;
    reZ = ujreZ;
    imZ = ujimZ;
    ++iteracio;
}
return iteracio;
}
/*
__global__ void
mandelkernel (int *kepadat)
{
int j = blockIdx.x;
int k = blockIdx.y;
kepadat[j + k * MERET] = mandel (j, k);
}
*/
__global__ void
mandelkernel (int *kepadat)
{
int tj = threadIdx.x;
int tk = threadIdx.y;
int j = blockIdx.x * 10 + tj;
int k = blockIdx.y * 10 + tk;
kepadat[j + k * MERET] = mandel (j, k);
}
void
cudamandel (int kepadat [MERET] [MERET])
{
int *device_kepadat;
cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));
dim3 grid (MERET / 10, MERET / 10);
dim3 tgrid (10, 10);
mandelkernel <<< grid, tgrid >>> (device_kepadat);
cudaMemcpy (kepadat, device_kepadat,
            MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost);
cudaFree (device_kepadat);
}
int
main (int argc, char *argv[])
{
clock_t delta = clock ();
struct tms tmsbuf1, tmsbuf2;
times (&tmsbuf1);
if (argc != 2)
{
    std::cout << "Hasznalat: ./mandelpngc fajlnev";
}
```

```
    return -1;
}

int kepadat[MERET][MERET];
cudamandel(kepadat);
png::image<png::rgb_pixel> kep(MERET, MERET);
for (int j = 0; j < MERET; ++j)
{
    for (int k = 0; k < MERET; ++k)
    {
        kep.set_pixel(k, j,
                      png::rgb_pixel(255 -
                                     (255 * kepadat[j][k]) / ITER_HAT,
                                     255 -
                                     (255 * kepadat[j][k]) / ITER_HAT,
                                     255 -
                                     (255 * kepadat[j][k]) / ITER_HAT));
    }
}
kep.write(argv[1]);
std::cout << argv[1] << " mentve" << std::endl;
times(&tmsbuf2);
std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
+ tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;
delta = clock() - delta;
std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << std::endl;
}
```

Nézzük a kódot. Az fejlécek alatt két állandót definiálunk, a kép méretét és az iterációs határt. A következő lépés a Mandelbrot halmaz létrehozása. Ezt egy függvényel hozzuk létre. A függvény előtt jelezzük, hogy a számításokat Cudával végezzük majd a fordításnál. A függvényen belül deklarálunk float tipusú változókat a számításokhoz. A matematikai számítás ugyan az mint az 5.1 feladatban, szóval ezt nem fejtem ki most. A következő függvény előtt nem "`__device__`" jelzés van hanem "`__global__`". Ezzel szintén azt jelezzük, hogy a Cuda fogja végezni a számítást. A "threadIdx" jelzi az aktuális szálat és a "blockIdx", hogy melyik blokban folyik a számítás. A kép értékeit a j és a k változókban tároljuk el. Ezt a két értéket fogja kapni az előző függvény. A következő függvény a cudamandel(). Ez egy Méret x Méret azaz 600x600-as tömböt kap. Deklarálunk egy mutatót és a Malloc segítségével lefoglaljuk a megfelelő tárhelyet és a mutató ide fog mutatni. Itt hozzuk létre a megfelelő blokkokat. A végén a tárhelyet felszabadítjuk. A fő függvényünkben sem történik nagy változás. Egyből egy idő méréssel kezdünk. Lemérjük mennyi időbe telik a gépnek, hogy megalkossa a képet. Utánna deklaráljuk a tömböt, meghívjuk a cudamandel() függvényt és már az ismert módon létrehozzuk a képet.

A kódot az "nvcc" fordítóval fordítjuk, le kell tölteni, ehez a gép ad segítséget. A következőképpen fordítjuk: "nvcc mandelpngc_60x60_100.cu -lpng16 -O3 -o mandelpngc". Miután fordítottuk utánna futtatjuk. Ha egymás mellé tesszük a Cudas és a nem kudás képalkotást, láthatjuk, hogy a kép elkészítési ideje a cudásnál sokkal gyorsabb.

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

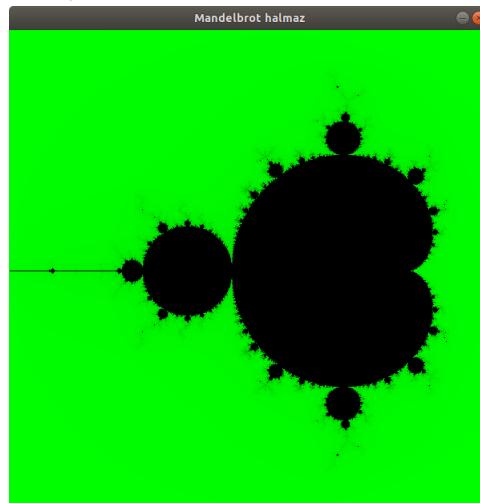
A program azt fogja csinálni, hogy létrejön nekünk egy mandelbrot halmaz és az egérrel képesek vagyunk belenagyítani akár a végtelenségig a halmazba.

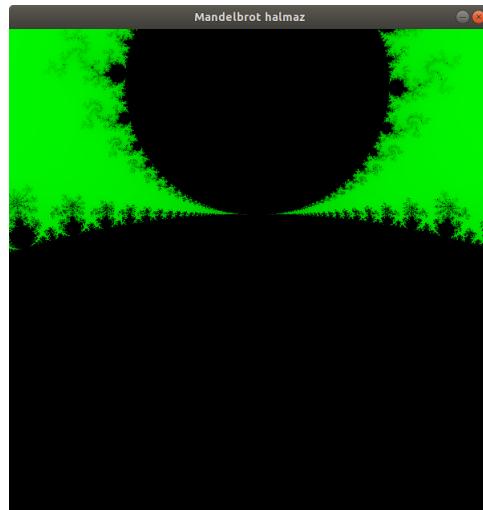
Kód:

```
#include<QApplication>
#include "frakablak.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Frakablak w1,
    w2(-.08292191725019529, -.082921917244591272,
        -.9662079988595939, -.9662079988551172, 1200, 3000),
    w3(-.08292191724880625, -.0829219172470933,
        -.9662079988581493, -.9662079988563615, 1200, 4000),
    w4(.14388310361318304, .14388310362702217,
        .6523089200729396, .6523089200854384, 1200, 38655);
    w1.show();
    w2.show();
    w3.show();
    w4.show();
    return a.exec();
}
```

Ez a program nem elég önmagában, több forrásra van szükséünk. Ilyen például a frakablak.h header. Egy mappába össze kell szednünk az összes forrást és telepítenünk kell ezt: "sudo apt-get install libqt4-dev". A qmake -project parancsal létrehozunk egy .pro fájlt. Ebbe meg kell adnunk a QT+=Widgets parancsot a megfelelő helyre. Ez létrehoz egy fájlokat .o kiterjesztéssel és egy makefilet, ezek után make parancsal létrehozzuk a nagyítót. Ezek után kész is a programunk. A fraksal.cpp-ben készül el az ábránk amit majd nagyítani fogunk. Az rgb pixel színezést azonban már a frakablak végzi.

Forrás: https://progpater.blog.hu/2011/03/26/kepes_egypercesek





5.6. Mandelbrot nagyító és utazó Java nyelven

Ebben a feladatban az 5.5 feladatot fogjuk megírni Java-ban. A program lényege itt is az, hogy a mandelbrothalmazba belenagyítunk.

A program elején létrehozzuk a Mandelbrot halmazt. Ehez az extends szóval hozzá kapcsoljuk a Mandelbrothalmazt építő java kódunkat. A mousePressed() függvényel megadjuk a programnak az egér által kijelölt kordinátákat. Ezután A kijelölt területen újraszámoljuk a halmazt. Majd feldolgozza a létre jött kép szélét és magasságát. A pillanatfelvétel() függvényel egy pillanatfelvételt készítünk. A függvényen belül elnevezzük a tartomány szerint és egy png formátumú képet készítünk a pillanatfelvételből. A nagyítás során láthatunk egy segítő négyzetet, ezt a paint() függvényel hozzuk létre.

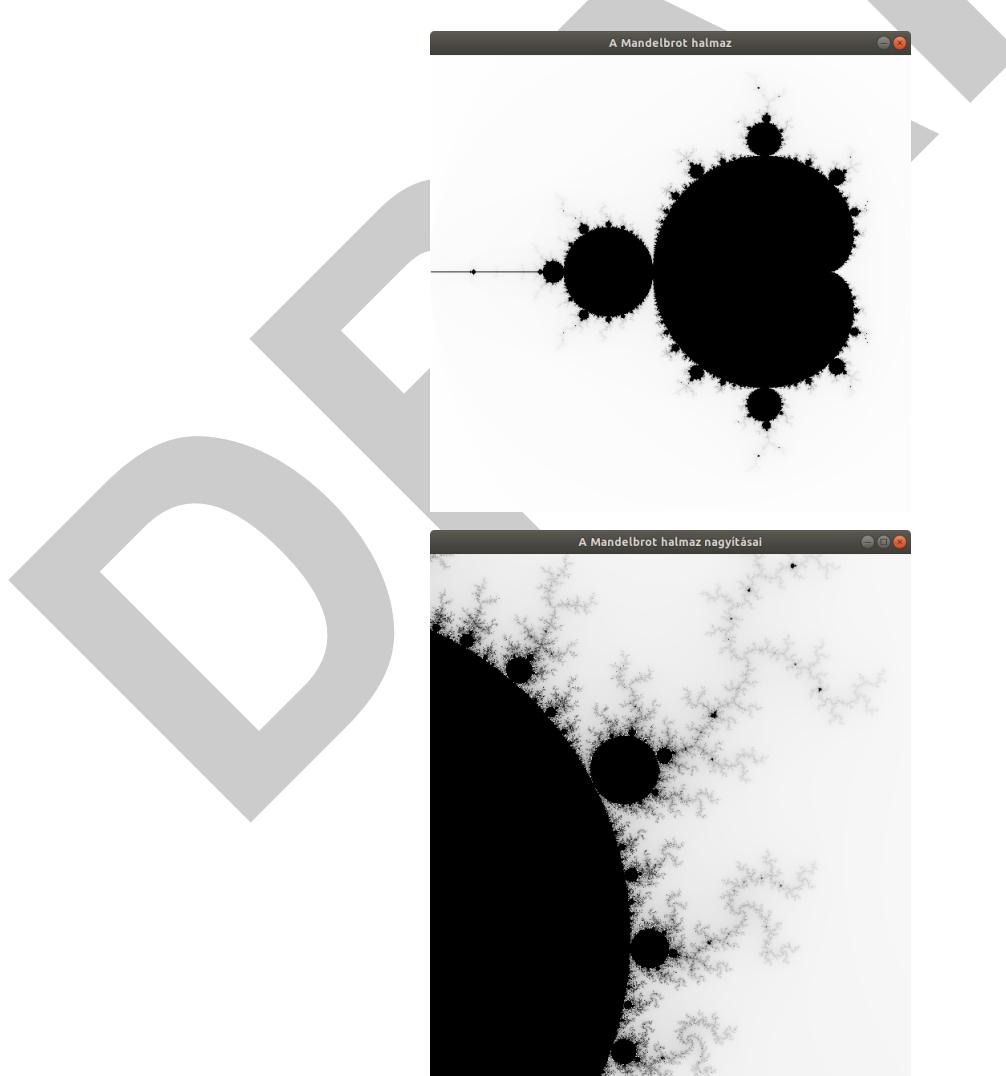
A kód forrása:<https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html>

```
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {  
    private int x, y;  
    private int mx, my;  
    public MandelbrotHalmazNagyító(double a, double b, double c, double d,  
        int szélesség, int iterációsHatár) {  
        super(a, b, c, d, szélesség, iterációsHatár);  
        setTitle("A Mandelbrot halmaz nagyításai");  
        addMouseListener(new java.awt.event.MouseAdapter() {  
            public void mousePressed(java.awt.event.MouseEvent m) {  
                x = m.getX();  
                y = m.getY();  
                mx = 0;  
                my = 0;  
                repaint();  
            }  
            public void mouseReleased(java.awt.event.MouseEvent m) {  
                double dx = (MandelbrotHalmazNagyító.this.b  
                    - MandelbrotHalmazNagyító.this.a)  
                    /MandelbrotHalmazNagyító.this.szélesség;  
                double dy = (MandelbrotHalmazNagyító.this.d  
                    - MandelbrotHalmazNagyító.this.c)
```

```
        /MandelbrotHalmazNagyító.this.magasság;
    new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+←
        x*dx,
        MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
        MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
        MandelbrotHalmazNagyító.this.d-y*dy,
        600,
        MandelbrotHalmazNagyító.this.iterációsHatár);
    }
});
addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
    public void mouseDragged(java.awt.event.MouseEvent m) {
        mx = m.getX() - x;
        my = m.getY() - y;
        repaint();
    }
});
}
public void pillanatfelvétel() {
    java.awt.image.BufferedImage mentKép =
        new java.awt.image.BufferedImage(szélesség, magasság,
            java.awt.image.BufferedImage.TYPE_INT_RGB);
    java.awt.Graphics g = mentKép.getGraphics();
    g.drawImage(kép, 0, 0, this);
    g.setColor(java.awt.Color.BLUE);
    g.drawString("a=" + a, 10, 15);
    g.drawString("b=" + b, 10, 30);
    g.drawString("c=" + c, 10, 45);
    g.drawString("d=" + d, 10, 60);
    g.drawString("n=" + iterációsHatár, 10, 75);
    if(számításFut) {
        g.setColor(java.awt.Color.RED);
        g.drawLine(0, sor, getWidth(), sor);
    }
    g.setColor(java.awt.Color.GREEN);
    g.drawRect(x, y, mx, my);
    g.dispose();
    StringBuffer sb = new StringBuffer();
    sb = sb.delete(0, sb.length());
    sb.append("MandelbrotHalmazNagyitas_");
    sb.append(++pillanatfelvételszámláló);
    sb.append("_");
    sb.append(a);
    sb.append("_");
    sb.append(b);
    sb.append("_");
    sb.append(c);
    sb.append("_");
    sb.append(d);
    sb.append(".");
    sb.append("png");
}
```

```
try {
    javax.imageio.ImageIO.write(mentKép, "png",
        new java.io.File(sb.toString()));
} catch(java.io.IOException e) {
    e.printStackTrace();
}
}

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



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.

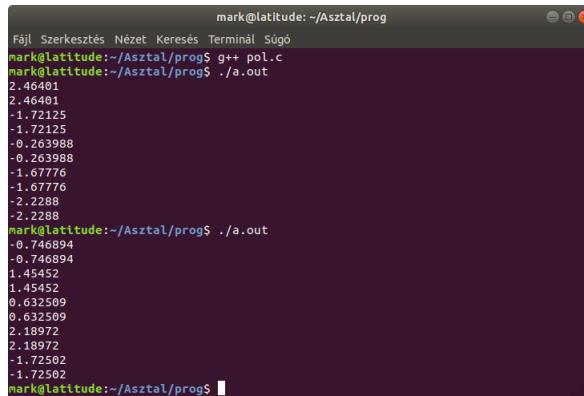
```
#include <iostream>
#include <math.h>
#include <pthread.h>
#include <ctime>

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

double PolarGen::kovetkezo ()
{
    if (nincsTarolt)
    {
        double u1, u2, v1, v2, w;
        do
```

```
{  
    u1= std::rand() / (RAND_MAX +1.0);  
    u2= std::rand() / (RAND_MAX +1.0);  
    v1=2*u1-1;  
    v2=2*u1-1;  
    w=v1*v1+v2*v2;  
}  
while (w>1);  
double r =std::sqrt ((-2 * std::log (w)) /w);  
tarolt=r*v2;  
nincsTarolt =!nincsTarolt;  
return r* v1;  
}  
else  
{  
    nincsTarolt =!nincsTarolt;  
    return tarolt;  
}  
}  
int main (int argc, char **argv)  
{  
    PolarGen pg;  
    for (int i= 0; i<10;i++)  
        std::cout<<pg.kovetkezo ()<< std::endl;  
    return 0;  
}
```

Ez a program véletlenszerűen fog számokat generálni nekünk. Azért csak véletlenszerűen, mert a véletlent nem lehet generálni, már ha egyáltalán úgy gondoljuk, hogy létezik véletlen. Ezt egy osztályon belül fogjuk kivitelezni. Az osztály neve a PolarGen-t kapta. Két részre tudjuk bontani. Van egy nyilvános és egy privát. A nyilvános részhez hozzá tudunk férfi, viszont a privát részt, csak az osztályon belül tudjuk meghívni. Az osztály elején egyből ott van a konstruktor ezt onnan tudjuk felismerni, hogy ugyan úgy hívjuk ahogyan az osztályt is. Ebben kezdő értékeket tudunk adni és egy objektum létrehozásával egyből lefut. Esetünkben most a "nincsTarolt" privát változó értékét fogja "True"-ra állítani és az strand is itt lesz, ami a véletlenszerű szám generálásához kell. Utána van a destruktur ami ugyan úgy néz ki mint a konstruktor csak előtte van '~' jel. Ez a program végén fog lefutni. Ebben felszabadítjuk a memóriát. A privát részben létrehozunk egy logikai és egy double típusú változót. A kovetkezo() függvény az, amiben a random számokat fogjuk létrehozni. Azt, hogy ezt hogyan végezzük matematikailag, most figyelmenkívül hagyjuk. A main függvényben meghívunk egy osztálytípusú változót. Ez fogja beindítani a konstruktort. Utána pedig egy forciklusban tíz véletlen szádot íratunk ki.



```
mark@latitude:~/Asztal/prog$ g++ pol.c
mark@latitude:~/Asztal/prog$ ./a.out
2.46401
1.72125
1.72125
-0.263988
-0.263988
-1.67776
-1.67776
-2.2288
-2.2288
mark@latitude:~/Asztal/prog$ ./a.out
-0.746894
-0.746894
1.45452
1.45452
0.632509
0.632509
2.18972
2.18972
-1.72502
-1.72502
mark@latitude:~/Asztal/prog$
```

Látható, hogy a meghívás során különböző értékeket kapunk vissza, amelyeket a program kétszer ír ki.

Java:

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

    public PolarGenerator()
    {
        nincsTarolt = true;
    }

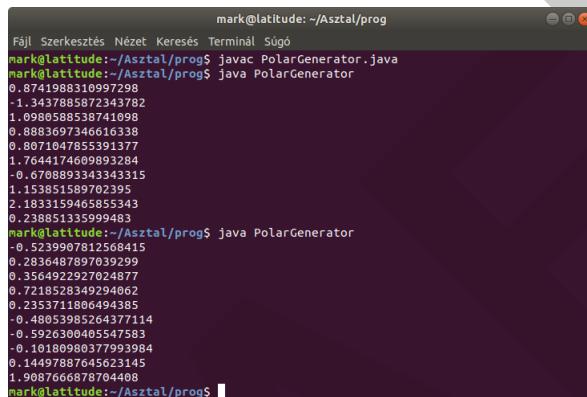
    public double kovetkezo()
    {
        if(nincsTarolt)
        {
            double u1, u2, v1, v2, w;
            do{
                u1 = Math.random();
                u2 = Math.random();
                v1 = 2* u1 -1;
                v2 = 2* u2 -1;
                w = v1*v1 + v2*v2;
            } while (w>1);

            double r = Math.sqrt((-2 * Math.log(w) / w));
            tarolt = r * v2;
            nincsTarolt = !nincsTarolt;
            return r * v1;
        }
        else
        {
            nincsTarolt = !nincsTarolt;
            return tarolt;
        }
    }

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

```
{  
    PolarGenerator g = new PolarGenerator();  
    for (int i = 0; i < 10; ++i)  
    {  
        System.out.println(g.kovetkezo());  
    }  
}
```

Ez az előző program csak javaban megírva. A program felépítése sokkal átláthatóbb és egyszerűbb lett. Alapjaiban ugyan úgy működik mint a C++ megfelelője.



A screenshot of a terminal window titled "mark@latitude: ~/Asztal/prog". The window shows the command "javac PolarGenerator.java" being run, followed by the output of the program's execution. The output consists of ten floating-point numbers ranging from approximately -1.34 to 1.76, each representing a polar coordinate value.

```
mark@latitude: ~/Asztal/prog$ javac PolarGenerator.java  
mark@latitude: ~/Asztal/prog$ java PolarGenerator  
0.8741988310997298  
-1.3437885872343782  
1.0980588538741098  
0.8883697346616338  
0.8071047855391377  
1.7644174609893284  
-0.6708893343343315  
1.153851589702393  
2.1833159465855343  
0.238851335999483  
mark@latitude: ~/Asztal/prog$ java PolarGenerator  
-0.5239907812568415  
0.2836487897039299  
0.3564922927924877  
0.7218528349294662  
0.2353711886494385  
-0.480533055264377114  
-0.5926300465547583  
-0.10180980377993984  
0.14497887645623145  
1.9097666878704408  
mark@latitude: ~/Asztal/prog$
```

A képen látható, hogy a Java változatt is különböző értékeket add a két meghívás során.

6.2. LZW

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

A program a bemeneti adatokból egy bináris fát épít. Bármely típusú fa ábrázolható bináris fa segítségével. A bináris fa legfőbb jellemzője az, hogy bármelyik csomópontnak csak legfeljebb két utóda lehet. A bináris fák utódjait megkülönböztetjük aszerint, hogy bal illetve jobb részfák. A fa 0 és 1 számokból épül fel. A kitüntetett elem a gyökér. Innen minden elemet el tudunk érni. A következőben megnézzük, hogy is működik ez. Az eltérés, hog itt nem fő függvény van, hanem minden egy osztály része.

A kód forrása:https://progpater.blog.hu/2011/03/05/labormeres_otthon_avagy_hogyan_dolgozok_fel_egen_pedat/

```
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <math.h>  
typedef struct binfa  
{  
    int ertekek;  
    struct binfa *bal nulla;  
    struct binfa *jobb egy;  
} BINFA, *BINFA_PTR;  
BINFA_PTR  
uj_elem ()
```

```
{  
    BINFA_PTR p;  
    if ((p = (BINFA_PTR) malloc (sizeof (BINFA))) == NULL)  
    {  
        perror ("memoria");  
        exit (EXIT_FAILURE);  
    }  
    return p;  
}  
extern void kiir (BINFA_PTR elem);  
extern void ratlag (BINFA_PTR elem);  
extern void rszoras (BINFA_PTR elem);  
extern void szabadit (BINFA_PTR elem);  
int  
main (int argc, char **argv)  
{  
    char b;  
    BINFA_PTR gyoker = uj_elem ();  
    gyoker->ertek = '/';  
    gyoker->bal nulla = gyoker->jobb_egy = NULL;  
    BINFA_PTR fa = gyoker;  
    while (read (0, (void *) &b, 1))  
    {  
        if (b == '0')  
        {  
            if (fa->bal nulla == NULL)  
            {  
                fa->bal nulla = uj_elem ();  
                fa->bal nulla->ertek = 0;  
                fa->bal nulla->bal nulla = fa->bal nulla->jobb_egy = NULL;  
                fa = gyoker;  
            }  
            else  
            {  
                fa = fa->bal nulla;  
            }  
        }  
        else  
        {  
            if (fa->jobb_egy == NULL)  
            {  
                fa->jobb_egy = uj_elem ();  
                fa->jobb_egy->ertek = 1;  
                fa->jobb_egy->bal nulla = fa->jobb_egy->jobb_egy = NULL;  
                fa = gyoker;  
            }  
            else  
            {  
                fa = fa->jobb_egy;  
            }  
        }  
    }  
}
```

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

extern int max_melyseg, atlagosszeg, melyseg, atlagdb;
extern double szorasosszeg, atlag;
printf ("melyseg=%d\n", max_melyseg-1);

atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
ratlag (gyoker);
atlag = ((double)atlagosszeg) / atlagdb;
atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
szorasosszeg = 0.0;
rszoras (gyoker);
double szoras = 0.0;
if (atlagdb - 1 > 0)
    szoras = sqrt( szorasosszeg / (atlagdb - 1));
else
    szoras = sqrt (szorasosszeg);
printf ("atlag=%f\nszoras=%f\n", atlag, szoras);/*
szabadit (gyoker);
}

int atlagosszeg = 0, melyseg = 0, atlagdb = 0;
void
ratlag (BINFA_PTR fa)
{
    if (fa != NULL)
    {
        ++melyseg;
        ratlag (fa->jobb_egy);
        ratlag (fa->bal nulla);
        --melyseg;
        if (fa->jobb_egy == NULL && fa->bal nulla == NULL)
        {
            ++atlagdb;
            atlagosszeg += melyseg;
        }
    }
}
double szorasosszeg = 0.0, atlag = 0.0;
void
rszoras (BINFA_PTR fa)
{
    if (fa != NULL)
```

```
{  
    ++melyseg;  
    rszoras (fa->jobb_egy);  
    rszoras (fa->bal_nulla);  
    --melyseg;  
    if (fa->jobb_egy == NULL && fa->bal_nulla == NULL)  
    {  
        ++atlagdb;  
        szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));  
    }  
}  
}  
  
int max_melyseg = 0;  
void  
kiir (BINFA_PTR elem)  
{  
    if (elem != NULL)  
    {  
        ++melyseg;  
        if (melyseg > max_melyseg)  
            max_melyseg = melyseg;  
        kiir (elem->jobb_egy);  
        for (int i = 0; i < melyseg; ++i)  
            printf ("---");  
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ↔  
               ,  
               melyseg-1);  
        kiir (elem->bal_nulla);  
        --melyseg;  
    }  
}  
void  
szabadit (BINFA_PTR elem)  
{  
    if (elem != NULL)  
    {  
        szabadit (elem->jobb_egy);  
        szabadit (elem->bal_nulla);  
        free (elem);  
    }  
}
```

Magyarázat: Elsőnek is a szükséges headereket deklaráljuk. Ezek után pedig a Binfánk struktúráját. A struktúra egy egészet tartalmaz aminek a neve "ertek" és 2 mutatóval fog rendelkezni, amiben bal és jobb gyermeket tároljuk majd. Az 1 érték jobbra fog kerülni, a 0 balra. A "typedef"-vel adunk neki egy nevet, amivel a programon belül fogjuk hívni. Ezután az uj_elem függvény következik. Ez fogja nekünk lefoglalni a tárhelyet a memóriában, ami "NULL" kezdőértékkkel fog rendelkezni. Ha nincs memória, akkor hibát dob ki. A végén vissza adja a lefoglalt mutatót. Utánna függvény prototípusokat kapunk, ezek közül a feladatnak megfelelően csak a kiir() és a szabadit() függvényeket fogjuk megvizsgálni. Ugorjunk a main

fő függvényre. Az első egy char típusú változó, ebben fogjuk tárolni ideiglenesen a beolvasott karaktert. Aztán létrehozzuk a gyökérelemet és értékül adunk neki egy karaktert, jelen esetben '/'. A while ciklusban fog zajlani a faépítés. Először is megvizsgálja a beolvasott karaktert. Mindig a gyökér elemtől indul. Ha a beolvasott karakter értéke 0, akkor először megvizsgálja, hogy a gyökérnek vagy az adott csomópontnak van-e bal_nullas gyermeke, ha van, akkor rálép a csomópontra, ha viszont nincs, akkor a gyökérnek vagy az adott csomópontnak létrehoz egy bal_nullas gyermeket. Ha a beolvasott karakter értéke 1, akkor a program ugyan ezen az elven mint a 0-ás értéknél végig vizsgálja, csak a jobb_egyes gyermekkel. Most következik a kiír és a szabadit függvény. A szabadit() függvény egy rekurzív függvény. Törli a memóriából az eltárolt elemeket. A kiir() függvény is rekúrzsiv függvény. Bejárja a fa elemeit.

```
mark@latitude:~/Asztal/prog$ g++ bInfa.c
mark@latitude:~/Asztal/prog$ ./a.out <be.txt
.....1(2)
.....-1(1)
.....1(3)
.....0(2)
.....0(3)
.../()
.....-1(2)
.....-0(1)
.....0(2)
melyseg=3
althag=2,400000
szoras=0,547723
mark@latitude:~/Asztal/prog$
```

A képen látható a futatás után létrehozott binfa kirajzolása és egyébb adatok.

6.3. Fabejárás

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

A fabejárásnak 3 tipusa van, preorder, inorder és postorder.

Kezdjük a preorder fabejárással. Itt minden a gyökérrel kezdi a program a vizsgálatot, aztán a bal oldalt legvégezetül, pedig a jobb oldalt fogja bejárni.

```
//preorder fabejárás:
void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        printf ("%c", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek);
        kiir (elem->bal nulla);
        kiir (elem->jobb_egy);
    }
}
```

Inorder fabejárás: Az inorder fabejárásnál először a bal oldalt vizsgáljuk meg, utánna jön a gyökér és legvégül pedig a jobb oldalt nézzük.

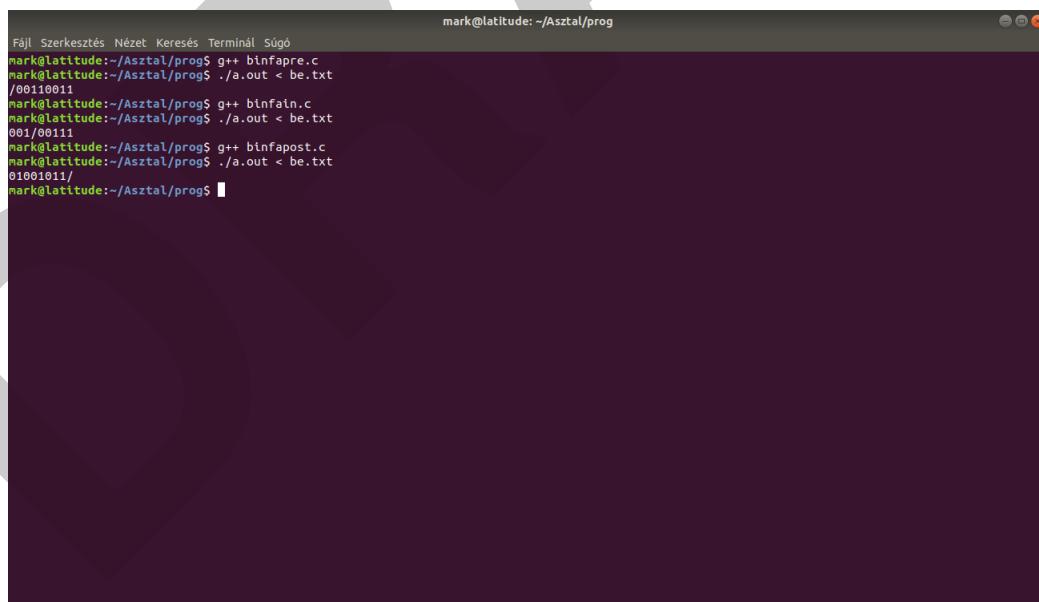
```
//inorder fabejárás:
void
kiir (BINFA_PTR elem)
```

```
{  
    if (elem != NULL)  
    {  
        kiir (elem->bal_nulla);  
        printf ("%c", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek);  
        kiir (elem->jobb_egy);  
    }  
}
```

A postorder fabejárás: Itt először a fa bal oldalát fogja átvizsgálni, aztán a jobb oldalt, legutoljára pedig a gyökeret vizsgáljuk.

```
//postorder fabejárás:  
void  
kiir (BINFA_PTR elem)  
{  
    if (elem != NULL)  
    {  
        kiir (elem->bal_nulla);  
        kiir (elem->jobb_egy);  
        printf ("%c", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek);  
    }  
}
```

Ahogy a kodolásban látszik a három bejárás megvalósítása szinte semmiben sem különbözik. A változás csupán annyi, hogy az elemet kiíró sor hanyadik utasítás az alprogramban. A lenti képen látható lesz, hogy az egyes bejárások során hova kerül a gyökér elem, ami a / karakter lesz.



```
Fájl Szerkesztés Nézet Keresés Terminál Súgó  
mark@latitude:~/Asztal/prog$ g++ binfapre.c  
mark@latitude:~/Asztal/prog$ ./a.out < be.txt  
/00110011  
mark@latitude:~/Asztal/prog$ g++ binfain.c  
mark@latitude:~/Asztal/prog$ ./a.out < be.txt  
00110011  
mark@latitude:~/Asztal/prog$ g++ binfapost.c  
mark@latitude:~/Asztal/prog$ ./a.out < be.txt  
01001011/  
mark@latitude:~/Asztal/prog$
```

6.4. Tag a gyökér

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

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

Az alábbi program a fenti C változatnak lesz úgymond a C++ változata. Az első lépés, hogy ami C-ben struktúra volt, azt átírjuk C++-ban egy osztályba, mivel a C++-ban megtehetjük. Ez az alábbi módon fog kinézni:

```
class LZWBInFa
{
public:
    LZWBInFa (char b = '/') : betu (b), balNulla (NULL), jobbEgy (NULL) ←
    {};
    ~LZWBInFa () {};
    void operator<<(char b)
    {
        if (b == '0')
        {
            // van '0'-s gyermek az aktuális csomópontnak?
            if (!fa->nullasGyermek ()) // ha nincs, csinálunk
            {
                Csomopont *uj = new Csomopont ('0');
                fa->ujNullasGyermek (uj);
                fa = &gyoker;
            }
            else // ha van, arra lépünk
            {
                fa = fa->nullasGyermek ();
            }
        }
        else
        {
            if (!fa->egyesGyermek ())
            {
                Csomopont *uj = new Csomopont ('1');
                fa->ujEgyesGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->egyesGyermek ();
            }
        }
    }
}
```

Ezen belül fogjuk a gyökeret létrehozni és értékül adni neki a '/'-t. A mutatóinak értékét 0-ra állítjuk. Ezek után jön a faépítés a már fentiekben elmagyarázott módon. A program megnézi, hogy 1-est vagy 0 érkezik a bemenetről. Itt azt látjuk, hogy a betétel a << operátorral történik, ez annyiban különbözik a C-ben írt programtól, hogy ez egyből beleteszi a fába a beérkezett karaktert. Egy új csomópontot a "new" szóval tudunk létrehozni, ha szükséges. Ez azért lehetséges mert van egy Csomopont osztályunk (Lásd a lenti programban). A Class csomóponton belül az egyesGyermek() és a nullasGyermek() függvények a gyermekükre mutató pointereket fogják tartalmazni. Az ujNullasGyermek és au ujEgyesGyermek-nek

pedig adunk egy gyermeket ás arra fogja állítani a mutatót. A private részben fogjuk ezeket deklarálni, ez azt jelenti, hogy csak az osztályon belül használhatóak ezek a változók. A legvégén jön a main főfüggvény. Itt deklaráljuk a char típusú változót amibe beolvassunk és innen kerül az osztályokhoz. Végül meghívjuk a kiir és a szabadít függvényeket amire példát az előző programokban találunk. Ugye a kiir()-al kiíratjuk az eredmény és a szabadít()-al pedig felszabadítjuk a lefoglalt merőriát.

```
class Csomopont
{
public:
    Csomopont (char b = '/') : betu (b), balNulla (0), jobbEgy (0) {};
    ~Csonopont () {};
    Csonopont *nullasGyermek () {
        return balNulla;
    }
    Csonopont *egyesGyermek () {
        return jobbEgy;
    }
    void ujNullasGyermek (Csonopont * gy)
    {
        balNulla = gy;
    }
    void ujEgyesGyermek (Csonopont * gy)
    {
        jobbEgy = gy;
    }
private:
    friend class LZWBInFa;
    char betu;
    Csonopont *balNulla;
    Csonopont *jobbEgy;
    Csonopont (const Csonopont &);
    Csonopont & operator=(const Csonopont &);
};

int main ()
{
    char b;
    LZWBInFa binFa;
    while (std::cin >> b)
    {
        binFa << b;
    }
    binFa.kiir ();
    binFa.szabadit ();
    return 0;
}
```

6.5. Mutató a gyökér

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

Vegyük alapul a C++ ban megírt LZWBina-t. Első dolgunk, hogy a fában a gyökér elemet átalakítjuk egy mutatóvá. Azt az alábbi módon fogjuk megcsinálni: A gyökér elem a protected részén van az osztálynak. Itt az eredeti "Csomopont gyoker;" az alábbi módon átírunk:

```
protected:  
Csomopont *gyoker;  
int maxMelyseg;  
double atlag, szoras;
```

Ugye C++ ban a mutatót egy '*'-al jelüljük. Ha most futtatnánk a programot, akkor számtalan hibába ütközünk. Ezeket ki kell javítanunk. A programban így nem a gyökér memóriacímét kell átadnunk (Töröljük az összes referenciajelet a gyokerek előtt) és mivel mutató lett a gyökér így nem '.'-al hiavatkozunk hanem '->'-al. Itt láthatunk példát arra, hogy hogyan:

```
//előtte:  
fa=&gyoker;  
//utánna:  
fa=gyoker;  
  
//előtte:  
szabadit (gyoker.egyesGyermek ());  
szabadit (gyoker.nullasGyermek ());  
//utánna:  
szabadit (gyoker->egyesGyermek ());  
szabadit (gyoker->>nullasGyermek ());  
}
```

Ha mindezek után lefuttatjuk a programunkat az lefordul, azonban futtatáskor szegmentális hibába ütközünk. Ez azért van, ugyanis a gyökér memóriacíme nincs lefoglalva. Ennek a megoldását a konstruktorkban és a destruktorkban fogjuk megalkotni. A konstruktorkban foglaljuk le és a destruktorkba fogjuk törölni a lefoglalt memóriát. Lásd:

```
LZWBInFa ()  
{  
    gyoker= new Csomopont ('/');  
    fa = gyoker;  
}  
~LZWBInFa ()  
{  
    szabadit (gyoker->egyesGyermek ());  
    szabadit (gyoker->>nullasGyermek ());  
    delete(gyoker);  
}
```

6.6. Mozgató szemantika

Megoldás forrása: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/vedes/elso/z3a7.cpp>

Védési videó: <https://www.youtube.com/watch?v=S8bM0rewaTE>

```
LZWBinFa ( const LZWBinFa & regi ) {

    gyoker.ujEgyesGyermek ( masol ( regi.gyoker.egyesGyermek (), regi ←
        .fa ) );
    gyoker.ujNullasGyermek ( masol ( regi.gyoker.nullasGyermek (), ←
        regi.fa ) );

    if ( regi.fa == & ( regi.gyoker ) )
        fa = &gyoker;

}

LZWBinFa ( LZWBinFa && regi ) {

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

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

}
```

```
Csomopont * masol ( Csomopont * elem, Csomopont * regifa ) {

    Csomopont * ujelem = NULL;

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

        ujelem->ujEgyesGyermek ( masol ( elem->egyesGyermek (), ←
            regifa ) );
        ujelem->ujNullasGyermek ( masol ( elem->nullasGyermek (), ←
            regifa ) );

        if ( regifa == elem )
            fa = ujelem;

    }

    return ujelem;
}
```

A forráskóban a binfa működése gyakorlatilag nem változik semmit, ezért arról nem is írnék részletesebben. A mozgató szemantika titka viszont annyiban merül ki, hogy a először is szükségünk van egy másoló

konstruktorra, ami egy olyan függvény lesz, ami megkapja a binfa elemeit és ezeket új memóriacímen új jobb és bal elemekként lementi. Ezzel építünk gyakorlatilag egy új fát, aminek minden eleme megegyezik az eredeti fánknak az elemeivel. Amikor ez megtörténik, régi fánknak a gyökerének a pointereit átállítjuk null pointerekre, ezzel pedig töröltük azoknak az elemeit és már csak az új fánk létezik.

A fánk mozgatását a main-ben a std::move függvénnyel fogjuk elérni, ami magától nem fogja mozgatni a binfánkat, csak akkor, ha ehhez meg van írva már a mozgató konstruktorkunk.



7. fejezet

Helló, Conway!

7.1. Hangyszimulációk

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

Ebben a feladatban egy QT programban fogjuk szimulálni a hangyák mozgását. A való életben a hangyák sose zavarodnak össze, sose torlódnak fel, sőt minél többen vannak annál jobban és gyorsabban mozognak. Továbbá tudjuk, hogy a látásuk nem a legjobb. Tehát a kulcs az egymás közötti kommunikáció, ezt feromonokkal érik el. Ebben a szimulációban mi is úgymond feromonokkal fogjuk a hangyák közötti kommunikációt elérni. A szabály, hogy mindenkor a legerősebb feromonú hártya felé lépünk, a programban látszik, hogy a hanygák feromon csíkokat hagynak maguk után, ami idő elteltével egyre gyengül míg el nem tűnik.

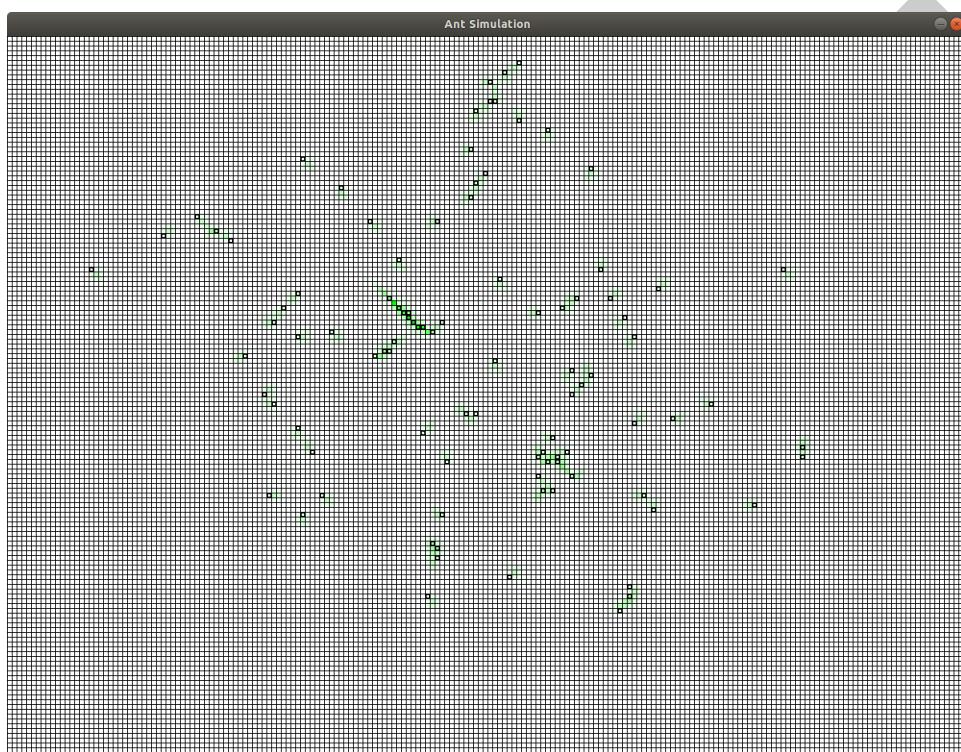
Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

```
#include <QApplication>
#include <QDesktopWidget>
#include <QDebug>
#include <QDateTime>
#include <QCommandLineOption>
#include <QCommandLineParser>
#include "antwin.h"
int main ( int argc, char *argv[] )
{
    QApplication a ( argc, argv );
    QCommandLineOption szeles_opt ( { "w", "szelessseg" }, "Oszlopok (cellákban ←
        ) száma.", "szelesség", "200" );
    QCommandLineOption magas_opt ( { "m", "magasság" }, "Sorok (cellákban) ←
        száma.", "magasság", "150" );
    QCommandLineOption hangyaszam_opt ( { "n", "hangyaszam" }, "Hangyák száma. ←
        ", "hangyaszam", "100" );
    QCommandLineOption sebesseg_opt ( { "t", "sebesseg" }, "2 lépés közötti ←
        idő (millisec-ben).", "sebesseg", "100" );
    QCommandLineOption parolgas_opt ( { "p", "parolgas" }, "A parolgas értéke. ←
        ", "parolgas", "8" );
```

```
QCommandLineOption feromon_opt ( {"f","feromon"}, "A hagyott nyom erteke.", "feromon", "11" );
QCommandLineOption szomszed_opt ( {"s","szomszed"}, "A hagyott nyom erteke a szomszedokban.", "szomszed", "3" );
QCommandLineOption alapertek_opt ( {"d","alapertek"}, "Indulo ertek a cellakban.", "alapertek", "1" );
QCommandLineOption maxcella_opt ( {"a","maxcella"}, "Cella max erteke." , "maxcella", "50" );
QCommandLineOption mincella_opt ( {"i","mincella"}, "Cella min erteke." , "mincella", "2" );
QCommandLineOption cellamerete_opt ( {"c","cellameret"}, "Hany hangya fer egy cellaba.", "cellameret", "4" );
QCommandLineParser parser;
parser.addHelpOption();
parser.addVersionOption();
parser.addOption ( szeles_opt );
parser.addOption ( magas_opt );
parser.addOption ( hangyaszam_opt );
parser.addOption ( sebesseg_opt );
parser.addOption ( parolgas_opt );
parser.addOption ( feromon_opt );
parser.addOption ( szomszed_opt );
parser.addOption ( alapertek_opt );
parser.addOption ( maxcella_opt );
parser.addOption ( mincella_opt );
parser.addOption ( cellamerete_opt );
parser.process ( a );
QString szeles = parser.value ( szeles_opt );
QString magas = parser.value ( magas_opt );
QString n = parser.value ( hangyaszam_opt );
QString t = parser.value ( sebesseg_opt );
QString parolgas = parser.value ( parolgas_opt );
QString feromon = parser.value ( feromon_opt );
QString szomszed = parser.value ( szomszed_opt );
QString alapertek = parser.value ( alapertek_opt );
QString maxcella = parser.value ( maxcella_opt );
QString mincella = parser.value ( mincella_opt );
QString cellameret = parser.value ( cellamerete_opt );
qrand ( QDateTime::currentMSecsSinceEpoch() );
AntWin w ( szeles.toInt(), magas.toInt(), t.toInt(), n.toInt(), feromon.toInt(), szomszed.toInt(), parolgas.toInt(),
           alapertek.toInt(), mincella.toInt(), maxcella.toInt(), cellameret.toInt() );
w.show();
return a.exec();
}
```

Kezdjük az ant.h tartalmazza a hangya tulajdonságait. Hol van az x és y tengelyen és hogy merre mutat az iránya, merre megy. Az antwin-ban pedig a hangyaboly(ants) van. Továbbá ezen belül adjuk meg, hogy

egy cella hánnyal pixelből álljon és hogy az ablak szélessége és magassága mekkora. Forciklus-akkal felépítjük cellákból az ablakot és elhelyezzük benne a hangyákat(azt ant-ből). Új és új hangyák jelennek meg. Ezek megváltoztatják a régebbi hangygák irányát. Az antwin.h tartalmazza a billentyűzet parancsait, például a p vel megállítjuk a folyamatot. Az antheard.cpp tartalmazza a mozgáshoz, törléshez, az új irány megadásához, a hangygák számának eltárolásához szükséges függvényeket. Itt vizsgálja azt is hogy a hangygák száma nő e vagy csökken az idő mulásával.



7.2. Java életjáték

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

Az életjátékot azaz sejtautómatakat először Naumen János vetette fel. A felvetés a gép önreprodukciójának matematikai modellalkotást tartalmazta. A legismertebb modell a John Horton Conway-féle életjáték. Maga a "játék" egy négyzetrácsos mezőn zajlik amin mozognak a sejtek. A sejtek "élete" szabályokhoz van kötve. Megvan adva egy sejt létrejötének, életbenmadardásának vagy elpusztulásának szabálya. Conway erre 3 feltételt szabott meg:

1. szabály: Egy sejt csak úgy éli túl, ha kettő vagy három szomszédja van.
2. szabály: Egy sejt akkor pusztul el, ha kettőnél kevesebb szomszédja van. Ezt elszigetelődésnek hívjuk. A másik eset hogy akkor pusztul el ha háromnál több szomszédja van. Ezt túlnépesedésnek hívjuk.
3. szabály: A harmadik szabály a születésre vonatkozik, és akkor történik meg ha egy cellának a körzetében 3 sejt található.

Ezen 3 szabály meghatározásával kapunk egy önműködő sejtautómatakát. Beleszolásunk csak kezdetben van, utánna a szabályok szerint önállóan működik a program. Mi most külön a sikló-kilövőt fogjuk vizsgálni. Hogy ezt elérjük, rögzítenünk kell adott cellákban sejteket, így létre jön egy "sikló ágyú", ez időközönként "siklókat" fog lőni.

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html?fbclid=IwAR0Gc-Q7v353l_qDrqAd4LfIhWrzTwYnnsTZ5wTpBAhQjwZ63pl2moebOpY

Kód:

```
public class Sejtautomata extends java.awt.Frame implements Runnable {
    public static final boolean ÉLŐ = true;
    public static final boolean HALOTT = false;
    protected boolean[][][] rácsok = new boolean [2][][];
    protected boolean[][] rács;
    protected int rácsIndex = 0;
    protected int cellaSzélesség = 20;
    protected int cellaMagasság = 20;/
    protected int szélesség = 20;
    protected int magasság = 10;
    protected int várakozás = 1000;
    private java.awt.Robot robot;
    private boolean pillanatfelvétel = false;
    private static int pillanatfelvételSzámláló = 0;
    public Sejtautomata(int szélesség, int magasság) {
        this.szélesség = szélesség;
        this.magasság = magasság;
        rácsok[0] = new boolean[magasság][szélesség];
        rácsok[1] = new boolean[magasság][szélesség];
        rácsIndex = 0;
        rács = rácsok[rácsIndex];
        for(int i=0; i<rács.length; ++i)
            for(int j=0; j<rács[0].length; ++j)
                rács[i][j] = HALOTT;
        siklóKilövő(rács, 5, 60);
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent e) {
                setVisible(false);
                System.exit(0);
            }
        });
        addKeyListener(new java.awt.event.KeyAdapter() {
            public void keyPressed(java.awt.event.KeyEvent e) {
                if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K) {
                    cellaSzélesség /= 2;
                    cellaMagasság /= 2;
                    setSize(Sejtautomata.this.szélesség*cellaSzélesség,
                            Sejtautomata.this.magasság*cellaMagasság);
                    validate();
                } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {
                    cellaSzélesség *= 2;
                    cellaMagasság *= 2;
                    setSize(Sejtautomata.this.szélesség*cellaSzélesség,
                            Sejtautomata.this.magasság*cellaMagasság);
                    validate();
                } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
```

```
        pillanatfelvétel = !pillanatfelvétel;
    else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
        várakozás /= 2;
    else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L)
        várakozás *= 2;
    repaint();
}
});
addMouseListener(new java.awt.event.MouseAdapter() {
    int x = m.getX()/cellaSzélesség;
    int y = m.getY()/cellaMagasság;
    rácsok[rácsIndex][y][x] = !rácsok[rácsIndex][y][x];
    repaint();
})
);
addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
    // Vonszolással jelöljük ki a négyzetet:
    public void mouseDragged(java.awt.event.MouseEvent m) {
        int x = m.getX()/cellaSzélesség;
        int y = m.getY()/cellaMagasság;
        rácsok[rácsIndex][y][x] = ÉLŐ;
        repaint();
    }
});
cellaSzélesség = 10;
cellaMagasság = 10;
try {
    robot = new java.awt.Robot(
        java.awt.GraphicsEnvironment.
        getLocalGraphicsEnvironment().
        getDefaultScreenDevice());
} catch(java.awt.AWTException e) {
    e.printStackTrace();
}
setTitle("Sejtautomata");
setResizable(false);
setSize(szélesség*cellaSzélesség,
           magasság*cellaMagasság);
setVisible(true);
new Thread(this).start();
}
public void paint(java.awt.Graphics g) {
    boolean [][] rács = rácsok[rácsIndex];
    for(int i=0; i<rács.length; ++i) { // végig lépked a sorokon
        for(int j=0; j<rács[0].length; ++j) { // s az oszlopok
            if(rács[i][j] == ÉLŐ)
                g.setColor(java.awt.Color.BLACK);
            else
                g.setColor(java.awt.Color.WHITE);
            g.fillRect(j*cellaSzélesség, i*cellaMagasság,
```

```
        cellaSzélesség, cellaMagasság);
g.setColor(java.awt.Color.LIGHT_GRAY);
g.drawRect(j*cellaSzélesség, i*cellaMagasság,
           cellaSzélesség, cellaMagasság);
    }
}
if(pillanatfelvétel) {
    pillanatfelvétel = false;
    pillanatfelvétel(robot.createScreenCapture
        (new java.awt.Rectangle
            (getLocation().x, getLocation().y,
             szélesség*cellaSzélesség,
             magasság*cellaMagasság)));
}
}

public int szomszédokSzáma(boolean [][] rács,
    int sor, int oszlop, boolean állapot) {
int állapotúSzomszéd = 0;
for(int i=-1; i<2; ++i)
    for(int j=-1; j<2; ++j)
        if(!((i==0) && (j==0))) {
            int o = oszlop + j;
            if(o < 0)
                o = szélesség-1;
            else if(o >= szélesség)
                o = 0;

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

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

    return állapotúSzomszéd;
}
public void időFejlődés() {

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

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

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

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

```

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

    while(true) {
        try {
            Thread.sleep(várakozás);
        } catch (InterruptedException e) { }

        időFejlődés();
        repaint();
    }
}
public void sikló(boolean [][] rács, int x, int y) {

    rács[y+ 0][x+ 2] = ÉLŐ;
    rács[y+ 1][x+ 1] = ÉLŐ;
    rács[y+ 2][x+ 1] = ÉLŐ;
    rács[y+ 2][x+ 2] = ÉLŐ;
    rács[y+ 2][x+ 3] = ÉLŐ;

}
public void siklóKilövő(boolean [][] rács, int x, int y) {

    rács[y+ 6][x+ 0] = ÉLŐ;
    rács[y+ 6][x+ 1] = ÉLŐ;
    rács[y+ 7][x+ 0] = ÉLŐ;
    rács[y+ 7][x+ 1] = ÉLŐ;

    rács[y+ 3][x+ 13] = ÉLŐ;

    rács[y+ 4][x+ 12] = ÉLŐ;
    rács[y+ 4][x+ 14] = ÉLŐ;

    rács[y+ 5][x+ 11] = ÉLŐ;
    rács[y+ 5][x+ 15] = ÉLŐ;
    rács[y+ 5][x+ 16] = ÉLŐ;
    rács[y+ 5][x+ 25] = ÉLŐ;
```

```
rács[y+ 6][x+ 11] = ÉLŐ;
rács[y+ 6][x+ 15] = ÉLŐ;
rács[y+ 6][x+ 16] = ÉLŐ;
rács[y+ 6][x+ 22] = ÉLŐ;
rács[y+ 6][x+ 23] = ÉLŐ;
rács[y+ 6][x+ 24] = ÉLŐ;
rács[y+ 6][x+ 25] = ÉLŐ;

rács[y+ 7][x+ 11] = ÉLŐ;
rács[y+ 7][x+ 15] = ÉLŐ;
rács[y+ 7][x+ 16] = ÉLŐ;
rács[y+ 7][x+ 21] = ÉLŐ;
rács[y+ 7][x+ 22] = ÉLŐ;
rács[y+ 7][x+ 23] = ÉLŐ;
rács[y+ 7][x+ 24] = ÉLŐ;

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

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

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

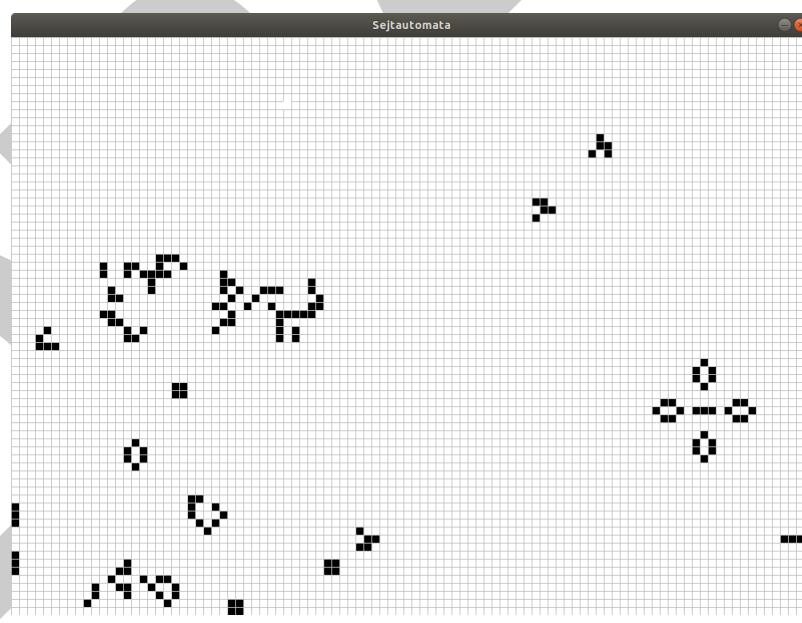
rács[y+ 11][x+ 25] = ÉLŐ;

}

public void pillanatfelvétel(java.awt.image.BufferedImage felvetel) {
    // A pillanatfelvétel kép fájlneve
    StringBuffer sb = new StringBuffer();
    sb = sb.delete(0, sb.length());
    sb.append("sejtautomata");
    sb.append(++pillanatfelvételszámláló);
    sb.append(".png");
    // png formátumú képet mentünk
    try {
        javax.imageio.ImageIO.write(felvetel, "png",
```

```
        new java.io.File(sb.toString()));
    } catch(java.io.IOException e) {
        e.printStackTrace();
    }
}
public void update(java.awt.Graphics g) {
    paint(g);
}
public static void main(String[] args) {
    new Sejtautomata(100, 75);
}
}
```

A program elején megadjuk, hogy egy sejt lehet élő vagy halott. A feladatban 2 rácsfélét használunk, az egyik rács a sejt állapotát fogja tárolni míg a második az egy másdopercel későbbi tulajdonságait. Meghatározzuk az aktuális rácsot a rácsIndex-el. Utánna egy cella magasságát és szélességét, ezt követően pedig, hogy hány cellából álljon a "játék". A következő hogy a az állapotok között mennyi idő teljen el. A függvények közül az első megkapja a méreteket és létrehozza az ablakot. Itt készíti el a 2 rácsot és az indexet is elindítja. Kezdetben minden rács HALOTT. Ezen belül lesz meghívva a siklólövő aminek a kód végen minden kordinátája megvan adva. Vannak billentyűről beérkezőparancsaink is, különböző feladatokkal ellátva pl a "g" betűvel, a két állapot közötti időt csökkentjük. Ugyan így vannak az egérrel történő infomációk feldolgozására szolgáló függvények. Külön a kattintásra és a mozgatásra. Külön tudunk készíteni pillanatfelvételt az aktuális állapotról az "s" gomb segítségével. A programban a sejtér rajzolását a paint() függvénytel végezzük. A szomszédokSzáma() függvéyenben vizsgáljuk a szabályokat és aszerint történik a sejtek viselkedése.



7.3. Qt C++ életjáték

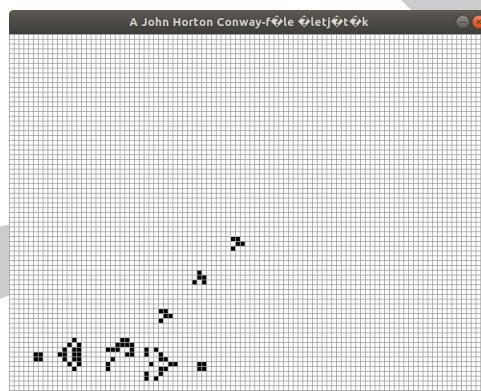
Most alkossuk meg az életjátékot Qt C++-ban is. A program lényege itt is ugyan az, mint a java-ban. A szabályok ugyan azok. Ha kettő vagy három szomszédja van, akkor életben marad, ha 2-nél kevesebb kipusztul, ha 3 nál több, akkor tulnápesedés miatt elpusztul. Itt is a siklóágyú lesz a fő célunk.

Megoldás forrása: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/Qt/Sejtauto/>

```
#include <QApplication>
#include "sejtablak.h"
#include <QDesktopWidget>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    SejtAblak w(100, 75);
    w.show();

    return a.exec();
}
```

A forrásfájljaink a Sejtablak.cpp , sejtszal.h, sejtszal.cpp és a sejtablak.h . Ezek az átláthatóság miatt vannak külön. A sejtablak.h és .cpp tartalmazza a függvényeket amivel majd a kirajzolás fog történni és ebben van a sikló lövés is, úgy mint a java-s pájránál, külön minden egyes cellát megadunk amiben sejt van. A szejtszal.h és .c pedig az életjátékhoz szükséges szabályokat . Ezen belül vannak a függvények melyek az adott állapotokat vizsgálják és a szabályok szerint alakítják a programot.



7.4. BrainB Benchmark

Elsőnek is a Benchmark jelentését nézzük meg. A benchmark egy elemzés, tesztfeladat. Egy bizonyos tesztet végez el és azt az elért pontszám alapján összehasonlíta a tesztet elvégzők között. Ilyen például telefonok teljesítményét végző benchmark , vagy az esetünkben az agy teljesítményét vizsgáló. Ennek segítségével tudunk egy vizsgálati alapot venni egy adott feladatban. Például a telefonoknál, hogy minél több pontot ér el annál jobb a teljesítménye és össze tudjuk hasonlítani más telefonokkal. Ugyan így a BraniBenchmarkban, a tesztet megoldó emberek közül az adott pontszám megadja, hogy ki teljesített a legjobban és egymáshoz is tudjuk vizsgálni őket. Az adott programunk az egyének figyelemképpeségét és koncentrációját fogja vizsgálni. Adott egy karakter, ami a mi karakterünk és azon kell tartanunk az egér kurzort. A program azt vizsgálja, hogy mennyi ideig vagyunk képesek a kurzort a mi karakterünkön tartani, azaz meddig nem veszítjük el azt. Persze nem ilyen egyszerű, mert közben rengeteg új karakter jelenik meg a monitoron befolyásolva ezzel minket, hogy elveszítsük a karakterünket. A program arra is reagál, ha elveszítjük a karakterünk.

Az adott programban a mi karakterünk Samu lesz. Samut figyelemmel kell tartani. Ahogy fent említettem ezt a kurzorral fogjuk megtenni. Minél tovább tartjuk Samun a kurzort annál több másik karakter lesz a

képernyőn. A feladat 10 percig tart és annál jobb vagy ha minél több kis karakter között is megtudod tartani a saját karaktered. Ha elveszítenéd abban az esetben belassul az új karakterek megjelenése még nem találd. Annál jobban teljesítettél, minél több pontod van a 10 perc végén.

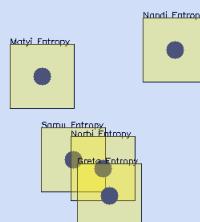
Megoldás forrása: <https://github.com/nbatfai/esport-talent-search>

```
eturn a.exec();  
#include <QApplication>  
#include <QTextStream>  
#include <QtWidgets>  
#include "BrainBWin.h"  
int main ( int argc, char **argv )  
{  
    QApplication app ( argc, argv );  
    QTextStream qout ( stdout );  
    qout.setCodec ( "UTF-8" );  
    qout << "\n" << BrainBWin::appName << QString::fromUtf8 ( " Copyright (C) 2017, 2018 Norbert Bátfai" ) << endl;  
    qout << "This program is free software: you can redistribute it and /or modify it under" << endl;  
    qout << "the terms of the GNU General Public License as published by the Free Software" << endl;  
    qout << "Foundation, either version 3 of the License, or (at your option) any later" << endl;  
    qout << "version.\n" << endl;  
    qout << "This program is distributed in the hope that it will be useful, but WITHOUT" << endl;  
    qout << "ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS" << endl;  
    qout << "FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.\n" << endl;  
    qout << QString::fromUtf8 ( "Ez a program szabad szoftver; terjeszthető illetve módosítható a Free Software" ) << endl;  
    qout << QString::fromUtf8 ( "Foundation által kiadott GNU General Public License dokumentumában leírtak;" ) << endl;  
    qout << QString::fromUtf8 ( "akár a licenc 3-as, akár (tetszőleges) későbbi változata szerint.\n" ) << endl;  
    qout << QString::fromUtf8 ( "Ez a program abban a reményben kerül közreadásra, hogy hasznos lesz, de minden" ) << endl;  
    qout << QString::fromUtf8 ( "egyéb GARANCIA NÉLKÜL, az ELADHATÓSÁGRA vagy VALAMELY CÉLRA VALÓ" ) << endl;  
    qout << QString::fromUtf8 ( "ALKALMAZHATÓSÁGRA való származtatott garanciát is beleértve. További" ) << endl;  
    qout << QString::fromUtf8 ( "részleteket a GNU General Public License tartalmaz.\n" ) << endl;  
    qout << "http://gnu.hu/gplv3.html" << endl;  
    QRect rect = QApplication::desktop()->availableGeometry();  
    BrainBWin brainBWin ( rect.width(), rect.height() );  
    brainBWin.setWindowState ( brainBWin.windowState() ^ Qt::WindowFullScreen );  
    brainBWin.show();
```

```
        return app.exec();  
    }
```

Ez ismét egy QT program. Fent csak a main.cpp látható de itt a többiről is beszélünk. Először nézzük a BraintBTheard.cpp-t. Itt áll elő a kezdő pozíció. Létrejön a mi karakterünk és 4 másik karakter. Ezek úgy vannak mindenig elhelyezve, hogy mindenig egymás közelébe legyenek, hogy a feladatunk ne legyen túl könnyű. A run függvény a teszt indításáért felel. Itt méri az időt is, azaz a program addig fut amíg az idő a megadott időnl(10 perc) kisebb. Itt található még a pause függvény aminek a neve a válasz. A következő BraintBTheard.cpp található a karakterek kinézete, ezeknek a neve programon belül "hero" azaz hős. Itt adjuk meg a nevét, az elhelyeszkedését, színét, és a mozgásának a gyorsaságát. Továbbá a többi karakter születése, gyorsasága és további információkat róluk itt adunk meg. A BrainBWin.cpp-ben megadjuk a program nevét a verzió számát. Az UpdateHeroes függvényben zajlik a kurzorunk menetének vizsgálata, itt figyeli hogy rajta vagyunk e az egérrel a karakteren, hányszor veszítettük el a karakterünk, vagy hogy éppen fut-e a teszt. Ezek utána a kövezkező függvény kirajzolja az ablakot. Itt van továbbra az óra megjelenítése vagy a pontszámunkkké. Ezek után jön az egér funkciói. Például ha lenyomjuk akkor elindul. Vagy az elmozdítás követése. És a billentyűzetről bevitt karaktereket is itt dolgozza fel.

Press and hold the mouse button on the center of Samu Entropy
0:0/10:0 0 bps PAUSED (0)



8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa Python

Először is a Python nyelvet mutatjuk be. A nyelv egy magas szintű programozási nyelv. 1991 ben került nyilvánosság elé, amit Guido holland származású programozó fejlesztett. A python interpreteres nyelv.

A Minst kézzel írott számok adatbázisa (6000 kép). Ez az alapja azoknak a programoknak ami képről ismeri fel a tárgyakat. A kézel beírt számokról a program el fogja dönteni hogy milyen szám, ez azért érdekes, mert a kézzel írott írás szinte mindenkinél más, viszont a programnak mindig tudnia kell, hogy melyik számot kell felismernie. A programhoz a TensorFlowet használjuk. A TensorFlow a Google által alkotott gépi tanulási rendszer. Sok helyen használják, az egyik leghasználtabb, az a google mapsban található utcakép. Ez neutrális háló helyett itt transzformációs gráfok találhatóak. A TensorFlow nyílt forráskódú, le kell töltenünk a használathoz. Nézzük a kódot. Először is a könyvtárakat amik kellenek, a from kulcsszóval fogjuk ezeket hozzáadni a programhoz. Utána beimportáljuk a Tensorflow könyvtárt. Ezután következik a main. Ebben van a kiíratás felépítése, hogy hogyan küldjük ki az eredményeket (a képen látszik.). A sess azzaz egy session segítségével fogjuk a tanítást végezni, hasonlóan mint a neurális hálózatnál. Az alaposság kedvéért, hogy minél pontosabb eredményt kapjunk ezerszer futtatjuk a ciklust. Ztánna kiíratjuk mennyire lett pontos az eredmény. A programban először felugrik maga a kép ami a kézel írott számot taartalmazza(ehez a matplotlib-ot használjuk, hogy meg tudjuk rajzolni), ha ezt bezárjuk jön a következő kép, a képeket az aktuális mappában lementjük. A programnak elég nagy a pontossága, jól felismeri. Az eredményeket egy tömben tárolja el a program.

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

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import argparse
# Import data
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
import matplotlib.pyplot
FLAGS = None
```

```
def main(_):
    mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
    # Create the model
    x = tf.placeholder(tf.float32, [None, 784])
    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))
    y = tf.matmul(x, W) + b
    # Define loss and optimizer
    y_ = tf.placeholder(tf.float32, [None, 10])
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, y_))
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
    sess = tf.InteractiveSession()
    # Train
    tf.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/10, "%")
    print("-----")
    # Test trained model
    print("-- A halozat tesztelese")
    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    print("-- Pontossag: ", sess.run(accuracy, feed_dict={x: mnist.test.images,
                                                          y_: mnist.test.labels}))
    print("-----")

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

    img = mnist.test.images[42]
    image = img
    matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm.binary)
    matplotlib.pyplot.savefig("4.png")
    matplotlib.pyplot.show()
    classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})
    print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
    print("-----")
    print("-- A sajat kezi 8-asom felismerese, mutatom a szamot, a tovabbolteshez 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)
```

```
.binary)
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()
classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image] })
print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/ ←
        mnist/input_data',
                        help='Directory for storing input data')
FLAGS = parser.parse_args()
tf.app.run()
```

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat: Itt el szeretném használni az első passzolási lehetőségem.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

A lisp programozási nyelv a mesterséges intelligenciát kutatók kedvelt nyelve, eredetileg nem ez volt a célja, de hamar az MI kutatásban lett használatos. Neve jelentése a listafeldolgozás, mivel a programok felépítése láncolt lista(zárojelekkel választjuk el a listákat). A lisp egy kifejezésorientált nyelv.

Nézzük meg elsőnek Iteratív módon.

```
(defun faktorialisi (n)
  (do
    ((i 1 (+ 1 i))
     (prod 1 (* i prod)))
    ((equal i (+ n 1)) prod)))
```

Az elején a defun-al adjuk meg a függvény nevét és a változót amibe majd az érték érkezik. A do egy konstrukció, ezt iteratív programknál használjuk mint ez is(iteratív struktúra). a (+ 1 i)-ben növeli az i értékét 1 el utánna végzi el a szorzást aztán növeli az n et, amíg szükséges.

Rekurzív módon:

```
(defun faktorialisr(n)
  (if (= n 1)
      1
      (* n (faktorialisr (- n 1)))))
```

Az előző program rekurzív változata. Az elején a defunnal adjuk meg a nevet és a paramétert. Aztán if el vizsgáljuk hogy n egyenlő e 1 el, ha egyenlő akkor meghívja önmagát n-1 re és összeszorozza az n el.

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

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

Tanulságok, tapasztalatok, magyarázat...

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Tanulságok, tapasztalatok, magyarázat...

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

[?]

A könyvben legelőször az alapfogalmakkal fogunk foglalkozni. Először is nézzük a programozási nyelveket. ezeknek 3 szintjét különböztetjük meg. Ez a három a gépi nyelv, assembly nyelv és a magasz szintű nyelv (C++, C ...stb). Minket a magasz szintű programozási nyelv fog érintetni. Az ebben írt programokat forrásprogramoknak nevezzük. Ezt a gépnek értelmeznie kell és feldolgoznia gépi nyelvre. Ezt egy fordító program segítségével jahtjuk végre, Ez először tárgyprogramot hoz létre majd ebből lesz a gépi nyelv. Az átfordításnak 4 lépése van. 1. lexikális elemzés. 2. szintaktikai elemzés. 3. szemantikai elemzés. 4.kód-generálás. Az első lépésnél a forrást lexikális egyégekre bontjuk. A 2 lépésnél a szintaktika helyességét vizsgálja, hogy megelel e a szintaktika szabályainak. Mivel, ha nem helyes a szintaktika, nem lehet gépi nyelvet eköállítani, ugyanis nem fogja megérteni. A 3 lépés a program megértése szemantikailag. A 4. lépés pedig legenerálni a kódot. Ezt kapja meg a vezérlő operációs rendszer. A másik technika az interpereteres technika, az első 3 lépés itt megegyezik a fordító programokkal, a különbség, hogy itt nem készül tárgyprogramot. Sorba veszi az utasításokat és sorban értelmezi azokat majd végrehajtja. minden nyelvnek saját hivatkozási nyelve van. Ebben van a nyelv szemantikai és szintaktikai szabáylai definiálva. A szemantikai részt emberi nyelven szokták megadni, a legelterjettebb az angol nyelv ezen a téren. A következő pontban a nyelvek osztályozásába kapunk betekintést. 2 fő osztályra bontjuk őket. Imperatív nyelvek és Deklaratív nyelvek. Az utóbbi algoritmikus nyelv, még az előbbi nem algoritmikus nyelv. A könyvben részletesen ki vannak fejtve. A következő a jelölés rendszer. A szintaktika formális leírásához alkalmazzuk ezeket. Vannak terminális én nem terminális jelölések. A jelölések után következnek a kifejezések. Ezek szintaktikai eszközök. Ezeknek van értéke és tipusa. A kifejezések formálisan operandusokból, operátorokból és kerek zárojelekből állnak. Az operadusok a változók vagy függvény meghívása lehet. Az operátorok pedig a műveletek amit az értékekkel elvégzünk. A záró jelek egybe foglalják és a sorrendet befolyásolják. Egy ilyen kifejezésnek 3 alakját tudjuk felírn: Prefix (pl: + 7 2), infix (pl: 7 + 2), postfix (pl: 7 2 +). Az infix alakban az operátorok nem azonos errőségűek. HA egyértelmű infix kifejezést akarunk létrehozni akkor teljesen zároljeleznünk kell azt. Ez felül írja a precedencia táblázatot. A könyvben továbbázt vizsgáljuk milyen nyelv, hogyan használja és hogyan értékeli ki a kifejezéseket. A C egy kifejezés orientált programozási nyelv. Típuskényszerítés elvét használja. A C precedencia táblázatát is megírjuk a könyv 51 oldalán. Például: [] ez a tömb operátor, . minősítő operátor, -> mutatóval minősítő operátor, ++ és -- értéknövelő és csökkentő operandusok, sizeof() operátor típus vagy kifejezés hosszát adja meg. Vannak a matematikai operátorok, mint a szorzás, osztás, összeadás, kivonás. Hasonlító operátorok mint az egyenlő,

nem egyenlő, nagyobb vagy egyenlő...stb. A logikai operátorok: "vagy", "kizárt vagy", "és". A könyvben ezekhez találunk leírást, hogy hogyan használjuk. Az adattípusokkal is foglalkozunk a 2.4 es pontban. Az adattípus megadja, hogy a változó milyen típusú, azaz minden típusnév egy azonosító. 3 dolog határoz meg egy adattípust: tartomyán, műveletek, reprezentáció. A programozási nyelvekben lehetőség van definiálni típusokat. Lehet létrehozni is de minden nyelvben vannak beépített típusok. Ilyen például az int azaz az egész típus, ennek vannak variánsai például shor vagy long int, a bool a logikai típus, float vagy a double amik lebegőpontos számokat tudnak tárolni. A könyvben megnézzük az egyszerű és összetett típusok. A könyvben nagy sújt kap a saját típus létrehozása. Szót kell még ejteni a tömbökről, ezeknek is van típusa ugyan úgy mint a változóknak, a tömbök olyan típusú változókat tartalmaznak amilyen típusú a tömb, ezen belül a tömb indexével tudunk mozogni. A mutató típust is átvesszük. A nevesített konstans egy olyan eszköz a programozásban aminek 3 része van: Név, Típus, Érték. Ezt a 3 at mindig deklarálni kell. Ugye a konstans jelentése állandó, tehát ez a programban mindenkor minden deklarálásnál megadott nevet, értéket és típust fogja tartalmazni. Ezeket érdemes beszélő nevekkel ellátni, és olyan értékeket adni amiket sokszor használunk. C ben ezt az alábbi módon kell: "#define név literál ". A változó a konstant "testvére" úgymond. Neki 4 komponense van, Név, attribútum, cím, érték. A változó ahogy a neve is mondja nem állandó. A név az azonosítója. ezzel hivatkozunk rá a programban. Az attributok közül a legfontosabb hogy milyen típusú, a típusokról már fentebb beszélünk, amilyen típusú olyan értéket tud tárolni. És tudunk címet adni neki, azaz hol helyezkedjen el a tárolóban. de az elhelyezést a gép végzi, de hogy mikor hozza létre az attol függ hogy hol deklaráljuk . Nézzük meg egy példát: int a=5 . ez egy egész típusú változó kezdőértéke 5 a neve pedig "a". Most nézzük meg a C nyelv alapelemeit. Vannak integrált típusok (int, char...stb). Származtatott típusok (tömb, függvény, union, mutató). Ezek az aritmetikai típusok. A tömböt az alábbi módon hozzuk létre: int a[elemszám]. megadjuk a típusát a nevét és hogy hány elemű. Az utasítások a következő rész. Az utasítások alkotják a programot. Ezekből épül fel egy algoritmus, ciklus, és szinte minden, az utasításokat fordítja le a fordítóprogram tárgyprogramra. Kér nagyobb részre tudjuk bontani, deklarációs utasítás és végrehajtható utasítás. A deklarációs utasítás a fordítóprogram miatt vannak. tőle kérnek szolgáltatást vagy egy üzzemmódbba való lépést. Befolyásolja a tárgykódot de nem kerül lefordításra. A végrehajtható utasítás, ahogy a nevében is benne van, bővégre lehet hajtani. Ezekből lesz generálva a tárgykód. Ezeket tudjuk csoportosítani például értékeadó utasítás, üres utasítás, ugró utasítás, elágazó utasítás, hívó utasítás, I/O utasítások ...stb. Ezekről a könyvben részletes leírást kapunk. Például az értékeadó utasításban értéket adunk vagy modosítunk egy változón vagy több változón. Az elágazó utasítások is szinte kihagyhatatlanok egy programban, ugye ezek az if és az else if feltételes utasítások. Ezek lehetővé teszik a programban a több irányú elágazást. A ciklusszervező utasításokat is ismerjük már ha nem kezdők vagyunk. Ugye ezek a for, while, do while ciklusok. A végtelen ciklus volt az első feladatunk, azt már ismerjük, Vannak feltételes ciklusok, ugye itt addig fut a ciklus még a feltétel igaz. Van kezdő feltételes ls végfeltételes ciklus. Ezeknek a működését a könyvben megismertük. Van az előírt lépésszámú ciklus (a for()). Annyiszor fut le a ciklus, ahányszor előírtuk neki. Vannak összetett ciklusok Ez az előzőök kombinációja. A működésük nagyon bonyolult. 3 vezérlő utasítás van C ben. 1. Return: Ez szabályosan befejezzi a függvényt és vissza adja azt a vezérlést hívónak, legtöbbször értékkal tér vissza. 2. BREAK: ezt a cikluson belül alkalmazzuk, ha életbe lép akkor kilép a ciklusból és az utána lévő cikluson belüli utasításokat nem hajtja végre. Ezt iffel szoktuk alkalmazni. 3. Continue: ez is a ciklus magban van. ez is kilép, vagy újabb cikluslépésbe kezd, vagy megvizsgálja újra az ismétlődés feltételeit. Az 5. fejezetben a programok szerkezetét ismerjük meg, hogy milyen részekből áll. Itt ismerjük meg az alprogram feladata egy bemeneti adatcsoport leképzése vagy kimeneti adatcsoportot készít le, egy megadott specifikáció szerint. Ebben a fejezetben ismerjük meg a blokkokat. a blokk egy programegység, ami utasításokat foglal magában. A kezdetét és végét speciális karakter jelzi. A blokk bárhol elhelyezhető a programban. A 13. I/O fogjuk venni. Ez egy olyan rész ahol a program nyelvek nagyban eltérnek egymástól. Állimányuk a funkciók szerint 3 lehet. lehet input,output és input-output állományán. Az I/O során a programban az adatok a tár és a periféria kö-

zött mozognak. Ezeknek van egy bizonyos ébrázolási módja. Az adatátvitelnek 3 fajtáját alkalmazzuk: formátumos módú, szerkesztett modú és listázott modú adatátvietl. Ha állományokat alkalmazunk azt a következpképpen kell csinálnunk. 1. deklaráció, 2. összerendelés, 3. állomány megnyitása, 4. feldolgozás, 5. lezárás. A C-nyelvnek nem része az I/O erre egy standart könyvtárat használunk.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

A könyvben a C nyelv alapjaival fogunk megismерkedni az alapoktól. A könyv programok segítségével segít elsajátítani a C nyelv ismeretét. Ha eddig egyáltalán nem találkoztunk a C nyelvel, a könyv akkor is nagyon hasznos lehet, hiszen nagyon részletesen, mindenre odafigyelve adja át a tudást. Először kezdjük a változók típusaival és a program beli alkalmazásával. A programban különböző típusú ezeken belül különböző méretű változók vannak. Típusok: int -intiger, azaz egész típusú számot deklarálunk ezzel, mérete a gép egészétől függ. char -Ez egy karaktert tud tárolni, mérete 1 bájt. float - egyszeresen pontos lebegőpontos számot tud tárolni (törtet). double- kétszeresen pontos lebegőpontos számot tud tárolni, ezt hatalmas számokkal való számolásnál használjuk. Az int típustnak vannak fajtái még, ilyen a short int vagy a long int, van továbbá még unsigned int amivel a negatív számok helyett, csak pozitív értékeket tudunk tárolni, de abból többet mint a sima int-nél. Továbbá ami még lényeges lehet, az lokális és globális változó, ahogy a nevük is jelzi, a globális az egész programban használható, még a lokális csak a megadott helyen. A könyvben a helyes alkalmazásról rengeteg példát láthatunk. Vannak az állandók, ezeknek megadunk egy értéket, és ez az érték állandó lesz a programban, nem változik. A könyv következő fejezetében a Vezérlési szerkezetet vesszük. Ez határozza meg, hogy milyen sorrendben hajtódnak végre a műveletek. Először az utasításokat és a blokkokat nézzük. Fontos szabály, hogy minden utasítást ";" -al zárunk le. Így válik egy sor utasítássá. Ha az utasítások {}-el vannak körbe zárva, akkor azt egy bloknak nevezzük és 1 db utasításlént kezeljük. A blokkok közé soroljuk a ciklusokat és a az elágazásokat. Először az if utasítással ismerkedünk meg. Ha a megadott feltétel igaz, akkor a megadott utasítások fognak lefutni, ha nem igaz akkor az utasításokat figyelmenkívül hagyja. Ennek a bővítése az else-if. Itt meg tudjuk adni, hogy a program milyen utasításokat hajtson végre, ha a feltétel nem igaz. A következő a switch utasítás. A lényege hogy megadunk neki mintákat, és ha a vizsgált elem megegyezik valamelyik mintával, a mintánál megadott utasítások fognak lefutni. ezek után vegyük a ciklusókat. A ciklus egy olyan blok ami a feltételig újra és újra lefut. Kezdjük a while ciklussal, a while jelentése amíg, tehát amíg a kifejezés igaz addig lefut a ciklus, itt az lesz fontos, hogy először vizsgálja meg , hogy a feltétel igaz e, aztán fog lefutni az utasítás vagy utasítások. Ezzel megegyezik a for utasítás, a lényege és működési elve ugyan ez. A felépítése különbözik. A do while ciklus először végrehajtja az utasítást és aztán vizsgálja meg a feltételt, ha nem igaz akkor kilép a ciklusból. A break utasítással a futás közben is kitudunk lépni egy ciklusból ezt a ciklusokban lévő if-ekben szoktok alkalmazni. A goto -val pedig egy adott címclére ugorhatunk. De ezt nem sűrűn alkalmazzák, sokan nem is szeretik. Rendezni tudjuk ezeket a futási sorrend alapján. Van a sima ami sorban fut le, de van a címkezett usasítás blokk az előbb említett goto-val, itt az adott feltételektől függ, milyen sorrendben fut le a program. A Függelékbén az utasításoknál csoporthozosítjuk az utasítások fajtáit. A címzett utasításokhoz előtagként megadott címke kapcsolódik. A kifejezésutasítás, a nevéből adódoan kifejezésekkel épül fel. Az összetett utasítás megszünteti a korlátozást ahol a fordító csak egyetlen utasítást fogad el. A kiválasztó utasítások, a lehetséges esetek közül választ a feltételnek megfelelően (if, switch). A vezérlés átadó utasítások, amár fentiekben említett goto, continuem break, return parancsokat alkalmazza. Az Iterációs utasítások a ciklusokat alkalmazzák (while, do, for).

10.3. Programozás

[BMECPP]

A könyv témája a szoftverfejlesztés C++-ban. A legelső fejezetben azokat az új dolgokat fogjuk megnézni amiket C-ben nem tudtunk de C++ ban már lehetséges. Elsőre függvényparaméterek és visszatérési értékek vesszük. A C ben üres paraméterlistával definiált függvényt itt C++ ban void paraméter megadásával oldjuk meg.A visszatérési típusnál is eltér a 2 nyelv. Míg a C nyelvél int, addig C++ nem támpatja az alapmérétezett típust. Az int main fő függvénynek is 2 típusa van C++ -ban. Az első mikor nem adunk meg semmit, a másik mikor argumentumokat adunk a mainnak a parancssorból. A C++ nyelvben jelent meg először a bool azaz a logikai típus aminek értéke true vagy false. A feladatok között van egy ahol az volt a hiba, mikor utasításon belül deklaráltunk. C++ ban már ez is lehetséges. A példa erre, hogy az i értéket a forcikluson belül deklaráljuk: "for(int i=0; i kisebb mint 10; i++)". Hasznos lehet, ha a deklarációt a felhasználás előtt végezzük. C++ ban a függvényeket a nevük és a megadott argumentumokkal azonosítjuk. Azonos nevű csak akkor lehet, ha más az argumentum lista. C-ben a paraméterátadás érték szerint történik. A megkapott érték klonózva lesz és a másolatra fogunk hivatkozni, azaz a visszatérési érték nem befolyásolja a programunkat. Ez az adott példában a könyv jól szemlélteti. Ennek C++ ban a megoldása hogy az eredeti változót fogjuk átadni mégpedig a memória címével. Ezt a referencia jellel fogjuk teljesíteni. Így a változtatások, az eredeti változón játódnak végbe, és az új értéket kapja vissza a program. Ez a referencia C-ben nem létezik. A harmadik fejezetben megírjuk az Objektumokat és osztályokat. A C++ nyelv egy objektumorientált nyelv, ennek alapelveivel megismerkedünk. A lényege hogy a függvényeket osztályokba foglaljuk az osztályok példányait nevezük objektumoknak. (objektumokkal hívjuk meg az osztályok függvényeit.) A 3.2 bekezdésben egy példán keresztül láthatjuk, az egységbázárás előnyeit. A struktúráknak így lesznek tagváltozói, és tagfüggvényei is. A tagfüggvényeket kétféle képpen adhatjuk meg. Osztálydefinícióban vagy struktúradefiníción kívül . Fontos még az adatrejtésről is beszélünk. Így csak az osztályon belül férhetünk hozzá, külső osztály nem férhet hozzá, az értéket függvényel tudjuk kiadni. függvény és változó is lehet ilyen, ezeket a private részbe írjuk, azaz ezeket privátá tesszük. A konstruktur onnan ismerjük fel, hogy ugyan az a neve mint az osztálynak és nem adunk típust neki, ez akkor fut le, ha meghívjuk az adott osztályt amiben benne van, egy objektummal, ezt gakran inicIALIZÁSRA használjuk. A destruktor annyiban különbözik szintaktikailag, hogy a név előtt egy '~' jel található, és automatikusan meghívódik egy objektum befejeződésekor, ezt a memória felszabadításra szoktuk használni főként. A c-ben a dinamikus memória foglalás malloc-al vagy a free kulcsszavakkal történtek. A C++ ban már operátorral tudunk lefoglalni memóriát, ez a new operátor, példa: "int *pelda; pelda= new int;" később ezt a változót fel kell majd szabadítanunk, vagy memóriászivárgás lép fel. A dinamikus adattámagatás miatt szükséges sokszor, hogy megadjuk a mutató állapotát. Ez a NULL kulcsszó. Ha az állapot NULL akkor nincsen lefoglalt adat, ha ezt nem adjuk meg akkor a new szócskával lefoglalt területre mutat. A 3.5.3 fejezetben ismerkedünk meg a másoló konstruktőrral. A másolókonstruktur egy referenciát kap, amely megegyezik az osztálynak a típusával. A másolókonstruktur is rendelkezik mindenkel, amivel egy egyszerű konstruktur. Ugyan úgy tudunk inicIALIZálni vele objektumokat. Amiben viszont több hogy érték szerint adunk egy függvényparamétert neki, akkor a megadott változó lemásolódik, és ezt használjuk a függvény törzsében. Az osztályainak lehetnek friend függvényei vagy osztályai. Ez azt jelenti, hogy jogot adunk egy függvénynek vagy egy osztálynak, hogy hozzá férjen az adott osztály védett, azaz private és protected változóihoz és függvényeihez. Ezt a friend kulcsszóval tesszük lehetővé. A feljogosítandó függvény vagy osztály elé írjuk és ezzel fel is jogosítjuk. A tagváltozóknál az értékadás és az inicializálás nem ugyan az. Az inicializálás azaz alaphelyzetbe álltjuk. A létrehozásnál megadunk neki egy alap értéket. Az értékadás az az "=" jelel történik a programtol függően, ez a programon belül bárhol megtörténhet ahol a szabályok engedik. Említenünk kell a statikus tagokat, ezek a tagok az osztályhoz tartoznak és nem az osztályob-

jektumhoz. Továbbá lehetőségünk van struktúráknak, osztályoknak típusdefinícióra, a `typedef` szóval. A hatodik fejezetben vesszük az operátorokat és az operátortérhelést. A C nyelvben az operátorok műveleteket végeznek az argumentumokon. Az operátorok kiértékelési sorrendjét egy speciális szabályrendszer határozza meg, ezeket ugyan úgy mint matematikában, zárójelekkel írhatunk felül. A C++ rendelkezik új operátorokkal a C -hez képest. Ilyen a hatókör operátor aminek jelölése a `:::` ezt az osztályok hatólörének megadásánál használjuk. Ismerjük már a `*` operátor, ugye ezzel jelöljük a mutatókat. A `->` operátorral pedig mutató esetén hivatkozunk. Fontos különbség a C és a C++ között a függvénymellékhatása, A C nem képes erre, de a C++ igen. Ráadásul mivel az operátor speciális függvény ezért különböző argumentumok esetén túl tudjuk őket terhelni. A 10-es fejezet a kivitelezésről szól. Ilyen a hagyományos hibák kezelése. De mi is az a kivitelezés? A kivitelezés egy mechanizmus, amely ha hibát fedez fel, akkor a hibakezelő ágra ugrik. Ilyen a `try_catch` blokk. A `throw` kulcsal küldünk egy kidobást. A `try-catch` pedig elkapja, ha egy `catch` ág megegyezik az elkapott típussal. Ha nem kapja el, azt kezeletlen kivitelnek nevezzük. Van egymásba ágyazott `try-catch` blokk, így tudjuk a kidobásokat külön szinten kezelní

DRAFT

III. rész

Második felvonás

DRAFT

**Bátf41 Haxor Stream**

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

DRAFT

11. fejezet

Helló, Arroway!

11.1. OO szemlélet

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

A következő programcsipet a JDK beéített metódusa amely megtalálható a `java.util.Random` forrásban a `nextGaussian` néven.

```
public class NextGuassian {  
  
    private double nextNextGaussian;  
    private boolean haveNextNextGaussian = false;  
  
    public double nextGaussian() {  
        if (haveNextNextGaussian) {  
            haveNextNextGaussian = false;  
            return nextNextGaussian;  
        } else {  
            double n1, n2, v1, v2, s;  
            do {  
                n1 = Math.random(); // nextDouble()  
                n2 = Math.random();  
  
                v1 = 2 * n1 - 1; // between -1.0 and 1.0  
                v2 = 2 * n2 - 1; // between -1.0 and 1.0  
                s = v1 * v1 + v2 * v2;  
            } while (s >= 1 || s == 0);  
            double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);  
            nextNextGaussian = v2 * multiplier;  
            haveNextNextGaussian = true;  
            return v1 * multiplier;  
        }  
    }  
}
```

A következő kód részlet a PolarGen, amit már előző félévből is ismerhetünk.

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

    public PolarGenerator()
    {
        nincsTarolt = true;
    }

    public double kovetkezo()
    {
        if(nincsTarolt)
        {
            double u1, u2, v1, v2, w;
            do{
                u1 = Math.random();
                u2 = Math.random();
                v1 = 2* u1 -1;
                v2 = 2* u2 -1;
                w = v1*v1 + v2*v2;
            } while (w>1);

            double r = Math.sqrt((-2 * Math.log(w) / w));
            tarolt = r * v2;
            nincsTarolt = !nincsTarolt;
            return r * v1;
        }
        else
        {
            nincsTarolt = !nincsTarolt;
            return tarolt;
        }
    }
    public static void main(String[] args)
    {
        PolarGenerator g = new PolarGenerator();
        for (int i = 0; i < 10; ++i)
        {
            System.out.println(g.kovetkezo());
        }
    }
}
```

Ha alaposan megfigyeljük a kér kódot, akkor látható, hogy lényegében csak a változok nevei változtak meg. Tehát levonhatjuk azt a következtetést, hogy a az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua., mint a mi kódunk.

Beszéljünk kicsit a programról és hogy mit csinál. Nem nagyon fogom részletezni, mivel a Welch fejezetbe

már le van írva. A kovetkezo() nevű metódus matematikai számítás alapján hoz létre egy új számokat. Az if-ben megnézzük, hogy van-e már létrehozott érték, illetve ha nincs, tovább lépünk a ciklusba. Ez létrehoz 2 "véletlen" számot a random() függvényel. Ha a szám nagyobb mint 1, a while() ciklusban matematikai számításokat végzünk, majd értéket ad a tarolt nevű változónak. Ezt követően nincsTarolt változó logikai értékét megváltoztatja. A returnnel pedig visszaadja a létrehozott értéket. Ha már volt generált szám, akkor a nincsTarolt változó logikai értékét megváltoztatja és azt a számot adja vissza.

Lássuk úgyan ezt a kódot C++ nyelven:

```
#include <iostream>
#include <math.h>
#include <pthread.h>
#include <ctime>

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

double PolarGen::kovetkezo ()
{
    if (nincsTarolt)
    {
        double u1, u2, v1, v2, w;
        do
        {
            u1= std::rand() / (RAND_MAX +1.0);
            u2= std::rand() / (RAND_MAX +1.0);
            v1=2*u1-1;
            v2=2*u1-1;
            w=v1*v1+v2*v2;
        }
        while (w>1);
        double r =std::sqrt ((-2 * std::log(w)) /w);
        tarolt=r*v2;
        nincsTarolt =!nincsTarolt;
        return r* v1;
    }
    else
```

```
{  
    nincsTarolt = !nincsTarolt;  
    return tarolt;  
}  
}  
int main (int argc, char **argv)  
{  
    PolarGen pg;  
    for (int i= 0; i<10;i++)  
        std::cout<<pg.kovetkezo ()<< std::endl;  
    return 0;  
}
```

11.2. „Gagyi”

Az ismert formális3 „while ($x \leq t \&& x \geq t \&& t \neq x$);” tesztkérdéstípusra adj a szokásosnál (miszerint x , t az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciaja) „mélyebb” választ, írj Java példaprogramot mely egyszer végtelen ciklus, más x , t értékekkel meg nem! A példát építsd a JDK Integer.java forrására4 , hogy a 128-nál inkluzív objektum példányokat poolozza!

A feladatban szerepeltetett feltétel azért érdekes, mivel az integerben a -128 127 intervallum előre tárolva van és a program az eltárolt memória címeket hasonlítja össze. Abban az esetben, ha az intervallumon kívül eső értéket veszünk fel, példa kép a -129-t mind két változóban, akkor a két memóriacím különbözik, ezt követően őeding, mivel a feltétel teljesül belép a végtelen ciklusba a program.

11.3. Yoda

Írunk olyan Java programot, ami java.lang.NullPointerException-le áll, ha nem követjük a Yoda conditions-t!

A Yoda conditions nevezhető egy programozási "nyelv jásrásnak". A lényege, hogy a feltétel két részét megfordítjuk. Az az az összehasonlítandó értéket a megszokással ellentétben, nem bal oldalra, hanem jobb oldalra tesszük.

```
if ($value == 42) /*e helyet*/  
if (42 == $value) /*ezt írjuk*/
```

A nevét a jelenség onnan kapta, hogy a Star Warsból jól ismert karakter Yoda, törve beszélte az angol nyelvet.

```
public class yoda {  
    public static void main (String[]args)  
    {  
        String myString = null;  
        /*if (myString.equals("foobar"))  
        System.out.println("Yoda nem ért");*/  
        if (!("foobar".equals(myString)))  
            System.out.println("Yoda érti");
```

```
}
```

A fenti programrészben elértük, hogy a kommentált rész java.lang.NullpointerEx hibával leálljon, mivel a NULL-t nem tudjuk hasonlítani a karakter sorhoz.

11.4. EPAM: Objektum példányosítás programozási mintákkal

Hozz példát mindegyik "creational design pattern"-re és mutasd be mikor érdemes használni őket!

A létrehozási minták a példányosítás folyamatát ábrázolja. Objektumok létrehozására valók és, hogy egyszerűsítsék kódot illetve a tervezést.

Gyártó metódus programtervezési minta

Ezzel a mintával a kódban található számos egyforma példányosító utosítás kiváltható. A függvények neve általában a Create, Make vagy Do szóval kezdődnek és a nevükben megadott osztály egy példányát adják vissza. Az ősosztályban elhelyezett gyártásimétódus írja le a gyártás algoritmusát, a gyermek osztály pedig eldönti mit kell legyártani.

A inkelt példában az "auto" osztály egy abstact osztály lesz, ennek a mintjára adunk meg három másik osztályt, amely ezt kibővíti. A "Getautomarka" osztály létre fogja hozni az objektumot a megadott információ szerint.

Absztrakt gyár programtervezési minta

Ez a programozási minta lehetőséget nyújt arra, hogy a közös témahez kapcsolódó gyártó metódusokat egyságba zárjuk, anélkül, hogy specifikálnánk azok konkrét osztályait. Szétválasztja egymástól objektumok egy csoportjának implementációját azok általános használatától és objektum összetételre hagyatkozik. Ennek a mintának a használata, lehetővé teszi egy rendszerben a konkrét típus implementációk kicseréléését anélkül, hogy az őket használó kódot módosítanánk.

A linkelt példában, elsőnek létrehozunk egy "bank" interfész, majd konkrét osztályokat, minden banknak. Ezt követően egy "kölcsön" abstact osztályt és konkrét osztályokat a kölcsön tipusoknak. Majd létrehozzuk az absztract gyárunkat, ami hivatkozni fog a bank és a kölcsöngyár osztályra.

Építő programtervezési minta

Ez a minta a gyártó és az absztrakt gyár metodustól abban különbözik, hogy alternatívát nyújt a teleszkópos konstruktor anti-mintára. Az anti-minta akkor jelentkezik, amikor a konstruktor paraméter-kombinációinka a száma növekszik és a konstruktor exponenciális listáját okozza. Az építő minta az építőt használja, ami egy másik objektum és minden egyes paramétert lépésről lépésre kap meg, majd egyben adja azt vissza. Ezt a programtervezési mintát használhatjuk lapos adatoknál. (HTML, SQL)

Prototípus programtervezési minta

A prototípus programozási minta lényege, a klónozás, az erekeli objektummal megegyző új objektum létrehozása. Ehhez az egyszerű értékkedés nem elegendő, úgyanis azzal csak másolás történik. A klónozásnak két típusa van a segély és a mély klónozás. A sekély klónozásban a hivatkozott objektumokat is másoljuk, mint elemi tipusú tulajdonságait. A mély klónozásnál az osztály által hivatkozott objektumokat is klónozzuk.

Egyke programtervezési minta

Ebben a programozási mintában egy objektumra korlátozzuk, a létrehozható példányok számát. Az egykének nem lehet publikus konstruktora, mivel ez esetben több példányt is létre lehetne hozni. De ha nincs konstruktur, akkor nem hozható létre a megoldást az osztályszintű metódusok jelentik. Ezeket akkor is lehet használni ha nincs példány. Szóval az egykének van osztály szintű metódusa, ami minden hívójának ugyan azt adja vissza. A létrehozáshoz pedig privát konstruktort



12. fejezet

Helló, Liskov!

12.1. Liskov helyettesítés sértése

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

Liskov helyettesítési elve a következő: 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.

Ez azt jelenti, hogy az S altípus mindenhol beilleszthető ahova a T ős, illetve minden tulajdonsága úgyan az. Ez nagyon jól hangzik mert rengeteget könnyít számunkra, mikor hasonló osztályokat szeretnénk létrehozni. Azonban ezt az elvet meg lehet sérteni, és a feladatban meg is fogjuk, a következő képen: olyan osztály csinálunk, ami felesleges utasítást örök az őstől. Mind a két programban (Java és C++) úgyan azt csináljuk, ráveszük a Pingvint, hogy repüljön, ami lehetetlen.

```
class Madar{
    public void repul()
    {
        System.out.println(super.getClass().getSimpleName() + " repül");
    }
}

class Sas extends Madar{};
class Golya extends Madar{};
class Pingvin extends Madar{};

public class liskov
{
    public static void main(String[] args)
    {
        Madar madar = new Madar();
        Sas sas = new Sas();
        Golya golya = new Golya();
        Pingvin pingvin = new Pingvin();
        sas.repul();
```

```
        golya.repul();
        pingvin.repul();
    }
}
```

Látható, hogy a programunkban a Pingvin osztály örökölte a Madar osztály repul függvényét, ami számára teljesen felesleges. Ezzel sértve Liskov helyettesítési elvét. A következő program úgyen ezt csinálja, csak C++ nyelven írodott.

```
#include <iostream>

using namespace std;
class Madar{
public:
    void repul(char nev[]){
        cout<<nev<<" repul\n";
    }
};

class Golya : public Madar
{
public:
    char nev[7]={"Golya"};
};

class Sas : public Madar
{
public:
    char nev[4]={"Sas"};
};

class Pingvin : public Madar
{
public:
    char nev[8]={"Pingvin"};
};

int main()
{
    Golya golya;
    Sas sas;
    Pingvin pingvin;
    golya.repul(golya.nev);
    sas.repul(sas.nev);
    pingvin.repul(pingvin.nev);
    return 0;
}
```

12.2. Szülő-gyerek

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

A következő feladatban egy programot láthatunk, amelyben a láthatjuk, hogy az alosztály képes használni az ős függvényeit, de ez fordítva nem igaz. Lássuk is a Java kódot:

```
class szulo{
    public void write(){
        System.out.println("Szülő");
    }
}
class gyerek extends szulo{
    public void kiir() {
        System.out.println("Gyerek");
    }
}
public class szulogyerek {
    public static void main(String[] args) {
        szulo egy = new szulo();
        gyerek ketto = new gyerek();
        egy.write();
        ketto.kiir();
        ketto.write();
    }
}
```

A program részletben látható, hogy a gyerek tudja használni a szulo write függvényét. Ellenben ha ezt fordítva szeretnénk eljátszani, az az egy szulo osztályú objektumot kérnénk meg arra, hogy a gyerek osztály kiir függvényét használja, a programunk nem fordulna, kompilálási hibába ütköznenek. Most pedig úgyan ez a program C++ nyelven:

```
#include <iostream>

using namespace std;

class Szulo{
    public: void ir(){cout<<"szulo";}
};

class gyerek : public Szulo
{
    public: void kiir(){
        cout<<"Gyerek";
    }
};

int main()
```

```
{  
    Szulo apa;  
    gyerek fia;  
    apa.ir();  
    fia.ir();  
    fia.kiir();  
    //apa.kiir();  
    return 0;  
}
```

12.3. EPAM: Liskov féle helyettesíthetőség elve, öröklődés

Adott az alábbi osztály hierarchia.

```
class Vehicle, class Car extends Vehicle, class Supercar extends Car
```

Mindegyik osztály konstruktorában történik egy kiíratás, valamint a Vehicle osztályban szereplő start() metódus mindegyik alosztályban felül van definiálva. Mi történik ezen kódok futtatása esetén, és miért?

```
Vehicle firstVehicle = new Supercar();  
firstVehicle.start();  
System.out.println(firstVehicle instanceof Car);  
Car secondVehicle = (Car) firstVehicle;  
secondVehicle.start();  
System.out.println(secondVehicle instanceof Supercar);  
Supercar thirdVehicle = new Vehicle();  
thirdVehicle.start();
```

Elsőnek is lássuk a teljes kodót:

```
class Vehicle{  
    public Vehicle() {  
        System.out.println("Vehicle");  
    };  
    public void start() {  
        System.out.println("first");  
    }  
};  
class Car extends Vehicle{  
    public Car() {  
        System.out.println("Car");  
    }  
    @Override  
    public void start() {  
        System.out.println("second");  
    }  
};  
class Supercar extends Car{
```

```
public Supercar()
{
    System.out.println("Supercar");
}
@Override
public void start() {
    System.out.println("third");
}
};

public class liskov {

    public static void main(String[] args) {
        Vehicle firstVehicle = new Supercar();
        firstVehicle.start();
        System.out.println(firstVehicle instanceof Car);
        Car secondVehicle = (Car) firstVehicle;
        secondVehicle.start();
        System.out.println(secondVehicle instanceof Supercar);
        //Supercar thirdVehicle = new Vehicle();
        //thirdVehicle.start();
    }
}
```

A kérdésünk az volt, hogy mi történik a kód futatását követően. Erre a válasz, hogy semmi, mivel nem fordul le a program. Az ok pedig, úgyan az, mint az előző feladatnál. Azt szeretnénk, hogy egy ős osztály a leszármazott osztály egyik függvényét használja. Ahogy látható a fenti kódban, az utolsó két sor kommentárba van, ezzel elérve, hogy leforduljon a program.

12.4. EPAM: Interfész, Osztály, Absztrakt Osztály

Mi a különbség Java-ban a Class, Abstract Class és az Interface között? Egy tetszőleges példával / példa kódon keresztül mutasd be őket és hogy mikor melyik koncepciót célszerű használni.

Osztály :

Egy osztály a következő képen hozunk létre: class <osztály neve>. Az osztályban deklarálhatunk változokat illetve függvényeket, az utóbbit kötelesek vagyunk kifejteni (az az leírni, hogy mit fognak csinálni.) Az osztálynak van konstruktora is, ami a nevére hivatkozik.

Egy egyszerű példa program egyszerű osztállyal:

```
class car2{
    public void brand() {
        System.out.println("Audi A" + 4);
    }
    public void color() {
        System.out.println("black");
    }
}
```

```
public class normalclass {  
  
    public static void main(String[] args) {  
        car2 car = new car2();  
        car.brand();  
        car.color();  
    }  
  
}
```

Interface :

Egy interfész a következő képen hozunk létre: interface < interface neve >. Az interfésszben deklarálhatunk változokat illetve függvényeket, az utóbbit azonban, az osztálytalálkozásban nem fejtjük ki. A függvény kifejtése majd csak az osztályon belül fog megtörténi. Az osztály úgy örökli az interfész tulajdonságait, hogy a létrehozásakor utána írjuk, hogy "implements < interface neve >". Egy osztály több interféssztől is örökölhet utasításokat, ez esetben vesszővel válasszuk el az interfések neveit az osztály deklarációjánál.

Egy egyszerű példa program interfésszel:

```
interface car{ //ez egy séma  
    public void getbrand();  
    public void getcolor();  
}  
  
class newcar implements car{ //osztályt csinálunk a séma alapján  
    public void getbrand(){  
        System.out.println("Audi A" + 4);  
    }  
    public void getcolor() {  
        System.out.println("Black");  
    }  
}  
  
public class ineface {  
  
    public static void main(String[] args) {  
        newcar car= new newcar(); //példányosítás  
        car.getbrand();  
        car.getcolor();  
    }  
}
```

Absztrakt osztály :

Egy absztrakt osztályt a következő képen hozunk létre: abstract < osztály neve >. Az absztrakt osztályban deklarálhatunk változokat illetve függvényeket. A függvényekkel azonban azt teszünk, amit akarunk, az

az vagy kifejtjük, vagy nem ez a programozon áll, azonban amennyiben nem fejtjük ki, akkor elé kell írni az abstract kulcs szót. Az osztály úgy örökli az absztrakt osztály tulajdonságait, hogy a létrehozásakor utána írjuk, hogy "extends < absztrakt osztály neve >". Egy osztály csak és csak is egy absztrakt osztálytól örökölhet.

Egy egyszerű példa program absztrak osztállyal:

```
abstract class car1{ //ez egy séma
    public abstract void getbrand();
    public void getcolor()
    {
        System.out.println("Black");
    }
}

class newcar1 extends car1{ //osztalyt csinálunk a séma alapján
    public void getbrand(){
        System.out.println("Audi A" + 4);
    }
}

public class abstract1{

    public static void main(String[] args) {
        newcar1 car= new newcar1(); //példányosítás
        car.getbrand();
        car.getcolor();
    }
}
```



13. fejezet

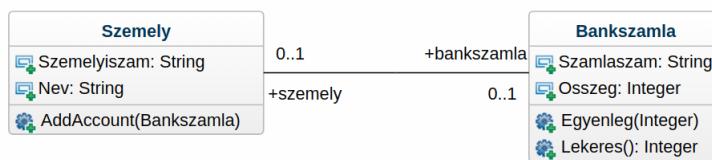
Helló, Mandelbrot!

13.1. Forward engineering UML osztálydiagram

UML-ben tervezünk osztályokat és generálunk belőle forrást!

A feladatban a GenMyCode nevezetű weboldalt fogom használni, ahol meg lehet építeni az UML fát, majd legenerálni vele az osztály fájlokat.

Jelenleg egy egyszerű példát fogok létrehozni, amiben van két osztály, az egyikben a Bankszámla adatokat, illetve a pénzfelvétel lesz, a másikban pedig bankszámlát hozunk létre. Az UML a következő képen nézz ki:



Az UML megszerkezése után a következő feladatunk legenerálni az osztályokat. Ha a fentebb említett GenMyCode nevezetű online felületet használjuk, akkor ez számunkra nagyon egyszerű. Bal oldalt menüpontot váltunk a Generators fülre kattintva. Itt a két megjelent mappából a következő útvonalon haladunk: Sample generators/UML/Java és én itt az uml2java generátort használtam. Erre kattintva, egy új fül nyílik meg a munka felületen, egy hosszú kód amit, ha futattunk, akkor generálja és le is tölti az osztályokat.

```
package Models ;  
  
/**  
 * <!-- begin-user-doc -->  
 * <!-- end-user-doc -->  
 * @generated  
 */  
  
public class Bankszamla  
{  
    /**  
     * <!-- begin-user-doc -->  
     * <!-- end-user-doc -->  
     * @generated  
     * @ordered  
     */  
    public String Szamlaszam;  
  
    /**  
     * <!-- begin-user-doc -->  
     * <!-- end-user-doc -->  
     * @generated  
     * @ordered  
     */  
    public int Osszeg;  
  
    /**  
     * <!-- begin-user-doc -->  
     * <!-- end-user-doc -->  
     * @generated  
     */  
    public Bankszamla(){
```

A fenti képen látható az egyik osztály, amit generáltunk. Észrevehető a rengetek megjegyzés mellett, hogy a változok neve tényleg az, amit megadtunk az UML ábrán. Innen a következő feladatunk definiálni a függvényeket és egy futatható programot írni. Nem szeretném nagyon részletezni a kódot, a lényege, hogy létrehozz egy bankszámlafiókot egy előre beépített összeggel rajta, és erről lehet készpénzt lekérni.

```
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
//Bankszamla.java  
public class Bankszamla {  
  
    public String Szamlaszam;  
    public int Osszeg = 60000;  
  
    public Bankszamla() {  
        super();  
    }  
  
    public void Egyenleg(int parameter) {  
        System.out.println("Az Ön egyenlege:" + parameter);  
    }  
  
    public int Lekeres() {  
        System.out.println("Írja be a kivenni kívánt összeget.");  
        String levon;  
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in) ←  
            );  
        try {  
            levon = br.readLine();  
            int levonas = Integer.parseInt(levon);  
            if (levonas < 0)  
                System.out.println("Érvénytelen összeg");  
            else if (levonas > Osszeg)  
                System.out.println("Nem áll rendelkezésre elég összeg");  
            else {  
                Osszeg = Osszeg - levonas;
```

```
        System.out.println("Sikeres tranzakció");
    }
} catch (IOException e) {
    System.out.println("Nem adott meg összeget");
}

return 0;
}

}
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
//Szemely.java
public class Szemely
{

    public String Szemelyiszam;

    public String Nev;

    public Szemely() {
        super();
    }

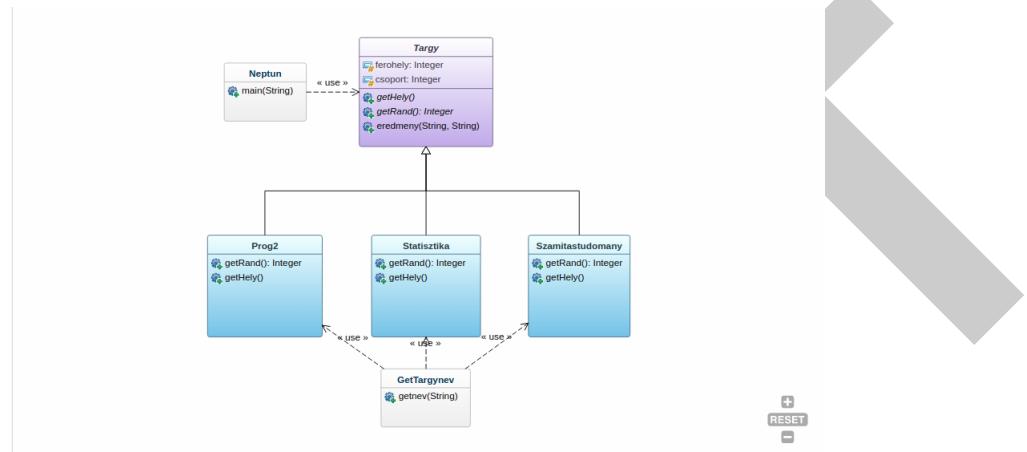
    public void AddAccount(Bankszamla parameter) {
        System.out.println("Adja meg a személyi azonosító számát!");
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in) ←
            );
        try {
            Szemelyiszam=br.readLine();
        } catch (IOException e) {
            System.out.println("Nem adott meg adatot");
        }
        System.out.println("Adja meg a nevét!");
        try {
            Nev=br.readLine();
        } catch (IOException e) {
            System.out.println("Nem adott meg adatot");
        }
    }

}
```

13.2. EPAM: Neptun tantárgyfelvétel modellezése UML-ben

Modellezd le a Neptun rendszer tárgyfelvételéhez szükséges objektumokat UML diagramm segítségével.

Az előző feledathoz hasonlóan itt is egy UML táblát fogunk létrehozni, ahol szintén a GenMyCode weboldalt fogom használni, az egyszerűbb kivitelezés céljából. Az UML tábla a következő képen nézz ki:



Induljunk a főosztálytól, ami a Neptun lesz, úgyan is ebben található a main függvény. Ez fogja használni a Targy abstract osztályt, aminek van két változója (ferohely, csoport) és három függvénye(getHely, getRand, eredmény). Ezt az abstract osztály három tantárgy osztály fogja örökölni (Prog2, Statisztika, Szamitastudomány), amelyeknek van két-két függvénye. A GetTargynev osztály, pedig egy függvényel rendelkezik, ami visszadja az egyes tantárgyak nevét.

13.3. EPAM: Neptun tantárgyfelvétel UML diagram implementálása

Implementáld le az előző feladatban létrehozott diagrammot egy tetszőleges nyelven.

Hogy meggyorsítsük kicsit a feladatunk az előző feladatban létrehozott UML táblát legenerálom, ahogyan az első feladatban leírtam. Így az osztályok létre vannak hozva és csak a függvényeket kell kifejteni, illetve a program és felhasználó közötti kommunikációt kell létre hozni.

```

public abstract class Targy
{
    protected int ferohely;
    protected int csoport;

    public abstract void getHely();

    public abstract int getRand();

    public void eredmény(String muvelet, String nev) {
        if (muvelet.equalsIgnoreCase("Felvész"))
        {
            if (getRand() < csoport * ferohely)
System.out.println("Sikeresen felvettet a " + nev + " tárgyat a " + (int) (Math.random() * (csoport - 1) + 1) + ". csoportba.");
        }
    }
}
  
```

```
        else
            System.out.println("A kurzusok beteltek.");
        for(int i=1; i<=csoport; i++) {
            System.out.println(i+". csoport: " + ferohely + "/" + ferohely);
        }
    }
    else if(muvelet.equalsIgnoreCase("LEAD")) {
        System.out.println("Sikeresen leadtad a " + (int) (Math.random() * (csoport - 1) + 1) + ". " + nev + " csoport tárgyat.");
    }
    else
        System.out.println("Nincs ilyen parancs. Próbáld újra.");
}
}
```

A fenti kód részlet az abstract Targy osztályunk kifejtése. A legenerálásnak köszönhetően, a feladatunk itt csak az "eredmeny" függvény kifejtése. Ez a függvény annyit tesz, hogy kiírja sikeres volt-e a tárgyfelvétel és leadás vagy nem. Ha rossz parancsot gépelünk be, akkor, hiba üzenetet küldünk vissza, ezzel értesítve a felhasználót, hogy olyan utasítást szeretne végrehajtani, amit nem lehetséges.

Mivel a tantárgyakat szimbolizáló osztályok felépítése és működése úgyan olyan, egyszere ki lehet fejteni mind a hármat. Ahogy a következő kód részletben látható lesz, a getRand vissza ad egy int számot, ami azt fogja jelképezni, hogy hányan vették fel az adott tantárgyat. Ez a szám véletlen szerű lesz a következő képletet használva: $(\text{Math.random()} * (\text{felső határérték}) - (\text{alsó határérték}+1)) + (\text{alsó határérték})$. A getHely pedig a férőhelyek számát adja vissza. Akkor lássuk a három osztályt:

```
public class Prog2 extends Targy
{
    public int getRand() {
        return (int) ((Math.random() * (120-60+1)) + 60);
    }

    public void getHely() {
        ferohely = 18;
        csoport = 5;
    }
}

public class Statisztika extends Targy
{
    public int getRand() {
        return (int) ((Math.random() * (100-70+1)) + 70);
    }

    public void getHely() {
        ferohely = 22;
        csoport = 4;
    }
}
```

```
}
```

```
public class Szamitastudomany extends Targy
{
    public int getRand() {
        return (int) ((Math.random() * (100-70+1)) + 70);
    }

    public void getHely() {

    }
}
```

A GetTagynev osztályban létrehozunk egy listát, amiben eltároljuk az összes tárgyat, majd ezt követően a beolvasott tárgynévet összehasonlítjuk a lista elemekkel. Ha találat van a listában, visszadja a tárgy nevét, ellenkező esetben egy null értéket.

```
import java.util.ArrayList;

public class GetTargynev
{
    public Targy getnev(String nev) {
        ArrayList<String> targyak = new ArrayList<String>();
        targyak.add("statisztika");
        targyak.add("prog2");
        targyak.add("Szamitastudomany");

        String vantargy = null;

        for(String i : targyak) {
            if(nev.equalsIgnoreCase(i))
                vantargy = nev;
        }
        if(vantargy != null) {
            if(nev.equalsIgnoreCase("PROG2")) {
                return new Prog2();
            }
            else if(nev.equalsIgnoreCase("STATISZTIKA")) {
                return new Statisztika();
            }
            else if(nev.equalsIgnoreCase("SZAMITASTUDOMANY")) {
                return new Szamitastudomany();
            }
        } else {
            System.out.println("Helytelen tárgynév vagy művelet.");
        }
    }
}
```

```
    return null;
}
}
```

A fő osztályban, azaz a Neptunban, a program kommunikál a felhasználóval, beolvassa a tárgyat illetve, hogy mit szeretne tenni a tanuló az adott tárgyal (felvenni vagy leadni). Abban az esetben, ha rossz tárgy nevet adunk hiba üzenetet kapunk és újra indul az adatok beolvasása.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Neptun {
    public static void main(String[] args) throws IOException {
        GetTargynev targyMelyik = new GetTargynev();
        BufferedReader targyfelvetel = new BufferedReader(new InputStreamReader ←
            (System.in));

        System.out.println("Tárgyfelvétel\n");
        System.out.println("Ha végeztél írd be az 'exit' kulcsszót.");
        boolean fut;

        do {
            System.out.println("Írd be egy tárgy névét ( prog2, statisztak, ←
                szamitastudomány ) : ");
            String targynev;
            try {
                targynev = targyfelvetel.readLine();
                System.out.println("Felvész vagy Lead?: ");
                String muvelet = targyfelvetel.readLine();

                Targy t = (Targy) targyMelyik.getnev(targynev);

                if(t != null)
                {
                    t.getRand();
                    t.getHely();
                    t.eredmeny(muvelet, targynev);
                }
                else
                    System.out.println("Érvénytelen tárgynév");
                System.out.println(
                    "Ha nem akarsz további műveleteket végezni\nírd be az 'exit' ←
                    kulcsszót egyébként nyomj enter-t:");
                String exit = targyfelvetel.readLine();
                if (exit.equalsIgnoreCase("EXIT"))
                    fut = false;
                else

```

```
        fut = true;
    } catch (IOException e) {
        System.out.println("Érvénytelen utasítás");
    }
    System.out.println(
        "Ha nem akarsz további műveleteket végezni\nnird be az 'exit' ←
        kulcsszót egyébként nyomj enter-t:");
    String exit = targyfelvetel.readLine();
    if (exit.equalsIgnoreCase("EXIT"))
        fut = false;
    else
        fut = true;
} while (fut == true);

System.out.println("A tárgyfelvétel megtörtént.");
}

}
```

13.4. EPAM: OO modellezés

Írj egy 1 oldalas esszét arról, hogy OO modellezés során milyen elveket tudsz követni (pl.: SOLID, KISS, DRY, YAGNI)

Ebben a feladatban az objektum orientált programozás egyes elveiről olvashatunk. Kezdjük is a SOLID-val.

Elsőnek is lássuk, honnan kapta a nevét ez az elv. A SOLID öt tervezési alapelvek kezdeti betűiből alkotott mozaik szó. Az öt alap elv Egyetlen felelőség elve (Single Responsibility Principle), Nyílt/zárt elv (Open/Closed Principle), Liskov helyettesítési elve (Liskov substitution Principle), Interfész elválasztási elv (Interface Degregation Principle) illetve a Függőség megfordítási elv (Dependency Inversion Principle). Beszéljünk most kicsit ezzekről az elvekről.

Egyetlen felelőség elve: Az elv azt mondja, hogy egy osztálynak vagy objektumnak csak egyetlen egy oka legyen változni. Ez az egyetlen ok azt jelenti, hogy egyetlen feladata lehet csak az objektumnak. Ha több feladattal rendelkezik, akkor ezek csatoltá válnak, több oka lesz a változásra, illetve egy felelőségen (okban) történő változás, gátolhatja vagy gyengítheti az osztály képességét a többi feladat végrehajtásában.

Nyílt/zárt elv: Az elv szerint az osztály vagy modul nyílt a kiterjesztésre az az kiterjeszhető a viselkedése, de zárt a módosításra, tehát a kiterjesztés nem változtatja meg a forrás vagy bináris kódját. Az előző fejezetben ilyen volt a gyártó metodus.

Liskov helyettesítési elve: Szerintem ezt az elvet mindenki ismeri, de azért írjuk le mit is jelent. Az elv azt mondja ki, hogy ha S típus a T altípusa, nem változhat meg a program működése, ha a programban a T típusú elemeket S típussal helyettesítjük.

Interfész elválasztási elv: Az elv szerint nem szabad rákényszeríteni az osztályokat arra, hogy olyan metódusoktól függjenek, amit nem is használnak. Itt meg szeretnék írni két fogalomról, ami ide tartozik. Az első a Vastag interfész (Fat Interface), amit azokra az osztályokra használunk, amelyeknek a szükségesnél több

tagfüggvényel és baráttal rendelkezik. Az említett elv, ezek szétválasztásával foglalkozik, az az megpróbálja az osztály több osztályra bontani, azon módon, hogy az új osztályok csak egy-egy klienst szolgáljanak ki. A másik fogalom pedig az Interfész szennyezés (interface pollution), ami azt jelenti, hogy az interfész felesleges metódusokat tartalmaza "szennyezve" van.

Függőség megfordítási elv: Az elv azt mondja ki, hogy a magas szintű modulok ne függjenek az alacsony szintű moduluktól, hanem mindenkor absztrakciótól függjen. Az absztrakció pedig ne függjen a részletektől, hanem fordítva legyen. A magas szintű modulok adják az alkalmazás identitását. Erre a megfordításra több ok miatt is szükség volt. Az első, hogy ha nem történt volna a megfordítás, az alacsony szintű modulok megváltoztatása kihatással lett volna a magas szintű modulokra (már pedig ezek határozzák meg a programot). A magas szintű modulokat szeretnénk többször újra felhasználni, ha ezek függnek az alacsony szintű moduluktól, a dolgunk jóval nehezebb, úgyanis ez esetben ezzeken keresztül kellene hivatkoznunk.

Egy másik modellezési elv a KISS (Keep it simple, stupid), aminek a célja az egyszerűségre való törekvés.

Szintén modellezési elv a DRY (Don't Repeat Yourself). Ahogy az angol nevéből következtethet, a lényege, hogy próbáljuk meg nem ismételni magunkat. Itt szeretném megemlíteni az ismétlés pár fajtáját: Kényszerített ismétlés (imposed duplication), a fejlesztő úgy érzi, hogy a környezet megköveteli az ismétlést; Nem szándékos ismétlés (inadvertent duplication) a fejlesztő nem veszi észre, hogy információt dublikál; Türelmetlen ismétlés (impatient duplication) a fejlesztő lustaságból ismétel; Fejlesztők közötti ismétlés (interdeveloper duplication) egy csapatban vagy több különböző csapatban többen dublikálnak egy információt. A DRY ellentéte a WET (We Enjoy Typing).

Az utolsó modellezési elv, amiről írni szeretnék a YAGNI (You aren't gonna need it). Az elv lényege, hogy ne deklaráljunk előre semmit, ha nem használjuk fel azonnal.

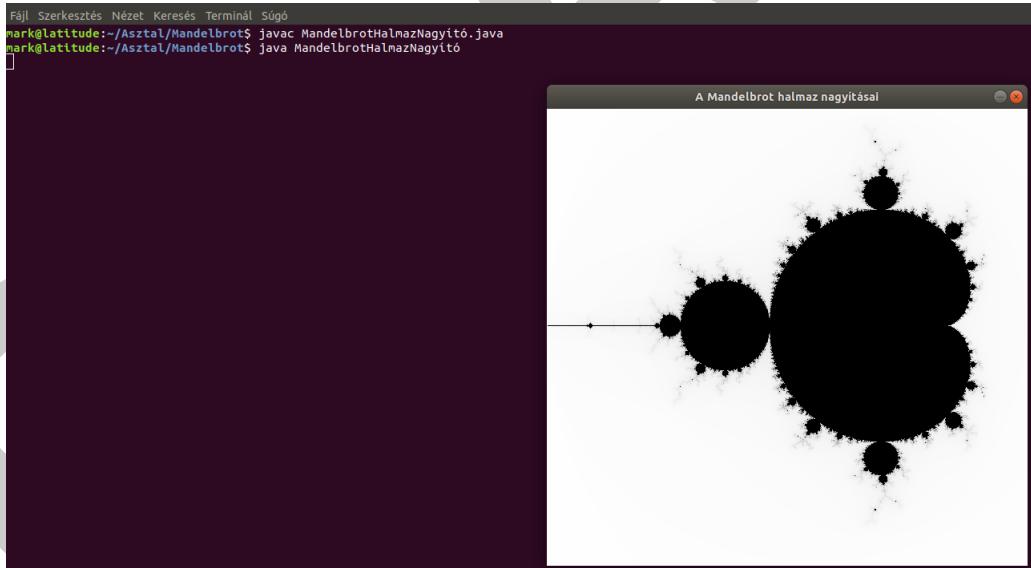
14. fejezet

Helló, Chomsky!

14.1. Encoding

Fordítsuk le és futtassuk a Javat tanítók könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékeszetes betűket!

Ahogy azt az utasításban is észrevehetük, a feladatunk csupán lefordítani és lefutatni, az előző évről is ismert Mandelbrot halmazos feladat java kódját. Az érdekesség itt az, hogy a megszokottól eltérően, most ékeszetes betűket is használunk az osztály nevében. A kérdés, hogy mi fog történi? Lefút a program vagy esetleg kompilálási hibát fog dobni? Nézzük meg.



Mint látható a program ékeszetes karakterek használatával is természetesen, hiba nélkül lefordul. Ez pedig azért van, mivel a JAVA nyelv minden UNICODE karaktert elfogad. Egy osztály vagy változó név adásakor csupán erre a pár szabályra kell figyeljünk: 1) nem kezdődhet számmal; 2) egyetlen egy karakter sor lehet (az az, ha azt a nevet szeretnénk adni, hogy Az Első Osztályom, akkor ezt egyben kell írnunk AzElsőOsztályom); 3) nem használhatóak a fentartot karakter sorok, kulcs szók (ilyen karakter sor a for, while, if, else) úgyan is ezeknek más jelentésük van.

A JAVA azért támogatja a teljes UNICODE karakter listát, mivel így minden programozó a saját anyanyelvén adhat nevet a programban szereplő osztályok és változók számára.

14.2. EPAM: Order of everything

Collection-ok rendezése esetén jellemzően futási időben derül ki, ha olyan típusú objektumokat próbálunk rendezni, amelyeken az összehasonlítás nem értelmezett (azaz T típus esetén nem implementálják a Comparable<T> interface-t). Pl. ClassCastException a Collections.sort() esetében, vagy ClassCastException a Stream.sorted() esetében. Írj olyan metódust, amely tetszőleges Collection esetén vissza adja az elemeket egy List-ben növekvően rendezve, amennyiben az elemek összehasonlíthatóak velük azonos típusú objektumokkal. Ha ez a feltétel nem teljesül, az eredményezzen syntax error-t. Például: List<Integer> actualOutput = createOrderedList(input); Ahol az input Collection<Integer> típusú. Természetesen más típusokkal is működnie kell, feltéve, hogy implementálják a Comparable interface-t.

A feladatban az epam által megosztott kodról fogok beszélni.

```
public void testOrderShouldReturnExpectedListWhenCollectionIsPassed( ←
    Collection<Integer> input, List<Integer> expectedOutput) {
    // Given as parameters

    // When
    // createOrderedList(List.of(new OrderOfEverythingTest()));
    // ^ ez piros, az OrderOfEverythingTest nem implementálja a ←
    // Comparable<OrderOfEverythingTest> -et
    List<Integer> actualOutput = createOrderedList(input);

    // Then
    assertThat(actualOutput, equalTo(expectedOutput));
}
```

Ahogy a nevében is látható, ez egy teszt függvény. Paraméterként a rendezett Collection-t illetve a várt Collectiont kapta meg.

```
private static Stream<Arguments> collectionsToSortDataProvider() {
    return Stream.of(
        Arguments.of(Collections.emptySet(), Collections.emptyList()),
        Arguments.of(Set.of(1), List.of(1)),
        Arguments.of(Set.of(2,1), List.of(1,2))
    );
}
```

A fent említett paraméterekre, itt láthatunk konkrét példákat, pontosabban hármat. Ezek a paraméterek listák, melyek zéró, egy illetve két elemet tartalmaznak. A legelső programrész azt nézi meg, hogy a programunk helyesen rendezi-e azokat. Fontos tisztázni, hogy csak akkor lehet rendezni egy listát, ha azok homogének, azaz nem tartalmaznak több típust, hanem azonosokat.

```
private <T extends Comparable<T>> List<T> createOrderedList(Collection<T> ←
    input) {
    return input.stream()
        .sorted()
        .collect(Collectors.toList());
}
```

Ez a program rész rendezi a lista elemeit, majd egy Collectioné alakítja azokat.

14.3. EPAM: Bináris keresés és Buborék rendezés implementálása

Implementálj egy Java osztályt, amely képes egy előre definiált n darab Integer tárolására. Ennek az osztálnak az alábbi funkcionálisokkal kell rendelkeznie: Elem hozzáadása a tárolt elemekhez; Egy tetszőleges Integer értékről tudja eldönteni, hogy már tároljuk-e (ehhez egy bináris keresőt implementálj); A tárolt elemeket az osztályunk be tudja rendezni és a rendezett (pl növekvő sorrend) struktúrával vissza tud térni (ehhez egy buborék rendezőt implementálj)

Ez a program már két fájlból áll, a main osztályból, illetve egy IntegerCollection.java fájlból. Kezdjük a mainnel.

```
public class Main {  
  
    public static void main(String[] args) {  
        IntegerCollection collection = new IntegerCollection(3);  
        collection.add(0);  
        collection.add(2);  
        collection.add(1);  
        System.out.println(collection);  
        collection.sort();  
        System.out.println(collection);  
        System.out.println(collection.contains(0));  
        System.out.println(collection.contains(1));  
        System.out.println(collection.contains(2));  
        System.out.println(collection.contains(3));  
        System.out.println(collection.contains(4));  
    }  
  
}
```

Mint látható a main fájlunk nem rendelkezik túl nagy terjedémmel. Itt létrehozunk egy három elemet tartalmazó IntegerCollectiont, ehhez hozzáadunk három elemet (0-t,2-t,1-t), kiíratjuk, rendezük, újra kiíratjuk, majd megnézzük, hogy 0-tól 4-ig melyik elemek szerepelnek a Collectionben.

```
public class IntegerCollection {  
  
    int[] array;  
    int index = 0;  
    int size;  
    boolean sorted = true;  
  
    public IntegerCollection(int size) {  
        this.size = size;  
        this.array = new int[size];  
    }  
  
    public IntegerCollection(int[] array) {  
        this.size = array.length;  
        this.index = this.size;  
        this.array = array;  
        this.sorted = false;  
    }  
  
    public void add(int value) {  
        if (!sorted) {  
            sort();  
        }  
        array[index] = value;  
        index++;  
    }  
  
    public boolean contains(int value) {  
        for (int i = 0; i < index; i++) {  
            if (array[i] == value) {  
                return true;  
            }  
        }  
        return false;  
    }  
  
    public void sort() {  
        for (int i = 0; i < index - 1; i++) {  
            for (int j = i + 1; j < index; j++) {  
                if (array[i] > array[j]) {  
                    int temp = array[i];  
                    array[i] = array[j];  
                    array[j] = temp;  
                }  
            }  
        }  
        sorted = true;  
    }  
  
    public void print() {  
        for (int i = 0; i < index; i++) {  
            System.out.print(array[i] + " ");  
        }  
        System.out.println();  
    }  
}
```

```
}
```

IntegerCollection.java fájlban létrehozzuk az IntegerCollection osztályt, melynek négy eleme van: egy vektor, amiben az elemek lesznek; index, amely azt mutatja hanyadik elemnél járunk; size, amely a vektor méretét definiálja; sorted, amely igaz, ha a vektor rendezet és hamis ha nem az. Ezek mellett láthatunk két konstruktort, melyek közül az elsőnél, csak a Collection méretét adjuk meg (mint a mainben) és a másodiknál pedig egy vektort tudunk átadni (itt a program el végzi a szükséges műveleteket, átveszi a vektort és a méretét).

```
public void add(int value) {
    if (size <= index) {
        throw new IllegalArgumentException("The collection is full");
    }
    sorted = false;
    array[index++] = value;
}
```

Az add az első függvény, amit létrehozunk. Ennek a feladata az új elem hozzáadása lesz. Mint látható a függvény leellenörzi, hogy tele van-e a listánk. Ha igen, akkor hibaüzenetet ad nekünk, ha nem akkor tovább lép, beteszi az elemet, a sorted hamis lesz (mivel nem tudjuk, hogy így rendezve maradt-e a listánk) és növeli az indexet.

```
public boolean contains(int value) {
    if (!sorted) {
        sort();
    }

    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (array[mid] == value) {
            return true;
        }

        if (array[mid] < value) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return false;
}
```

A contains függvény már egy picit nagyobb terjedelemmel rendelkezik. Ennek a feladata leellenörizni, hogy a paramáterként kapott érték (value) benne van-e a Collectionben. Az első lépés, amit tesz, hogy amennyiben nincsenek rendezve az elemek, rendezzi őket. Ezt követően az ismert bináris keresést használja. Amennyiben megtalálja az elemet igaz értékel tér vissza, különben hamis értéket add.

```
public int[] sort() {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
    sorted = true;
    return array;
}
```

A sort függvény, mint a neve is mondja rendezni fogja az elemeket. Ehhez a régen tanult buborék módszert használja.

14.4. EPAM: Saját HashMap implementáció

Írj egy saját java.util.Map implementációt, mely nem használja a Java Collection API-t. Az implementáció meg kell feleljen az összes megadott unit tesztnek, nem kell tudjon kezelni null értékű kulcsokat és a “keySet”, “values”, “entrySet” metódusok nem kell támogassák az elem törlést. Plusz feladatok: 1. az implementáció támogat null kulcsokat, a “keySet”, “values”, “entrySet” metódusok támogatják az elem törlést.

```
public class ArrayMap<K, V> implements Map<K, V> {

    private static final int INITIAL_SIZE = 16;
    private static final String NULL_KEY_NOT_SUPPORTED = "This Map ↵
        implementation does not support null keys!";

    private int size = 0;
    private K[] keys = (K[]) new Object[INITIAL_SIZE];
    private V[] values = (V[]) new Object[INITIAL_SIZE];

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean isEmpty() {
        return size <= 0;
    }
}
```

A feladatban egy saját MAP-t fogunk létrehozni. Kezdés kép létrehozzunk pár statikus változót. Ezek azért lettek létrehozva, hogy átláthatóbb legyen a program, azok számára is, akik elsőnek látják. Ezt követően

létrehozzuk a kulcs érték párosokat. Az első két metódus, amit átírunk a size és az isEmpty, melyek közül az első a méretet, a második pedig azt adja vissza, hogy üres-e.

```
@Override
public boolean containsKey(Object key) {
    Objects.requireNonNull(key, NULL_KEY_NOT_SUPPORTED);

    return searchItemInArray(key, keys, Object::equals) != -1;
}

@Override
public boolean containsValue(Object value) {
    int valueIndex = searchItemInArray(value, values, Object::equals);
    return valueIndex > -1 && keys[valueIndex] != null;
}
```

Ez a két felülírt metódus azt nézi meg, hogy szerepel-e a kulcs vagy az érték a listában.

```
@Override
public V get(Object key) {
    Objects.requireNonNull(key, NULL_KEY_NOT_SUPPORTED);
    if(size <= 0) {
        return null;
    }

    int keyIndex = searchItemInArray(key, keys, Object::equals);
    if (keyIndex > -1) {
        return values[keyIndex];
    }

    return null;
}

@Override
public V put(K key, V value) {
    Objects.requireNonNull(key, NULL_KEY_NOT_SUPPORTED);

    int keyIndex = searchItemInArray(key, keys, Objects::equals);
    if (keyIndex < 0) {
        keyIndex = findFirstEmptyPlace();
        if (keyIndex < 0) {
            expandArrays();
        }
        keyIndex = size;
    }

    V prevValue = values[keyIndex];

    keys[keyIndex] = key;
    values[keyIndex] = value;
    size++;
}
```

```
        return prevValue;
    }
    private <I> int searchItemInArray(I item, I[] array, BiPredicate<I, I> equalFunction) {
        for (int index = 0; index < array.length; index++) {
            if (equalFunction.test(item, array[index]))
                return index;
        }
        return -1;
    }

    private int findFirstEmptyPlace() {
        return searchItemInArray(null, keys, Objects::equals);
    }

    private void expandArrays() {
        int expandedSize = size * 2;

        keys = Arrays.copyOf(keys, expandedSize);
        values = Arrays.copyOf(values, expandedSize);
    }
}
```

A get szintén megkeresi az elemet a listában. A put beleteszi, de elsőnek megnézi, hogy biztos nincs-e benne, ezt követően meg keresi az első üres helyt, ahova beteheti. Ezek a metódusok, használnak pár a program végén megírt függvényt. Ezek a searchItemInArray, findFirstEmptyPlace, expandArrays.

```
@Override
public void putAll(Map<? extends K, ? extends V> m) {
    m.forEach(this::put);
}

@Override
public void clear() {
    Arrays.fill(keys, null);
    Arrays.fill(values, null);
    size = 0;
}

@Override
public Set<K> keySet() {
    Set<K> result = new HashSet<>();
    for(K i : keys) {
        if (i != null) {
            result.add(i);
        }
    }
}
```

```
        return result;
    }

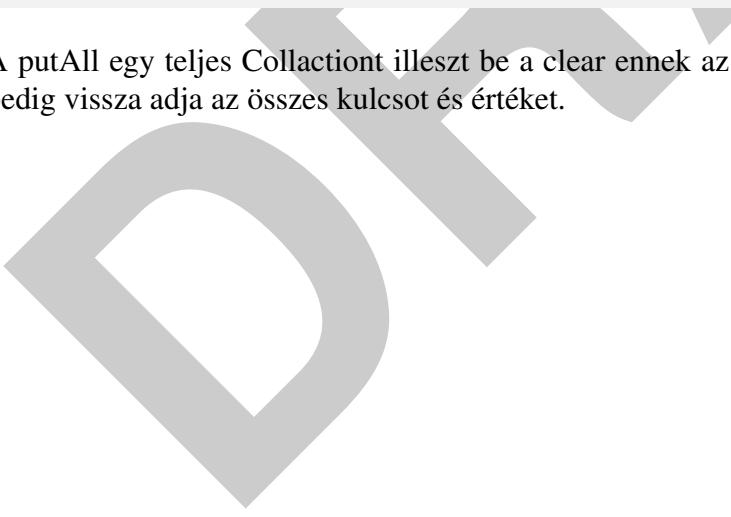
@Override
public Collection<V> values() {
    Collection<V> result = new ArrayList<>();
    for(V i : values) {
        if (i != null) {
            result.add(i);
        }
    }

    return result;
}

@Override
public Set<Entry<K, V>> entrySet() {
    Set<Entry<K, V>> result = new HashSet<>();
    for(int i = 0; i < keys.length; ++i) {
        K key = keys[i];
        if (key != null) {
            V value = values[i];
            result.add(new AbstractMap.SimpleEntry<>(key, value));
        }
    }

    return result;
}
```

A putAll egy teljes Collactiont illeszt be a clear ennek az elentéte, a teljes listát törli. A keyset és values pedig vissza adja az összes kulcsot és értéket.



15. fejezet

Helló, Stroustrup!

15.1. JDK osztályok

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

Feladatunkban a Boost könyvtárat felhasználva kell kiíratnunk egy C++ program segítségével a JDK összes osztályát. Az első feladat a Boost könyvtárat feltenni, ha még nincs a gépünkön akkor azt az alábbi módon tudjuk megtenni: "sudo apt-get install libboost-all-dev". Azért van erre szükségünk, ugyanis az elérési utat és a kiterjesztést is ezel tudjuk elvégezni.

```
#include <iostream>
#include <string>
#include <map>
#include <iomanip>
#include <fstream>
#include <vector>

#include <boost/filesystem.hpp>
#include <boost/filesystem/fstream.hpp>
#include <boost/program_options.hpp>
#include <boost/tokenizer.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

using namespace std;
using namespace boost::filesystem;

int szamlalo = 0;

void read_file ( boost::filesystem::path path, std::vector<std::string> <-- acts )
{
    if ( is_regular_file ( path ) ) {

        std::string ext ( ".java" );
    }
}
```

```
if ( !ext.compare ( boost::filesystem::extension ( path ) ) ) {  
  
    cout<<path.string()<<'\n';  
    std::string actpropspath = path.string();  
    std::size_t end = actpropspath.find_last_of ( "/" );  
    std::string act = actpropspath.substr ( 0, end );  
  
    acts.push_back (act);  
    szamlalo++;  
}  
  
} else if ( is_directory ( path ) )  
    for ( boost::filesystem::directory_entry & entry :  
        boost::filesystem::directory_iterator ( path ) )  
        read_file ( entry.path(), acts );  
}  
  
int main ( int argc, char *argv[] )  
{  
string path="src";  
vector<string> acts;  
read_file(path,acts);  
cout<<"szamlalo: "<<szamlalo<< std::endl;  
return 0;  
}
```

A programunk azt fogja csinálnni, hogy kírja a ".java" végzodés "ú fájlokat, és ezek elérési útját, majd a végén hogy összesen hány darab van ezekb " ol. A program- " ban elosz " or elvégezzük a szükséges include-kat a boost mappából és amikre még szükségünk lesz (vector, "iostream ...stb). Ezek után elosz " or is létre kell hozni egy egész típusú változót amivel számolni fogjuk a "fájlok számát mégpedig úgy, hogyha van egy ".java" végzodés "ú fájl, akkor növeljük egyel az értékét. A re- " ad_file() függvény segítségével fogjuk beolvasni a mappákat és azok tartalmait. If-ek használatával fogjuk megvizsgálni a kiterjesztést. És itt adjuk meg a zelérési utat is. A vectorba fogjuk belevenni az összes .java fájlt az elérési úttal együtt. A legvégén pedig ezeket kiíratjuk.

15.2. EPAM: It's gone. Or is it?

Adott a következő osztály:

```
public class BugousStuffProducer {  
private final Writer writer;  
public BugousStuffProducer(String outputFileName) throws  
IOException {  
writer = new FileWriter(outputFileName);  
}  
public void writeStuff() throws IOException {  
writer.write("Stuff");
```

```
}
```

```
@Override
```

```
public void finalize() throws IOException {
```

```
    writer.close();
```

```
}
```

```
}
```

Mutass példát arra az esetre, amikor előfordulhat, hogy bár a program futása során meghívztuk a writeStuff() metódust, a fájl, amibe írtunk még is üres.

A fenti kód részlettel azért fordulhat elő az, hogy nem íratunk ki semmit a fájlba, a meghívás ellenére, mivel eldobtuk az IOException-t. Az az a program nem nézi meg, hogy sikerült-e megnyitni a fájlt vagy sem. A félév során azt is tapasztalhattuk, hogy ha nem dobjuk el az IOException-t, akkor minden kéri, hogy try-catchbe helyezük a beolvasásokat, ez itt elmaradt ezért van lehetőség arra, hogy fájlbaírás nélkül fut le a program. Az orvoslása nagyon egyszerű, ahol a fájl megnyitásra kerül és ahol írunk bele az IOException-t nem dobjuk el, illetve a fájl megnyitását egy try-ba teszük. Az utóbbi lépés azért szükséges, mivel így nem is fog tovább menni a program, ha nincs fájl megnyitva. A javítót verzió:

```
public class FinalizeFixedExample {
```

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

```
        try (BugousStuffProducer stuffProducer = new BugousStuffProducer("←  
someFile.txt")) {
```

```
            stuffProducer.writeStuff();
```

```
        }
```

```
    }
```

```
    private static class BugousStuffProducer implements AutoCloseable {
```

```
        private final Writer writer;
```

```
        public BugousStuffProducer(String outputFileName) throws ←  
IOException {
```

```
            writer = new FileWriter(outputFileName);
```

```
        }
```

```
        public void writeStuff() throws IOException {
```

```
            writer.write("Stuff");
```

```
        }
```

```
        @Override
```

```
        public void close() throws Exception {
```

```
            writer.close();
```

```
        }
```

```
    }
```

```
}
```

15.3. EPAM: Kind of equal

Adott az alábbi kódrészlet.

```
p// Given
String first = "...";
String second = "...";
String third = "...";
// When
var firstMatchesSecondWithEquals = first.equals(second);
var firstMatchesSecondWithEqualToOperator = first == second;
var firstMatchesThirdWithEquals = first.equals(third);
var firstMatchesThirdWithEqualToOperator = first == third;
```

Változtasd meg a String third = "..."; sort úgy, hogy a firstMatchesSecondWithEquals, firstMatchesSecondWithEqualToOperator, firstMatchesThirdWithEquals értéke true, a firstMatchesThirdWithEqualToOperator értéke pedig false legyen. Magyarázd meg, mi történik a háttérben.

Mielőtt elkezdjük a változtatásokat nézzük meg, hogy mi is történik. Mind látható mindhárom változónkban úgyan az az érték. A Java ezt nagyon okosan kezeli, ez esetben nem hozza létre az adot elemet, csak egyszer és a három változónk minden részén arra mutat, ezzel erőforrást sporolva. Tisztázás képpen a három változó különbözik, de egy adott memóriacímre utal, ahol az érték elmentésre került. Tegyük meg a következő átfirást:

```
String third = new String("...");
```

Erre az átfirásra azért volt szükség, mivel így, annak ellenére, hogy úgyan az van a minden változónkban, a harmadik újra elmentésre kerül egy másik memóriacímben. Na de miért van erre szükség? Mint ahogyan azt már tanultuk az equals() és az == között különbség vannak. Az equals() a két változó tartalmát hasonlítja össze, az == pedig azok memóriacímét, ezért lesz az utolsó állítás hamis. A first és a third bár úgyan azt taralmazzák és az equals() erre igazat is adott, a két érték nem egy memóriacímen szerepel ezért az == hamis értéket ad vissza.

15.4. EPAM: Java GC

Mutasd be nagy vonalakban hogyan működik Java-ban a GC (Garbage Collector). Lehetséges az OutOfMemoryError kezelése, ha igen milyen esetekben?

A GC (Garbage Collector - Szemet gyűjtő) lényege, a memória kezelésben keresendő. Feladata a program által már nem használt memóriát felszabadítani. A Javanak van beépített GC-je, de nézzük meg melyik hét fajta szemétygyűjtővel találkozhatunk.

Serial Garbage Collector - egy egyszerű GC implementáció. Alapjában egyszálon történik a szemet gyűjtés és felszabadítás. Teljesen leállítja az alkalmazást a memória kezelés idejére.

Parallel Garbage Collector - több szálon történik a szemet gyűjtés és felszabadítás, e miatt nagyobb a CPU igénye, mint a Serial Garbage Collectornak, de gyorsabb is. Teljesen leállítja az alkalmazást a memória kezelés idejére.

CMS Garbage Collector - Több szálon történik a szemet gyűjtés. Olyan alkalmazásokban használjuk, ahol fontos a választ idő. A szemetgyűjtés itt négy fázisból áll: initial mark, concurrent marking phase/pre-cleaning, remarking, concurrent sweeping. Az initial mark-ban a program megjelöli az objektumokat, ez idő alatt a program áll. Concurrent marking phase fázisban az alkalmazás futásával egy időben (és ettől

konkurens) bejelöli a tranzitíven elérhető objektumokat. A remarkban ismét leáll a program és az előző fázisban módosult objektumokat, ezzel végegesítve az élő objektumok megjelölését. Az utolsó fázisban a sweepingban pedig törli a memória szemetet.

G1 Garbage Collector - Több magon futó, nagy memóriával rendelkező aplikációkhoz terveztek. A memóriát több részre pontja és egyszere fésüli át őket. A G1 sok tekintetben a CMS-hez hasonlóan működik. Ez is egy konkurens marking fázisban határozza meg, mely objektumok vannak használatban. Mikor a mark fázis kész, a G1 már tudja, hogy melyek a nagyrészt üres régiók. Elsőként ezeket tisztítja ki, ami általában nagy szabad területet eredményez.

Epsilon Garbage Collector - Egy passzív szemétgyűjtő, lefoglalja a memóriát az aplikációnak, de nem gyűjtí össze a nem használatos elemeket. Ez a módszer megengedi, hogy az aplikáció kifusson a memóriából és összeomoljon. Jó arra, hogy letesztelejük, a programunk mennyi memóriát használ, mivel elzártan futtatja azt, a megadott memória értékkel.

Z garbage collector - A programot maximum 10ms-ra állítja le, e miatt ajánlott olyan aplikációkhoz, ahol a nagyobb leállás problémát okozhat. Három lépésben dolgozik.

Shenandoah - Nagyon minimális időre állítja le a programot, mivel nagyrészt egyidőben fut a programmal. A válasz igen is és nem is. Abban az esetben, ha az OOME nem a mi programunk okozta, hanem egy egyébb a háttérben futó aplikáció, ez esetben nem. Abban az esetben, ha a mi programunk okozza, és pontosan tudjuk, hogy hol és micsoda képesek vagyunk rá, hogy elkapjuk és kezeljük. Szerintem ennek a legértelmesebb felhasználása az Epsilon GC használatakor lenne, úgyan is ezzel megelőzhetjük a program összeomlását, ha az kifút a memóriából.

16. fejezet

Helló, Gödel!

16.1. STL map érték szerinti rendezése

Feladatunk a mapot érték szerint rendezni. A map egy olyan asszociatív tároló, mely kulcs-érték párokat fog tartalmazni. Nem tudunk duplikált kulcsokat tárolni benne, ugyanis csak egy érték tartozhat egy kulchhoz. Az STL pedig a Standard Template Library, ez a szabványos C++ nyelv könyvtár. Ez biztosítja a megfelelő hash() függvényeket. A feladat azért érdekes, mivel a map kulcs szerint rendez. Tehát hogy érték szerint tudjuk rendezni szükségünk lesz egy kis változtatásra.

```
#include <iostream>
#include <string>
#include <map>
#include <iomanip>

#include <boost/filesystem.hpp>
#include <boost/filesystem/fstream.hpp>
#include <boost/program_options.hpp>
#include <boost/tokenizer.hpp>

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

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

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

```
) ;

    return ordered;
}

int main(){

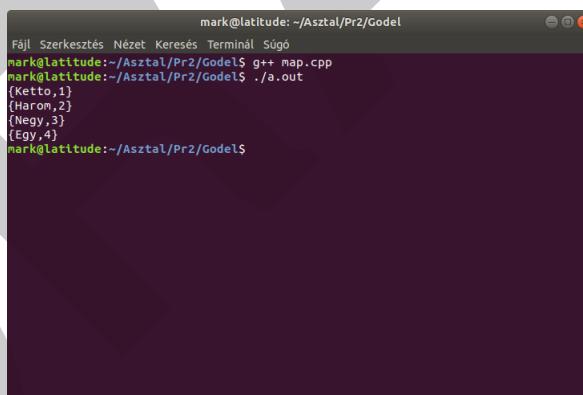
    std::map<std::string, int> map;
    map["egy"] = 3;
    map["kettő"] = 4;
    map["harom"] = 2;
    map["negy"] = 1;

    std::vector<std::pair<std::string, int>> megold = sort_map(map);

    for(auto & i: megold){
        std::cout << i.first << " " << i.second << std::endl;
    }

}
```

Most pedig lássuk a kódot. Az int main() -be létrehozzuk a mapunkat ami egy string és egy int értéket fog kapni (esetünkben például: "egy", 3). A kód alapján láthatjuk, hogy az érték szerinti rendezést úgy oldottuk meg, hogy a map kulcs-érték párokat egy pair vektorba rakjuk. Erre azért van szükség, mivel a vektoron belül már képesek vagyunk érték szerint csökkenőre rendezni a párokat. A végén pedig egy speciális for ciklussal sorba kiíratjuk a vektorunkat.



```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
mark@latitude:~/Asztal/Pr2/Godel$ g++ map.cpp
mark@latitude:~/Asztal/Pr2/Godel$ ./a.out
{Ketto,1}
{Harom,2}
{Negy,3}
{Egy,4}
mark@latitude:~/Asztal/Pr2/Godel$
```

16.2. EPAM: Mátrix szorzás Stream API-val

Implementáld le a mátrix szorzást Java-ban for és while ciklusok használata nélkül.

A mátrix szorzása for és while ismétlő ciklus nélkül funkcionális programozással lehet megoldani. Hogy ezt meg tudjuk oldani elsőnek is létre kell hozni egy abstarcet mátrix osztály, ami a következő képen fog kinézni:

```
public abstract class AbstractMatrix implements Matrix {

    protected final int[][] matrix;
```

```
protected final int rowsLength;
protected final int columnsLength;

public AbstractMatrix(int[][] matrix) {
    this.matrix = matrix;
    this.rowsLength = matrix.length;
    this.columnsLength = matrix[0].length;
}

public AbstractMatrix(int rowsLength, int columnsLength) {
    this.matrix = new int[rowsLength][columnsLength];
    this.rowsLength = rowsLength;
    this.columnsLength = columnsLength;
}

@Override
public void setElement(int x, int y, int value) {
    matrix[x][y] = value;
}

@Override
public Matrix multiply(Matrix input) {
    if (input instanceof AbstractMatrix) {
        return multiply((AbstractMatrix) input);
    }
    throw new IllegalArgumentException("The input matrix should be an instance of AbstractMatrix");
}

abstract protected Matrix multiply(AbstractMatrix input);

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + columnsLength;
    result = prime * result + Arrays.deepHashCode(matrix);
    result = prime * result + rowsLength;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof AbstractMatrix)) {
        return false;
    }
    AbstractMatrix other = (AbstractMatrix) obj;
```

```
if (columnsLength != other.columnsLength) {
    return false;
}
if (!Arrays.deepEquals(matrix, other.matrix)) {
    return false;
}
if (rowsLength != other.rowsLength) {
    return false;
}
return true;
}

@Override
public String toString() {
    return "Matrix [matrix=" + Arrays.toString(matrix) + ", rowsLength=" + ←
        rowsLength + ", columnsLength=" +
        columnsLength + "]";
}
}
```

Mielőtt neki állunk a szorzásnak, tanulmányozuk át a Matrix osztályunkat. Azért hoztuk létre abstractként, mivel így könnyen bővíthetővé válik, ha ez szükséges. Ha elkezdjük olvasni a deklarálást, láthatjuk, hogy három változóval kezdünk. Az első maga a mátrix, a második a sorok száma, a harmadik pedig az oszlopok száma. Két konstruktorunk van, mivel a mátrixot kétféle képen lehet megadni. Az első módja egy már meglévő mátrixátadása, a második módja a sorok és oszlopok számának megadása (két int érték). Maga a szorzás is kétszer szerepel, az első alkalommal egy Overridedal felül definiáljuk. A felül írt függvény ellenőrzi, hogy a szorzás lehetséges-e és ha igen akkor megy tovább az abstact szorzásra, amiről később lesz szó. Továbbá felül lett írva a hashCode, az equals és a toString függvény is, amit nem szeretnék részleteszni, mivel átlátható a működésük, e helyett lásuk az eljárás orientált megvalósítását a szorzásnak.

```
public class LambdaMatrix extends AbstractMatrix {

    public LambdaMatrix(int[][] matrix) {
        super(matrix);
    }

    public LambdaMatrix(int rowsLength, int columnsLength) {
        super(rowsLength, columnsLength);
    }

    @Override
    protected Matrix multiply(AbstractMatrix input) {
        int[][] result = Arrays.stream(this.matrix)
            .map(r -> IntStream.range(0, input.columnsLength)
                .map(i -> IntStream.range(0, input.rowsLength).map(j -> r[j] * ←
                    input.matrix[j][i]).sum())
                .toArray())
            .toArray(int[][]::new);
        return new LambdaMatrix(result);
    }
}
```

```
    }  
}  
}
```

Látható a megoldás csupen egy hosszú utasítás. Látható hogy elsőnek 0-tól az oszlopok számáig veszik az értékeket, majd 0-tól a sorok számáig. Ezt követően kiszámoljuk az értéket, amit egy vektorban mentünk, majd ezeket a vektorokat belementjük a mátrixunkba és visszaadjuk.

16.3. EPAM: LinkedList vs ArrayList

Mutass rá konkrét esetekre amikor a Java-beli LinkedList és ArrayList rosszabb performanciát eredményezhet a másikhoz képest. (Lásd még LinkedList és ArrayList forráskódja). Végezz méréseket is. (mit csinál az ArrayList amikor megtelik)

A két lista között a lényeges különbség és természetesen majd ez okozza a többi eltérést a memória kezelés. A LinkedList nem foglalja le előre az ellemeinek a memóriát. Kezdetben csak az első elemet, a gyökeret ismerjük. Ennek előnye, hogy könnyen bővíthető egészen addig, amíg azt az erőforrásaink megengedik. Azonban megnehezíti a rendezést, a keresést és az elem eltávolítást a lista közepében. Ezzel ellentétben az ArrayList előre lefoglalja a megadot elemszámnak megfelelő memória területet. Ebből adótoan, ezt akkor édemesebb használni, ha előre és biztosan tudjuk az elemek számát. Természetesen a bővítés lehetséges, azonban több időt vesz igénybe, mint a LinkedListnél, mivel első lépésekben lefoglalja az új memória teret, majd átmásolja bele az elemeket.

16.4. EPAM: Refactoring

Adott egy “legacy” kód mely tartalmaz anonymous interface implementációkat, ciklusokat és feltételes kifejezések. Ebben a feladatban ezt a “legacy” kódot szeretnénk átírni lambda kifejezések segítségével (metódus referencia használata előnyt jelent!)

```
public class Refactored {  
  
    public void refactored() {  
        Runnable runnable = createRunnable();  
        runnable.run();  
  
        Calculator calculator = createCalculator();  
        System.out.println("Calculation result: " + calculator.calculate(3));  
    };  
  
    List<Integer> inputNumbers = Arrays.asList(1, null, 3, null, 5);  
    List<Integer> resultNumbers = inputNumbers  
        .stream()  
        .filter(Objects::nonNull) // number -> number != null  
        .map(calculator::calculate) // number -> calculator.  
            calculate(number)  
        .collect(Collectors.toList());  
}
```

```
Consumer<Integer> consumerMethod = createConsumerMethod();
System.out.println("Result numbers: ");
resultNumbers.forEach(consumerMethod);

Formatter formatter = createFormatter();
System.out.println("Formatted numbers: " + formatter.format(←
    resultNumbers));
}

private Runnable createRunnable() {
    return () -> System.out.println("Runnable!");
}

private Formatter createFormatter() {
    return numbers -> numbers.stream()
        .map(String::valueOf)
        .collect(Collectors.joining());
}

private Consumer<Integer> createConsumerMethod() {
    return System.out::println;
}

private Calculator createCalculator() {
    return number -> number * number;
}

}
```

Egy rövid leírás a programról. A Runnable típusú eljárások kifogják íratni a "Runnable!" szöveget. A Calculetor eljárás négyzetre emel. A Consumer kiíratja az int típusú értéket, a Formatter pedig az int típusú értéket Stringé alakítja át. A List<Integer> eljárásánál pedig kiválogatjuk az int tipusú értékeket és csak azokkal dolgozunk.

17. fejezet

Helló, !

17.1. OOCWC Boost ASIO hálózatkezelése

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

Feladatunk a scanf szerepének és használatának bemutatása, a megadott forráskódon keresztül. A scanf() egy függvény az std bemenetről olvas be a függvényben megadott paraméterek szerint. Működése olyan, mint az ismert printf-nek, azonban ez ellentétes irányú, azaz beolvas.

```
{STAT} {WS} {INT}
{
    std::sscanf(yytext, "<stat %d", &m_id);
    m_cmd = 1003;
}
```

A kódban nem a scanf, hanem a sscanf szerepel, ami formázott stringeket olvas be. A függvény működése az alábbi, először a szöveget kell megadni, amit vizsgálni akarunk. Ez a példánkban az yytext, ezt követi a "<stat %d", ez lesz a formátum ami alapján kiolvassa a megadott karaktersorozatot. A m_id argumentum, amibe elhelyezük a konvertált értékeket. Álltalánosan az scanf és a sscanf így néznek ki.

```
//scanf:
int scanf(char *formatum, ...)
//sscanf:
int sscanf(char *string, char *formatum, argu1, argu2, ...)
```

A konzerviós karakterei a scanf-nek: "d" decimális egész; "i" egész szám; "o" oktális szám; "u" előjel nélküli egész; "x" hexadecimális szám, ezeknek az argumentum típusai az int*. A char* típusuak pedig: "c" karakter; "s" karaktersorozat. Ez nem az összes konverziós karakter. További karaktereket találhatunk az interneten. Lehetőségünk van a scanf-en belül szűrésre is. Ezt [] belül tudjuk megtenni, például a [ˆabc] azt jelenti, hogy az inputból minden beolvas kivéve az a,b,c karaktereket. A [a-z0-9] pedig a kisbetűket és számokat nem olvassa be.

17.2. EPAM: XML feldolgozás

Adott egy koordinátákat és államokat tartalmazó XML (kb 210ezer sor), ezt az XML-t feldolgozva szeretnék létrehozni egy SVG fájlt, melyben minden város megjelenik egy pont formájában az adott koordináták alapján (tetszőleges színnel) Plusz feladat: A városokat csoportosíthatjuk államok szerint, és minden állam külön színnel jelenjen meg a térképen, így látszódni fognak a határok is.

Első lépésként a létrehozzuk a városz osztályt az xml alapján. Mivel látható, hogy az xml-ben a városról három információt tárolunk ezért három változonál nincs értelme, hogy többet tároljon. Maga az osztály így nézz ki:

```
@Data  
@Builder  
  
public class City {  
  
    private String coordinateX;  
    private String coordinateY;  
    private String state;  
  
}
```

Most pedig nézzük az osztályt, ami elvégzi a feladatot. Elsőnek is deklarálni kell a kimeneti és bemeneti fájlt, majd pedig végre hajtani a feladatot.

```
public class ParserSolution {  
  
    private static final String INPUT_FILE = "src/main/resources/input.xml" ↵  
        ;  
    private static final String OUTPUT_FILE = "src/main/resources/map.svg";  
  
    public void parse() throws XMLStreamException, TransformerException, ↵  
        FileNotFoundException {  
        List<City> cityList = createCityList();  
        Map<String, String> colorsOfStates = createColorsOfStates(cityList) ↵  
            ;  
        Document document = createDOMSource(cityList, colorsOfStates);  
        TransformerFactory.newInstance().newTransformer().transform(new ↵  
            DOMSource(document), new StreamResult(OUTPUT_FILE));  
    }  
}
```

Haladjunk szépen egyesével a sorokon és vizsgálunk meg minden metodust.

```
private List<City> createCityList() throws XMLStreamException, ↵  
    FileNotFoundException {  
    XMLStreamReader xmlStreamReader = XMLInputFactory.newInstance(). ↵  
        createXMLStreamReader(new FileInputStream(INPUT_FILE));  
    List<City> cityList = new LinkedList<>();  
    CityBuilder builder = City.builder();  
    while (xmlStreamReader.hasNext()) {  
        int actual = xmlStreamReader.next();  
    }
```

```
if (isStartElement(actual, "coordinateX", xmlStreamReader)) {
    builder.coordinateX(xmlStreamReader.getElementText());
} else if (isStartElement(actual, "coordinateY", ←
    xmlStreamReader)) {
    builder.coordinateY(xmlStreamReader.getElementText());
} else if (isStartElement(actual, "state", xmlStreamReader)) {
    builder.state(xmlStreamReader.getElementText());
} else if (isEndElement(actual, "city", xmlStreamReader)) {
    cityList.add(builder.build());
}
}
return cityList;
}
```

A createCityList az első metódus amivel találkozunk, feladata a városok kiolasása az xml fájlból. Észrevehetjük, hogy az előzőhétről tanultan nem ArrayListben mentjük a városokat, mivel sok adatot tárolunk, nem tervezett a törés és a keresés. Meghívásra kerül a City osztály buildere majd kiolasásra kerül az x,y koordináta, az állam és ha elértek a város elem végét, akkor mentésre is kerül.

```
private boolean isStartElement(int actual, String tagName, XMLStreamReader ←
    xmlStreamReader) {
    return actual == XMLStreamReader.START_ELEMENT && tagName.equals(←
        xmlStreamReader.getLocalName());
}

private boolean isEndElement(int actual, String tagName, ←
    XMLStreamReader xmlStreamReader) {
    return actual == XMLStreamReader.END_ELEMENT && tagName.equals(←
        xmlStreamReader.getLocalName());
}
```

Az if else részben a fent látható két metodust használjuk, ahogy a nevük is mondták, ezek csupán megnézik, hogy az adott osztály kezdő vagy végző elem-e.

```
private Map<String, String> createColorsOfStates(List<City> cityList) {
    return cityList.stream()
        .map(City::getState)
        .distinct()
        .collect(Collectors.toMap(hexColor -> hexColor, hexColor -> ←
            randomHexColor()));
}

private String randomHexColor() {
    return String.format("#%06X", new Random().nextInt(0x1000000));
}
```

A createColorOfStates eljárásban átmapaljuk az listánkat, és kiszűrjük belőke az államokat, ezt követően azt leegyszerűsítjük, hogy minden állam csak egyszer szerepeljen. Az utolsó lépésben, meghívjuk a randomHexColor metodust, ami egy véletlenszerű színet ad vissza.

```
private Document createDOMSource(List<City> cityList, Map<String, String> colorsOfStates) {
    Document document = SVGDOMImplementation.getDOMImplementation().createDocument(SVGDOMImplementation.SVG_NAMESPACE_URI, "svg", null);
    Element rootElement = createRootElement(document);
    cityList.forEach(city -> {
        Element element = document.createElementNS(SVGDOMImplementation.SVG_NAMESPACE_URI, "circle");
        element.setAttributeNS(null, "cy", city.getCoordinateX());
        element.setAttributeNS(null, "cx", city.getCoordinateY());
        element.setAttributeNS(null, "fill", colorsOfStates.get(city.getState()));
        element.setAttributeNS(null, "r", "1");
        rootElement.appendChild(element);
    });
    return document;
}

private Element createRootElement(Document document) {
    Element rootElement = document.getDocumentElement();
    rootElement.setAttributeNS(null, "width", "800");
    rootElement.setAttributeNS(null, "height", "600");
    return rootElement;
}
```

Ezt követően létre hozunk egy SVG elemet, ami az eredményünk is lesz. A createRootElement adja meg ennek a méretét. Ezt követően a lista összes városán átmegyünk és a tárolt adatokat a megfelelő attributumba helyezük bele.

17.3. EPAM: ASCII Art

ASCII Art in Java! Implementálj egy Java parancssori programot, ami beolvas egy képet és kirajzolja azt a parancssorba és / vagy egy szöveges fájlba is ASCII karakterekkel.

```
public class Main {

    public static void main(String[] args) throws IOException {
        String imageName = args[0];
        String textFileName = args.length != 2 ? null : args[1];
        OutputStream outputStream = textFileName == null ? System.out : new FileOutputStream(textFileName);
        BufferedImage image = ImageIO.read(new File(imageName));

        new AsciiPrinter(outputStream, image).print();
    }
}
```

A main függvényben elsőnek deklaráljuk a kép nevét, majd a fájl nevét, és ennek a függvényében létrehozzuk a kimeneti fájl nevét. Természetesen ellenörizük, hogy ez a név ne legyen üres. Majd beolvassuk a képet és meghívjuk az AsciiPrinter-t ami ki fogja rajzoni a beolvasott képet

```
public class AsciiPrinter {

    private static final char[] ASCII_PIXELS = { '$', '#', '*', ':', '.', ' ' ←
    };
    private static final byte[] NEW_LINE = "\n".getBytes();

    private OutputStream outputStream;
    private BufferedImage image;

    public AsciiPrinter(OutputStream outputStream, BufferedImage image) {
        this.outputStream = outputStream;
        this.image = image;
    }
}
```

Kezdetben létrehozzuk a karakterkészletünket és egy változót a sortörésre. Majd deklaráljuk a kimeneti fájlt és a képet, amit a konstruktorban kapunk meg.

```
public void print() throws IOException {
    for (int i = 0; i < image.getHeight(); i++) {
        for (int j = 0; j < image.getWidth(); j++) {
            outputStream.write(getAsciiChar(image.getRGB(j, i)));
        }
        outputStream.write(NEW_LINE);
    }
}

public static char getAsciiChar(int pixel) {
    return getAsciiCharFromGrayScale(getGreyScale(pixel));
}

public static int getGreyScale(int argb) {
    int red = (argb >> 16) & 0xff;
    int green = (argb >> 8) & 0xff;
    int blue = (argb) & 0xff;
    return (red + green + blue) / 3;
}

public static char getAsciiCharFromGrayScale(int greyScale) {
    return ASCII_PIXELS[greyScale / 51];
}
```

A print metodus bejárja a képet pixelről pixelre, mintha egy mátrix lenne és minden pixelt helyére kiírja a megfelelő karaktert, a fent létrehozott készletből. Azt, hogy melyik karaktert szükséges beszűrni a getAsciiChar metódus mondja meg, ami segítségül hívja a getAsciiCharFromGrayScale-t , aminek paraméterül adja a pixel színének az elszürkített értékét. Ezt az értéket a getGreyScale fogja visszadni. A szürkítéshez

az eltolást használjuk, amit nem sűrűn szoktunk a Javaban. A kiírt karakter eldöntése a kapott szürkepont érték 51-vel való osztásával történik meg.

17.4. EPAM: Titkos üzenet, száll a gépben!

Implementájl egy olyan parancssori alkalmazást, amely a billentyűzetről olvas soronként ASCII karakterből álló sorokat, és a beolvasott szöveget Caesar kódolással egy txt fájlba írja soronként.

Mielőtt belekezdünk a feladatba, nézzük meg, mit jelent a Caesar kódolás. A Ceasar kódolás az egyik legegyszerűbb titkosítási módszer, lényege, hogy az ABC-t egy adott értékkel eltolja. Az az ha 3-val toljuk el az ÁBC-t, akkor az A betű helyet egy C betűt írunk. A nevét onnan kapta, hogy Julius Ceasar Róma császára így kommunikált tábornokaival. De kezdjük is el a programot a main osztállyal.

```
public class Main {  
  
    public static void main(String[] args) throws IOException {  
        String fileName = args[0];  
        int offset = Integer.valueOf(args[1]);  
        try (StreamEncoder handler = new ConsoleInputToFileCeasarEncoder(←  
            fileName, offset)) {  
            handler.handleInputs();  
        }  
    }  
}
```

Mint látható a main függvényünk nem túl bonyolult. Megadjuk a fájl nevét amibe dolgozni fogunk, illetve az offset-t ami az eltollás érdéke lesz. Ezt követően átadjuk az adatokat a handlernek ami egy StreamEncoder, amiben végre hajtjuk a titkosítást, de elsőnek nézzük meg magát a titkosítást.

```
public class ConsoleInputToFileCeasarEncoder extends ←  
    StreamToFileCeasarEncoder {  
  
    public ConsoleInputToFileCeasarEncoder(String fileName, int offset)  
        throws FileNotFoundException {  
        super(System.in, fileName, offset);  
    }  
}
```

A ConsoleInputToFileCeaserEncoderrel olvasunk be az interfészről, majd tovább adjuk a kimeneti fájl és az offset méretét

```
public class CeasarCoder implements Encoder, Decoder {  
  
    private final int offset;  
  
    public CeasarCoder(int offset) {  
        if (offset < 1 || offset > 127) {
```

```
        throw new IllegalArgumentException("Offset must be between 1 and 127" ↵
    );
}
this.offset = offset;
}

public CeasarCoder() {
    this.offset = 1;
}
```

A CeasarCodert két részben fogjuk átnézni. Elsőnek is megemlíteném, hogy az Encoder és Decoder osztály egyaránt implementálja de mivel ezzen kívül más nem tesznek, nem tartom lényegesnek beszúrni a kójukat. Ahogy láthatjuk két konstruktőr van, ezt egy fajta hibakezelés miatt csináltuk így. Amennyiben offset érték nélkül kerül meghívásra az osztály, ez esetben egy lesz annak értéke, ha pedig kap értéket, akkor leelenörzi, hogy 1 és 127 közötti-e az érték.

```
@Override
public String decode(String text) {
    return buildString(text, character -> (char) ((character - offset) % ↵
        128));
}

@Override
public String encode(String text) {
    return buildString(text, character -> (char) ((character + offset) % ↵
        128));
}

private String buildString(String text, Function<Character, Character> ↵
    function) {
    StringBuilder result = new StringBuilder();
    for (char character : text.toCharArray()) {
        if (character != ' ') {
            result.append(function.apply(character));
        } else {
            result.append(character);
        }
    }
    return result.toString();
}
```

A fenti kódrészben történik maga a kodolás és visszafejtés. Mint látható az encode és Decode függvények úgyan úgy működnek, csupán abban különböznek, hogy az anonim eljárásban a kódalsnál hozzáadjuk, a visszafejtésben, pedig kivonjuk az eltolás értékét. A szöveg felépítését StringBuilder segítségével oldjuk meg, ami azt jelenti, hogy magát a szöveget nem hozzuk létre, ameddig meg nem történt a teljes átalakítás. Látható, hogy a for ciklusban egyenként végig megyünk a karakter sor elemein, és ha azok nem szöközzök, átalakítjuk, majd csak appendeljük azt a többihez. A végént a result.toString() lesz az, ami létrehozza a kész karaktersort.

```
public class StreamEncoder implements AutoCloseable {
```

```
private static final byte[] NEW_LINE = "\n".getBytes();

private Scanner inputScanner;
private OutputStream outputStream;
private Encoder encoder;

public StreamEncoder(InputStream inputStream, OutputStream outputStream, ←
    Encoder encoder) {
    this.inputScanner = new Scanner(inputStream);
    this.outputStream = outputStream;
    this.encoder = encoder;
}

public void handleInputs() throws IOException {
    String line;
    do {
        line = inputScanner.nextLine();
        String encodedLine = encoder.encode(line);
        outputStream.write(encodedLine.getBytes());
        outputStream.write(NEW_LINE);
    } while (!"exit".equals(line));
}

@Override
public void close() throws IOException {
    inputScanner.close();
    outputStream.close();
}

}
```

Most pedig nézzük meg a StreamEncoder-t. Mint látható lényegesen a handleInputs metódust fogjuk használni. Létrehozunk benne egy üres sort és do while ciklussal addig hajtjuk végre az utasításokat, amíg a sor egyenlő nem lesz az "exit" szöveggel. A Scannar nextLine metódusa egy teljes sort olvas be nekünk, majd ezt titkosítjuk, kiíratjuk és egy sortörést is kiíratunk.

18. fejezet

Helló, Lauda!

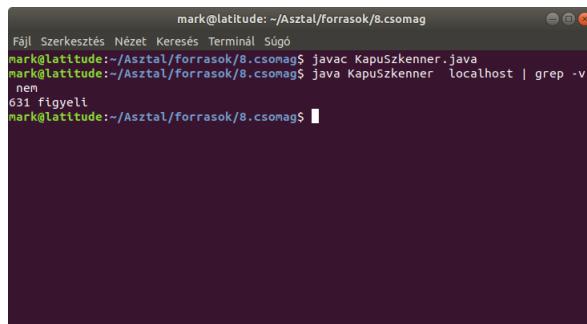
18.1. Port scan

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

A Port Scan azt mutatja meg, hogy egy hálózatnak melyik portjai nyitottak a kommunikációra. Mi most meg fogjuk nézni a kódot, amit tanár úr adott meg:

```
public static void main(String[] args) {
    for(int i=0; i<1024; ++i)
        try {
            java.net.Socket socket = new java.net.Socket(args[0], i);
            System.out.println(i + " figyeli");
            socket.close();
        } catch (Exception e) {
            System.out.println(i + " nem figyeli");
        }
}
```

A programunk azt nézi át, hogy melyik portunkat figyeli épp a számítógép. Láthatjuk hogy a program egy for ciklusbol, azon belül egy try-catch-bol áll. A program lelke a try blokkon belül van, mégpedig " a "java.net.Socket socket = new java.net.Socket(args[0], i);" A vizsgálathoz az argumentumként megkapott IP-címmel próbálunk meg egy TCP kapcsolatot létrehozni. Ha sikerült a kapcsolat létrehozása akkor kiíratjuk, hogy "figyeli". Ezek után a socketet, amit nyitottunk be is zárjuk, hogy feloldjuk a lefoglalt portot. Ha viszont a kapcsolat nem sikerült a kapcsolat, akkor az Expection lép életbe, ami kiírja, hogy nem figyeli a portot a gép.



```
mark@latitude:~/Asztal/forrasok/8.csomag$ javac KapuSzkenner.java
mark@latitude:~/Asztal/forrasok/8.csomag$ java KapuSzkenner localhost | grep -v
nem
631 figyeli
mark@latitude:~/Asztal/forrasok/8.csomag$
```

18.2. EPAM: DI

Implementálj egy alap DI (Dependency Injection) keretrendszerit Java-ban annotációk és reflexiós használatával megvalósítva az IoC-t (Inversion Of Control).

Elsőnek is beszéljünk az Inversion of Controlról. Tömörean a lényege, hogy megfordíjuk a függőség kezelését. Az az példányosítást és metódus hívást egy más eszközre, például keret rendszerekre bízzuk. Az osztály kér példányokat, de nem mondja meg hogyan. A Dependency Injection pedig ezt megadja. Azonban nem minden esetben hozz létre új pédányt, hanem ha talál el egy olyan objektumot, ami szükséges azt adjva vissza. Ekkor a konstruktörben kapja meg paraméterről. Lássuk most a feladat megvalósítását.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Bean {
```

```
}
```



```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface Qualifier {
```



```
    public String name();
```

A fentlátható két kód részletben létrehoztunk két annotációt. Ezeket az onnótációkat általában arra használjuk, hogy jelezzük JVM-nek a futatásban való eltérést. Definiálás során használtuk még annotációkat. A Target azt mondja meg, hogy milyen tipusúra lehet ráenni.

```
package com.epam.training.di;
```



```
public interface Configuration {
```



```
}
```



```
public interface DiContext {
```



```
    public void addBean(String beanName, Object bean);
```

```
public <T> Optional<T> getBean(String beanName, Class<T> clazz);  
  
public <T> Optional<T> getBean(Class<T> clazz);  
  
}
```

A két annotáció mellett, létrehozunk két interfacet is. Az első a Configuration láthatóan üres, ezt csak nevezékre használjuk majd. A DiContext egy kontextus keszelő lesz. Ebben absztrak metodokat fogunk használni. Beaneket fogunk tudni beleenni illetve kinyerni. A methodunkat két féle képen lehet majd használni, ezért kétféle képpen definiáljuk.

```
public class DiContextImpl implements DiContext {  
  
    private Map<String, Object> context = new HashMap<>();  
    private Map<Class< ?>, List<Object>> contextMappedByType = new HashMap< ?>();  
  
    public void addBean(String beanName, Object bean) {  
        if (context.containsKey(beanName)) {  
            throw new IllegalArgumentException("Bean has been already created: " + beanName);  
        }  
        context.put(beanName, bean);  
        if (contextMappedByType.containsKey(bean.getClass())) {  
            contextMappedByType.get(bean.getClass()).add(bean);  
        } else {  
            List<Object> objectList = new LinkedList<>();  
            objectList.add(bean);  
            contextMappedByType.put(bean.getClass(), objectList);  
        }  
    }  
  
    @SuppressWarnings("unchecked")  
    public <T> Optional<T> getBean(String beanName, Class<T> clazz) {  
        Optional<T> ret = null;  
        if (context.containsKey(beanName)) {  
            ret = Optional.of((T) context.get(beanName));  
        } else {  
            ret = Optional.empty();  
        }  
        return ret;  
    }  
  
    @SuppressWarnings("unchecked")  
    @Override  
    public <T> Optional<T> getBean(Class<T> clazz) {  
        Optional<T> ret = null;  
        List<Object> objectList = contextMappedByType.get(clazz);  
        if (objectList == null) {  
            ret = Optional.empty();  
        } else {  
            ret = objectList.stream().filter(clazz::isInstance).map(clazz::cast).findAny();  
        }  
        return ret;  
    }  
}
```

```
    } else if (objestList.size() != 1) {
        throw new IllegalArgumentException("There are multiple bean available ←
            for type: " + clazz);
    } else {
        ret = Optional.of((T) objestList.get(0));
    }
    return ret;
}

}
```

A fenti kódrészletben a DiContext implmentációját láthatjuk. Első sorban létrehozunk egy kontext mappat, ahol a kulcs egy név a Bean neve, és az Object maga az elem. A contextMappedByType-ban pedig, egy osztáyt, amiből van egy listányi példánk. Az addBean, paraméterül kap egy Bean nevet és példányt, és megnézi, hogy létezik-e, ha nem akkor beleteszi a mappba, különben nem. Ha létrehozza, megnézi a következő mappben, és ha abban már abban benne van ez az osztály, akkor az ahoz tartozó listához adja hozzá. A getBean-ek megnézik, hogy a paraméter szerepel-e már a mapban, ha igen, akkor becsomagoljuk egy optional típusba és visszaadjuk. A második metodus annyiban különbözik, hogy ha az adott osztályhoz több elem tartozik, akkor hiba üzenet küld, mivel ekkor az Optional nem működik megfelelően.

```
public class DiContextBuilder {

    public DiContext build(Configuration contextConfiguration) {
        DiContext context = new DiContextImpl();

        List<BeanDefinition> beanDefinitions = new LinkedList<BeanDefinition>() ←
            ;

        for (Method method : contextConfiguration.getClass().getMethods()) {
            if (isAnnotatedWithBean(method)) {
                List<BeanDefinitionParameter> beanDependencies = new LinkedList<←
                    DiContextBuilder.BeanDefinitionParameter>();
                for (Parameter parameter : method.getParameters()) {
                    String name = null;
                    if (parameter.getAnnotation(Qualifier.class) != null) {
                        name = parameter.getAnnotation(Qualifier.class).name();
                    }
                    beanDependencies.add(new BeanDefinitionParameter<>(name, ←
                        parameter.getType()));
                }
                beanDefinitions
                    .add(new BeanDefinition(method.getName(), method.getReturnType ←
                        (), method, beanDependencies));
            }
        }

        int beanDefinitionsSize = beanDefinitions.size();
        context.addBean("context", context);

        while (beanDefinitions.size() > 0) {
```

```
List<BeanDefinition> resolvedBeanDefinitions = new LinkedList<>();
for (BeanDefinition beanDefinition : beanDefinitions) {
    if (isBeanDefinitionResolvable(beanDefinition, context)) {
        context.addBean(beanDefinition.name, createBean(beanDefinition, ←
            context, contextConfiguration));
        resolvedBeanDefinitions.add(beanDefinition);
    }
}
beanDefinitions.removeAll(resolvedBeanDefinitions);

if (beanDefinitionsSize == beanDefinitions.size()) {
    throw new IllegalArgumentException("Circular dependency!");
} else {
    beanDefinitionsSize = beanDefinitions.size();
}
}

return context;
}

private Object createBean(BeanDefinition beanDefinition, DiContext ←
    context, Object contextConfiguration) {
List<Object> dependencies = new ArrayList<>(beanDefinition.dependencies ←
    .size());

for (BeanDefinitionParameter parameter : beanDefinition.dependencies) {
    Optional<Object> dependency;
    if (parameter.name != null) {
        dependency = context.getBean(parameter.name, parameter.type);
    } else {
        dependency = context.getBean(parameter.type);
    }
    dependencies.add(dependency.get());
}

try {
    return beanDefinition.builderMethod.invoke(contextConfiguration, ←
        dependencies.toArray());
} catch (IllegalAccessException | IllegalArgumentException | ←
    InvocationTargetException e) {
    throw new IllegalArgumentException(e);
}
}

private boolean isBeanDefinitionResolvable(BeanDefinition beanDefinition, ←
    DiContext context) {
for (BeanDefinitionParameter parameter : beanDefinition.dependencies) {
    Optional<Object> dependency;
    if (parameter.name != null) {
        dependency = context.getBean(parameter.name, parameter.type);
    }
}
```

```
        } else {
            dependency = context.getBean(parameter.type);
        }
        if (dependency.isEmpty()) {
            return false;
        }
    }
    return true;
}

private boolean isAnnotatedWithBean(Method method) {
    for (Annotation annotation : method.getDeclaredAnnotations()) {
        if (annotation.annotationType().equals(Bean.class)) {
            return true;
        }
    }
    return false;
}

private class BeanDefinition {
    String name;
    Class type;
    Method builderMethod;
    List<BeanDefinitionParameter> dependencies;

    public BeanDefinition(String name, Class type, Method builderMethod,
        List<BeanDefinitionParameter> dependencies) {
        super();
        this.name = name;
        this.type = type;
        this.builderMethod = builderMethod;
        this.dependencies = dependencies;
    }
}

private class BeanDefinitionParameter<T> {
    String name;
    Class<T> type;

    public BeanDefinitionParameter(String name, Class<T> type) {
        super();
        this.name = name;
        this.type = type;
    }
}
}
```

A DiContextBuilderben pedig létrehozza a kontextust, amiben dolgozni fogunk tudni. A bulidernek csak egy publikus metódusa van a build, ami paraméterül fog kapni egy configot, és ezzel fogunk dolgozni.

Ezt követően létrehozunk egy példányt a kontextből és egy üres listát, ami a Bean-eket fogja tárolni. A BeanDefinition és A BeanDefinitionParameter arra szolgál, hogy a Beanről információkat szerezzenk. Meg tudjuk a példány nevét, osztályát, builderMethodját és függőségeit. Az első forban végig megyünk az osztály összes metódusán. Majd a isAnnotatedWithBean segítségével, ami úgyan úgy végig megy az összes metóduson, megnézi, hogy van-e egyforma metódusa az osztálynak és a Beannek. Ezt követően az összes megegyező metódusnak létrehozunk egy listát és beletesszük.

18.3. EPAM: JSON szerelmezés

Implementálj egy JSON szerelmezési könyvtárat, mely képes kezelni sztringeket, számokat, listákat és beágyazott objektumokat. A megoldás meg kell feleljen az összes adott unit tesztnak. Plusz feladat: 1. a könyvtár tudjon deszerelmezni

A JSON fájlról és anak használatáról nem szeretnék írni, úgyan is tanultuk webtechnologiák órán. Annyit említenék meg róla, hogy egyes adatokat hogy tárol. A szövegeket idézőjelekbe teszi, a listát pedig szögletes zárójelekbe, az elemeket vesszővel elválasztva. Térjünk is át a kódra, és nézzük meg a fő osztályt.

```
public class MiniObjectMapper {  
  
    private static final Serializer< ? > nullSerializer = new NullSerializer ←  
        ();  
    private static final List<Serializer< ?>> SERIALIZER_LIST = List.of(  
        new StringSerializer(),  
        new NumberSerializer(),  
        new BooleanSerializer(),  
        new ListSerializer(),  
        new ObjectSerializer()  
    );  
  
    private final MainSerializer mainSerializer;  
  
    public MiniObjectMapper() {  
        this.mainSerializer = new MainSerializer(SERIALIZER_LIST, ←  
            nullSerializer);  
    }  
  
    public MiniObjectMapper(List<Serializer< ?>> customSerializerList) {  
        List<Serializer< ?>> finalSerializerList = new ArrayList<>(<←  
            customSerializerList);  
        finalSerializerList.addAll(SERIALIZER_LIST);  
        this.mainSerializer = new MainSerializer(finalSerializerList, ←  
            nullSerializer);  
    }  
  
    public String toJson(Object obj) {  
        return mainSerializer.serialize(obj, mainSerializer);  
    }  
}
```

A főosztályban indulás kép létrehozunk egy új Serializer interfészket tartalmazó listát, amiben minden érték típusnak egy saját Serializere lesz. Ezt a feladat végén típusonként végig nézzük. Ezt követően MiniObjectMapper-t, ami az össze tett objectet mappolás segítségével felbontja egyszerűbb részlekre. A legvégén pedig a toJson kírja az így elkészült Json fájlt, a return résznél megfigyelgető, hogy a meghívja saját magát rekúrzívan.

Nézzük magát a Serializer interfacet. Mint látható ezt generikusan írtuk meg, így ezt az interfacet több típusra is használhatjuk, egyszer egyszerűsítve, és meggyósítva feladtunk. Itt vissza is tudjuk adni a paraméterként kapott objektum osztályát.

```
public interface Serializer<T> {
    String serialize(T obj, Serializer<Object> mainSerializer);
    Class<T> getSourceClass();
}

public class MainSerializer implements Serializer<Object> {

    private final List<Serializer< ?>> serializerList;
    private final Serializer< ?> nullSerializer;

    public MainSerializer(List<Serializer< ?>> serializerList, Serializer< ?> nullSerializer) {
        this.serializerList = serializerList;
        this.nullSerializer = nullSerializer;
    }

    @Override
    public String serialize(Object obj, Serializer<Object> mainSerializer) {
        Serializer<Object> selectedSerializer = (Serializer<Object>) selectSerializer(obj);
        return selectedSerializer.serialize(obj, this);
    }

    @Override
    public Class<Object> getSourceClass() {
        return Object.class;
    }

    private Serializer< ? > selectSerializer(Object obj) {
        return serializerList.stream()
            .filter(serializer -> serializer.getSourceClass().isInstance(obj))
            .findFirst()
            .orElse(nullSerializer);
    }
}
```

A MainSerializer-ben csupán egy elemmel dolgozunk. A selectSerializer fogja kiválasztani a megfelelő Serializert a serialize-ben található getSourceClass segítségével, aamit Overrideltünk. Ha listán végig érve

nem találja meg a megfelelő osztályt, akkor a nullSerializer-t adja vissza. Most pedig akkor lássuk a típusok Serializerét.

```
public class BooleanSerializer implements Serializer<Boolean> {

    @Override
    public String serialize(Boolean obj, Serializer<Object> mainSerializer) ←
    {
        return obj ? "true" : "false";
    }

    @Override
    public Class<Boolean> getSourceClass() {
        return Boolean.class;
    }

}
```

A BooleanSerializer a "true" vagy "false" karaktersort adja vissza, a boolean igazságértéke szerint.

```
public class NullSerializer implements Serializer<Object> {

    @Override
    public String serialize(Object obj, Serializer<Object> mainSerializer) ←
    {
        return "null";
    }

    @Override
    public Class<Object> getSourceClass() {
        return Object.class;
    }

}
```

A nullSerializer a "null" karaktersort adja vissza.

```
public class NumberSerializer implements Serializer<Number> {

    @Override
    public String serialize(Number obj, Serializer<Object> mainSerializer) ←
    {
        return obj.toString();
    }

    @Override
    public Class<Number> getSourceClass() {
        return Number.class;
    }

}
```

A NumberSerializer a paraméterként szám értéket adja vissza string formában.

```
public class StringSerializer implements Serializer<String> {

    @Override
    public String serialize(String obj, Serializer<Object> mainSerializer) ←
    {
        return "\"" + obj + "\"";
    }

    @Override
    public Class<String> getSourceClass() {
        return String.class;
    }

}
```

A StringSerializer a paraméter ként adott stringer adja vissza idézőjelekben.

```
public class ListSerializer implements Serializer<List> {

    @Override
    public String serialize(List obj, Serializer<Object> mainSerializer) {
        return (String) obj.stream()
            .map(o -> mainSerializer.serialize(o, mainSerializer))
            .collect(Collectors.joining(", ", "[", "]"));
    }

    @Override
    public Class<List> getSourceClass() {
        return List.class;
    }

}
```

A ListSerializer egy streamet készít a lista elemeiből, majd ezt egy collectionbe helyezi egy kapcsos zárójel párosba, az elemeket vesszővel elválasztva. A végeredményt kasztoljuk és sztringként adjuk vissza.

```
public class ObjectSerializer implements Serializer<Object> {

    @Override
    public String serialize(Object obj, Serializer<Object> mainSerializer) {
        return Arrays.stream(obj.getClass().getDeclaredFields())
            .map(field -> {
                try {
                    String name = field.getName();

                    if (!field.canAccess(obj)) {
                        field.setAccessible(true);
                    }
                }
            })
            .collect(Collectors.joining(", "));
    }
}
```

```
        String value = mainSerializer.serialize(field.get( obj), mainSerializer);

        return String.format("\\"%s\" : %s", name, value);
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
    return "";
})
.collect(Collectors.joining(", ", " {", " }"));
}

@Override
public Class<Object> getSourceClass() {
    return Object.class;
}

}
```

A az ObjectSerializer úgyan úgy működik, mint a ListSerializer, annyi az eltérés, hogy az objectben több típus is szerepelhet ezét minden elemére meg kell hívni a mainSerializer-t.

18.4. EPAM: Kivételkezelés

Adott az alábbi kódrészlet. Mi történik, ha az input változó 1F, “string” vagy pedig null? Meghívódik e minden esetben a finally ág? Válaszd indokold!

```
public void test(Object input) {
    try {
        System.out.println("Try!");
        if (input instanceof Float) {
            throw new ChildException();
        } else if (input instanceof String) {
            throw new ParentException();
        } else {
            throw new RuntimeException();
        }
    } catch (ChildException e) {
        System.out.println("Child Exception is caught!");
        if (e instanceof ParentException) {
            throw new ParentException();
        }
    } catch (ParentException e) {
        System.out.println("Parent Exception is caught!");
        System.exit(1);
    } catch (Exception e) {
        System.out.println("Exception is caught!");
    } finally {
```

```
        System.out.println("Finally!\n");
    }
}
```

A try finally ágáról azt kell tudni, hogy minden esetben meghívódik ha a program nem áll le. Az az a fenti kód részletnek a kimenetén kétszer fog megjelenni, mivel a ParentException-ben egy 1-es kóddal kiléünk mielőtt a Finally ág meghívódásra kerülne.

DRAFT

19. fejezet

Helló, Berners-Lee!

19.1. Java és C++ OOP programozási nyelvek:

Az első felvonásban a C++ program nyelvel foglalkoztunk. Ebben a fejezetben betekintést nyerhetünk a Java program nyelvbe, amely az objektum orientált programozási nyelvek családjába tartozik.

Bevezetőként ismerjük meg a jelölések és rövidítések jelentését:

OOP = Objektum Orientált Programozás

OO = Objektum Orientált

Példány = Egy osztály objektum, melyből egyetlen és egyedi létezik. Viszont egy osztálynak több objekuma is létezhet.

Mi az az objektum oriáltság? Erre a kérdésre egy egyszerű válasz az osztályozás használata. Tehát az osztályozás azt jelenti, hogy egy programban több kategorizálást hozhatunk létre és ezekkel jól elkülönítve tudunk dolgozni. Egy egyszerűbb példán keresztül demonstrálva az osztályozás és az objektum orientáltság azt jelenti, hogy választunk például egy bizonyos tárgyat, amelynek több tulajdonsága megegyezik egy másik hasonló tárgyéval, jelent esetben legyen egy autó. Ha van egy autónk akkor biztosan vannak kerekei, ablakai, súlya, színe, és még sorolhatnánk, tehát bizonyos tulajdonságokkal rendelkezik. De ha nekünk van autónk akkor másnak miért ne lenne, és ha több autó is létezik akkor kell legyen egy gyár ahol elkészülnek az autók. Ha az autó gyárra gondolunk képzeliük el, hogy a gyárban több autót gyártanak naponta, tehát minden egyes autó egy példány, hiszen minden autó kap gyártáskor egy alvázszámot, ezért még ha nagyon is hasonlíthatók két autó sem ugyan arról az autóról beszélünk. A java-ban az OO-ságot képzeliük el gyáraknak ahol minden gyárban, azaz osztályban, más-más termék gyártása történik. Az osztályokat, mint a gyárat, azért hozták létre, hogy megkönyítsenek valamit, hiszen egyszerűbb általánosságban foglalkozni egy autó megtervezésével mintha minden egyes autó legyártása után el kellene gondolkozni azon, hogy hogyan is készült az előbb elkészült autó? Hanem egy osztályt, azaz egy tervet hozva létre, így minden autót márktól, modeltől függően gyártanak egymás után.

Ezen egyszerű példán keresztül a java nyelv is felfogható nagyon egyszerűen, hiszen egy OO nyelv. A C++ viszont többelvű programozási nyelvnek mondható, mert abban nem feltétlenül kell OO szemléletben programozni. A java nyelv a C nyelvhez nagyon hasonlít szintakszisban hiszen a minden két nyelv egymáshoz időben közel kerül kiadásra. A Java még a régebbi C nyelvhez hasonlít kulcsszavaiban, de a C++ mára már sok fejlesztésen esett keresztül, ezért kulcsszavakban már rövidebb szintaktikát használ. Mindkét nyelvről

elmondható, hogy magas szintű programozási nyelv, azaz felhasználó közeli nyelv, ellentétben az alacsony szintűekkel amelyek gépközeli, a gép által könnyen megértett nyelvet képviselik.

A Java nyelv hírnevét eredetileg a WWW oldalakon betölött szerepének köszönheti, ahol appleteket hoztak létre melyeket HTML oldalak használtak. Egyszerű példa erre egy bejelentkező ablak felület melynek a működését java nyelven írtak.

Mivel a Java nyelv nagyban hasonlít a C++ nyelvhez ezért érdemes szinkronban beszélni róluk.

Változó:

Olyan tárterület amely értékeket tárol, mely rendelkezik névvel, értékkel, és azonosítóval.

A változóknak több típusa lehet:

- boolean : igaz vagy hamis logiaki érték.
- char : 16 bites Unicode karakter.
- int : 32 bites előjeles egész szám.
- short : 16 bites előjeles egész szám.
- long : 64 bites előjeles egész szám.
- float : 32 bites lebegőpontos racionális szám.
- double : 64 bites lebegőpontos racionális szám.

A operátorral a bal oldalán lévő változóhoz a jobb oldalán lévő értéket rendeljük.

Kommenteket is lehet írni amiben magyarázni tudjuk a kódunk mellett működését. Egy soros kommenthez "://" jelet, több soros kommenthez "/* commnet */"-et használunk.

Literálok:

Bármilyen konstans értéket, amit egy változónak értékül adhatunk literálnak nevezünk. Például:

- "Peti" - Sztring literál
- 45 - Egész szám (int) literál
- 21.043 - Tört szám (double) literál
- 53.023f - Tört szám (float) literál

Operátorok:

A java nyelvben is léteznek értékkadó operátorok mint a C++-ban, ezek az értékkadás (=), értékkadás összeadással (+=), értékkadás kivonással (-=), értékkadás szorzással (*=), értékkadás egész osztással (/=), értékkadás maradékos osztással (%=).

Az aritmetikai operátorok is megtalálhatók, ezek az összeadás (+), kivonás (-), osztás (/), szorzás (*) és maradékos osztás (%). Érdekességképpen a java nyelvben van előre definiált függvény operátor is ami az előbb említett operátoroknak a tulajdonságával bír, mint például a minus() vagy a plus().

A relációs operátorok a következők, egyenlő (==), nem egyenlő (!=), kisebb (<), nagyobb (>), kisebb egyenlő (<=) és nagyobb egyenlő (>=).

A logikai operátorok a következők, vagy (||), és (&&), negáció (!);

A tömbök vagy listák:

Javaban is mint a C++-ban léteznek tömbök amelyket itt inkább listaként definiálnak. A tömbök lehetnek szöveg és szám tipusúak, a lényegük, hogy több értéket képesek eltárolni. Az enum kulcsszóval speciális listát hozhatunk létre, melyre hivatkozhatunk később a kódunkban, amely egy nem bővíthető listaként funkcionál.

Structúra:

A Struct kulcsszóval hozhatjuk létre és megadhatunk több típusú változót egy structurában amelyet kézőbb mint egy definiált új tipust használhatunk összesített típusként. Egy struktura elemeire külön-külön hivatkozhatunk a strukturaNeve.elem kombinációval.

A Math osztályban előre definiált matematikai műveletek vannak, mint például a sum() azaz az összegzés, az sqrt() azaz a négyzetgyökvonás, az avg() azaz az átlag számítás vagy a rand() a véletlenszerű szám létrehozás.

A vektor már jól ismert a C++-ból. Egy vektor létrehozása a javaban a következőképpen néz ki:

```
Vector <int> num = new Vector <int> ();
Vector <String> str = new Vector <String> ();
```

Ha értékeket akarunk adni egy vektorba azt az add() függvényel tehetjük meg.

```
num.add(1);
num.add(2);
num.add(3);

str.add("Név: Pisti");
str.add("Kor: 21");
str.add("Lakhely: Debrecen");
```

A stringes vagy szöveges vektor helyet érdemees egy structurát létrehozni, ha a fent látott tulajdonságokat akarjuk eltárolni és így több ilyen személynek a tulajdonságait is tárolhatjuk, ha tömbként hozzuk létre.

Struct

Javaban a legkisebb önálló elem az osztály, míg C++-ban nem feltétenül kell osztályokat használni. Tehát egy osztályt a class kulcsszóval hozunk létre. Az osztály fogalmát feljebb megérthettük, röviden az azonos típusú dolgok modeljét írja le. Az osztályban létrehozhatunk változókat és függvényeket, metódusokat. Változókat a fent ismertetett típusnév kulcsszót a neve elő írva hozhatunk létre. Egy függvényt pedig úgyan úgy mint egy változót, tehát a típusát megadva, de a név után egy "(" zárójeleket írva majd ezt követve a "{}" zárójelek közé írva megadjuk a működését.

Osztályok:

A java egy objektum orientált nyelv. A javaban minden példányokhoz, vagy azaz objektumokhoz és osztályokhoz van társítva. Az osztály egy objektumkészítő, amellyel létrehozhatunk új objektumokat. Olyan dolgokat, tárgyakat, élőlényeket érdemes osztályban definálni amelyek közös tulajdonságokkal rendelkeznek, mint például az autó, aminek van színe, súlya, kereke stb..

Az osztályok számunkra megkönnyítik a programozást mivel egy átláthatóbb osztályozott környezeten keresztül biztosítva bármely más programozó egy kis tanulmányozással megérte a programíró célját a programmal kapcsolatban. Egy osztályban érdemes olyan függvényeket létrehozni amelyeket sűrűn használ a programunk, ezzel egy helyre csoportosítva őket, programozói nézetben könnyebb tájékozódni a program kódján.

Egy osztály létrehozása a class kulcsszóval majd egy név megadásával történik, példa egy osztályra benne egy függvénnnyel:

```
//java
public class Pelda{

    String szoveg;
    int szam;

    void peldametodus(int parameter) {
        szam+= parameter
    }
}

//C++
Class Pelda{

public:

    void Hello(){
        cout<<"Hello";
    }
}
```

Az osztály szintjei, vagy a jogosultságok:

Ezeket a C++-ban teljes kód részek amelyeket elég egyszer kiírni aztán, minden ami utána van az ahoz a részhez tarozik, egészen addig amíg egy másik rész kulcsszava nem következik. C++-ban minden osztályban egy-egy ilyen rész hozható létre, míg Javaban a függvény vagy változó elé írva lesz definiálva, másszóval Javaban egyfajta tipusként van definiálva.

Public rész, amely egy kulcsszó is egyben, tehát Javaban ezt elé írva egy változónak vagy függvénynek elérjük hogy más osztályban is látható legyen, ne csak abban ahol létrehoztuk.

Private rész, amely lehetővé teszi, hogy csak az az osztály lássa amelyben létrejött.

Protected rész, amely lehetővé teszi a megosztását más osztályokban, vagyis a leszármazott osztályok, vagy örökölt osztályok, C++-ban általában azokat a függvényeket amiket itt definiálunk, friend, azaz barát függvényeknek nevezünk.

A Java támogat úgynevezett félnyilvános tagokat is, ami azt jelenti hogy nincs jelezve milyen jogosultsága van, viszont ezekre nem hivatkozhatunk bármely osztállyal.

Objektumokat, vagy példányokat a következő módon hozhatunk létre:

```
//java
Pelda objektumNeve = new Pelda();
```

```
//C++  
  
Pelda objektumNeve;
```

Az osztálynak létezik egy inicializáló függvénye is amely neve hivatkozik maga az osztály nevére, ezt konstruktornak nevezzük.

```
public class Pelda {  
  
    Pelda () {  
        //Bármilyen inicializálást elvégezhetünk itt  
        System.out.println("A Pelda osztály konstrukrota");  
    }  
}
```

A fenti konstruktornak nincs paramétere és csak egy kiiratás van benne, de több konstruktort is létrehozhatunk, mint a C++-ban több fajta paraméterrel, amelyek alapján már több bejövő adat alapján is inicializálhatunk.

Minden programnak van egy fő része, egy fő függvénye, ezt nevezik a main-nek. A létrehozása eltér a két nyelvben. Míg C++-ban csak egyszerű függvényként hozzuk létre, addig Javában egy fő osztály lesz amely a main-t tartalmazza. A main() függvény:

```
//java  
  
public class Main {  
  
    public static void main(String[] argv) {  
  
    }  
}  
  
//C++  
  
int main(String[] argv) {  
  
    return 0;  
}
```

Létrehozhatunk más függvényeket, azaz metódusokat is amelyek majd a main() metódusban hívhatóak meg, ahol öszpontosul a programunk.

Metódusok:

Egy metódust a következőképpen hozunk létre:

```
metódus_Típusa metódus_Neve () { utasítás1; utasítás2; stb... }
```

Egy osztályban egy metódus lehet public, private és protected, ezt a típus előre írni.

A metódusoknak vagy függvényeknek létezik visszatérési értéke. A visszatérési értéke a metódus típusától függ. A void az egyetlen függvény típus amelynek nincs visszatérési értéke.

A metódusokat a {...} között írt utasítások alkotnak, úgynévezett blokkok. minden utasítást pontosveszővel zárunk le. Léteznek elágazások amelyek minden két nyelvben azonosan működnek. Elágazást kétféleképpen hozhatunk létre, egyszerűt az if kulcsszóval, összetettet a switch kulcsszóval, ez utóbbi esetben eseteket hozunk létre amelyeket case kulcsszavakkal hozunk létre.

Egy metódusnak létezhet ciklusa, az a rész ami az utasításokat tartalmazza. Több ciklusfjata létezik, a java nyelvben a ciklus fajták megegyeznek a C++ nyelvben is létező ciklusokkal.

A ciklusok:

A for() ciklus ami lépésről lépésre végighalad egy megadott struktúrán. Ennek a ciklusnak 3 része van, amelyeket pontosveszővel választunk el egymástól és a ()-zárójelek közé írunk. AZ első rész, a kezdőpont (pl. i = 0) ahonnan kezdjük az elemeket számolni. A második rész a feltétel (pl. i < array.length()), avagy végállapot, azaz meddig olvassuk az elemeket és haladjunk a strukturában. A harmadik rész a lépésköz (pl. i++), tehát az idndexelést milyen nagyságban léptetjük, ha minden elemen végig akarunk haladni, akkor egyesével, ha pedig bizonyos elemekre van szükség akkor több elemet is kihagyva.

A while() ciklus, másnéven előtesztelő vagy kezdőfeltételes ciklus, a ciklus ()-zárójelek között lévő feltétel alapján eldönti hogy lefuttatja-e a benne lévő utasításokat vagy sem, ha igaz a feltétel akkor lefut, egyébként hamis érték esetén nem.

A do {} while(); ciklus, másnéven a hálttesztelő vagy végfeltételes ciklus, a ciklus hasonlít az előzőre, viszont itt a do kulcssónak köszönhetően minden utasítás lefut a ciklusban, aztán következik a while()-ban lévő feltétel kiértékelése, ha ez hamis akkor nem fut le újra a ciklus, egyébként igaz érték esetén újra lefuttatja a benne lévő utasításokat.

A foreach ciklus, amely olyan mint a for() ciklus, viszont ebben nem lehet megadni a lépésközt, és a feltételt, tehát a ciklus a megadott struktura minden elemén végighalad.

Az öröklődés:

Az osztályokat örököltethetjük más osztályokba.

Egy osztály létrehozása a class kulcsszóval majd egy név megadásával történik amelyet feljebb olvashattunk. minden függvényt ki kell fejteni amit létrehoztunk egy egyszerű osztályban.

Egy interfész létrehozása az interface kulcsszóval majd egy névvvel történik. Majd mint egy osztályban függvényeket hozunk létre amelyeknek csak a deklarációja van. A kifejtést egy öröklített osztályban fogjuk végrehajtani.

```
interface Worker_interface {  
    public void getPayment();  
    public void getWorktime();  
}
```

Egy interfész osztálynál az öröklést a következő formula alapján hajtjuk végre az implements kulcsszóval:
class az_Osztályunk_Neve implements az_Osztály_Neve_Amelyből_Örökíteni_Akarunk

Több interfészt is örökölhet egy osztály, így elkülönítve különböző tulajdonságú osztályokat.

Az absztrakt osztály az előbb említett két osztály ötvözetén alapszik, hiszen ebben az osztály típusban van olyan függvény amely csak definiálva van, és van kifejtett is.

Azt a függvényt amelyik csak definiált, a létrehozásnál abstract kulcsszóval kell jelezni, amelyik pedig nincs jelezve azt kötelesek vagyunk kifejteni.

```
class Programmer_abstract extends Worker_abstract {  
  
    public long payment = 210000;  
    public int worktime = 8;  
  
    public void getPayment() {  
        System.out.println("The payment of prog: " + this.payment + ←  
            " Ft.");  
    }  
}
```

Egy abstract osztálynál az öröklést a következő formula alapján hajtjuk végre az extends kulcsszóval:

```
class az_Osztalyunk_Neve extends az_Osztaly_Neve_Amelyből_Örökíteni_Akarunk
```

19.2. Bevezetés a mobilprogramozásba

Az alábbi szövegben a Python programozási nyelv foglalkozni a megadott könyv alapján. Kezdetben a nyelv jellemzőiről olvashatunk. A Python nyelv, más programozási nyelvekkel (C++, Java, C) ellentétben elég csak a forrást megadni, ugyanis a fordítási fázisra itt nincs szükség. A Python használható a neves platformokon, mint például Unix, Windows, MacOS ...stb. A nyelv alkalmas prototípus alkalmazások elkészítésére, hiszen sokkal kevesebb erőfelhasználással lehet benne dolgozni mint például a C++-ban vagy a Javaban.

A Python nyelv egy magas szintű programozási nyelv, mégis egyszerűsége hasonlítható az awk vagy Perl nyelkekhez. A nyelvben a Python kódkönyvtárat használjuk. Az ebben lévo modulok, gyorsabbá teszik a programok fejlesztését. A modulok használhatóak rendszerhívásokra, hálózatkeresésre és fájkezelésre is. A nyelv könnyen olvasható alkalmazást készíthetünk, ennek oka az, hogy az adattípusok engedik, hogy összetett kifejezéseket röviden tudunk leírni. Továbbá a más nyelvekkel ellentétbe a kód csoportosításának tagolása tabulátorral vagy új sorral történik és nem kell definiálnunk a változókat vagy az argumentumokat.

A Python nyelv szintaxisa behúzás alapú. Nem szükséges kapcsos zárójel vagy kulcsszavak használata. Egy blokk végét egy behúzással végezzük, ez lehet akár üres sor. Behúzással nem lehet kezdeni egy szkriptet. minden utasítás a sor végéig tart, így nem szükséges a C,C++ vagy a Java-ban ismert ";" használatára. Ha túl hosszú lenne egy sor, akkor "\n" jellel lehet ezt jelölni. Egy behúzás nem érvényes a folytatósorokra.

A sorokat tokenekre bontja a nyelv, ezek között lehet tetszőleges üres karakter. A token fajták: azonosító operátor, kulcsszó ...stb.

A nyelvben megkülönböztetjük a kis és nagy betűket. Pythonban objektumokkal reprezentálunk minden adatot és az ezzel kapcsolatos műveleteket az objektumhatároz meg. Adattípusok hasonlóan a többi nyelvhez itt is lehetnek, sztringek, számok, ennesek, szótárak és listák. A szám típuson belül lehet egész szám, ami lehet lebegő pontos és komplex szám, ezen belül is decimális, oktális vagy hexadecimális. A lebegőpontos szám a C++ ban ismert double-nek felel meg.

A pythonba megtalálható a szekvencia is, ez egy nem negatív egész számokkal indexelt gyűjtő. A sztringet kétféleképpen lehet megadni: idéziskelek között "példa" vagy aposztrófok között 'példa'. Az ennes típusok vesszővel elválasztott gyűjteményei az objektumoknak. A lista lehet több különböző típusú elemekből.

A elemeket szögletes zárójelek között kell felsorolni. A szótár pedig kulcsokkal azonosított rendezetlen halmaza az elemeknek.

A változók Pythonba objektumokra mutató referenciák. A változóknak itt nincs típusa, így különböző típusú objektumokra lehet hívatkozni. A változóknak a "=" egyenlőség jellel lehet értéket adni. A "del" kulcsszóval törölhetünk egy változó hozzárendelést. A nyelvben két féle változót tudunk megadni, a globálist és a lokálist. A globális változót úgy tudjuk létrehozni, hogy "global", ezzel jelezve, hogy egy globális válzotó lesz. Továbbá fontos, hogy a globális változókat a függvény elején kell felvenni. minden más, a függvényben létrehozott változó alapmérézetben lokális lesz.

A Python rövidrezzárt kiértékelést hajt végre, ezt magyarázzuk egy példával: `a < b and b <= c and c == d` kifejezést a Pythonba így tudjuk írni: `a < b <= c == d`. A beépített típusaink között lehetőség van a típus konverzióra (ilyen az int,float,long is). Képesek vagyunk sztringbol is számot képezni, ezt a használt szárendszer megadásával tudjuk. Pl `int('11',8)`. Ennek az értéke a 9 lesz.

Sokféle műveletet tudunk a szekvenciákon végrehajtani, továbbá beépített függvényeket is alkalmazhatunk rajtuk. Ilyen függvényekkel már találkozhattunk számtalanszor más programozási nyelvekben. Ilyenek a max és min függvények, amivel szélsoérteket tudunk meghatározni, vagy a len() függvény amivel mekapjuk egy szekvencia hosszát. A szekvenciákat '+'-al tudjuk össze fűzni. Tudunk kifejezéseket is alkalmazni rajtuk, ezek az 'in' és a 'not in', ezekkel tudjuk megnézni, hogy eleme-e vagy nem eleme a szekvenciának. Az elemeket indexekkel látjuk el, ezek alapján fogjuk majd elérni, ezt 0-tól szoktuk indítani. Van a negatív indexelés, ez a szekvencia végéről kezdve vissza felé halad. Intervallumot a ':' jellel tudunk megadni. Például ha `a=[5,6,7,8,9]` akkor ha `b=a[1:2]` az eredmény az 6,7 értékek lesznek.

A könyven láthatjuk a további műveleteket mint az ismert `pop([j])` függvényt, ezzel eltávolítjuk a 'j'-ik elemet, ha nem adunk meg elemet, akkor az utolsó elem kerül törlésre. A `reverse()` függvény megfodítja a sorrendjét az elemeknek. Az `append()` függvénytel tudjuk bovíteni egy lista végét, míg az `extend()` függvényel egy másik lista elemeit fűzzük egy lista végéhez. A `clear()` függvénytel pedig töröljük az egész listát.

Most pedig következzenek a nyelv eszközei, ezek közül nézzük meg a `print` metódust, ezzel a metódussal változókat vagy egy sztringet írhatunk ki konzolra.

A python nyelvben is van lehetőségünk az elágazásokra. Ezeket csak úgy mint más programozási nyelvekben if, elif és else kulcsszavakkal hívjuk meg. A szintaktika is hasonló: Első a kulcszó, azután a feltétel és végül pedig hogy mi történjen ha a feltétel teljesül.

A másik nagyon fontos nyelvi eszközök a ciklusok. A programozásban az egyik legfontosabb és leghasznosabb metódus a ciklus, ezzel képesek vagyunk adatokat bejárni, feltölteni, megvizsgálni, keresni és még számtalan hasznos dolog.

Elsőnek nézzük a `for` ciklust. A `for` ciklussal akár szótárakban, tömbökben, vektorokon is végig lehetünk, ha a cikust az előbbiek elemire definiáljuk. Fontos még szót ejteni a `range` függvényről. Ez a ciklus futása közben listát generál egész típusú értékekből.

A következő ciklusunk a `while` ciklus, ez a feltételes ciklus, ugyanis a ciklus addig fut, amíg a megadott feltétel teljesül. Bónusz még, hogy a Python a `break` és `continue` kulcsszavakat is ismeri és támogatja. Használhatunk címkéket és ugrásokat is. A címkék kulcsszava a `label`, ezzel tudunk a programban címkéket elhelyezni, amikhez a `goto` parancsal ugorkhatunk.

A Python programozási nyelvben is léteznek a függvények. Ezeket pedig a `def` kulcsszóval definiálhatjuk. A függvényt hasonlíthatjuk egy értékhez, hiszen tovább lehet adni és másfüggvény is megkaphatja. A függvények vannak paraméterei, ezeket számunkra megfelelo megkötésekkel szintaxissal adhatunk meg.

Fontos dolog, hogy a paraméterek az érték alapján adódnak át, kivételt képez a mutable típusú érték. Az argumentumokat a függvény hívásánál tudjuk megadni. A függvények rendelkeznek egy visszatérési értékkel, de ennesekkel is visszatérhet.

Az Osztályokat és objektumokat is támogatja a nyelv. Ez azt jelenti, hogy tudunk osztályokat definiálni, ezeknek az objektumok lesznek a példányai. Az osztály attributumai: függvények és objektumok. Az osztály lépes örökölni és örökölhetni más osztályokból/nak. Az osztályt a "class" kulcsszóval definiáljuk. A osztályok definíciója az, hogy: már definiált osztályok, opcionális listái vesszővel elválasztva. Attributumokat tudjuk bővíteni az osztályban és a példányokban is. Az attribútumokat az osztály törzsén belül határozzuk meg. Az osztály metódusai hasonlóak mint a globális függvények, annyi különbséggel, hogy itt az első paraméternek kötelezően a selfnek kell lennie, ennek az értéke minden az adott objektumpéldány lesz, melyen a függvényt meghívták.

Az `_intin_` egy speciális konstruktor metódus. A nyelv tartalmaz modulokat a könyebb fejlesztés érdekében. Ilyen például az appuifw, a messaging (SMS, MMS), camera, audio vagy a sysinfo modul. A váratlan helyzetekre itt is tudunk alkalmazni kivitelezést. Az adott kódot a try blokkban megadjuk, ezután egy expect jön, ami hiba esetén fog életbelépni.

Végezetül pedig essen pár szó a PYS60 grafikus felületetől, hiszen minden komolyabb program rendelkezik grafikus felülettel. A GUI szolgál az alkalmazáson belül az információk elrejtéséről, megjelenítéséről. De ezen felül még kezeli is a felhasználói interakciókat.

DRAFT

IV. rész

Irodalomjegyzék

DRAFT

19.3. Általános

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

19.4. C

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

19.5. C++

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

19.6. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf , 2004.

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPORG 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 érdeklődésekért és hasznos észrevételeikért.

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