

Univerzális programozás

Így neveld a programozód!

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert Ács Földesi, Zoltán	2019. szeptember 25.	

REVISION HISTORY

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

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	4
2. Helló, Turing!	6
2.1. Végtelen ciklus	6
2.2. Lefagyott, nem fagyott, akkor most mi van?	8
2.3. Változók értékének felcserélése	10
2.4. Labdapattogás	11
2.5. Szóhossz és a Linus Torvalds féle BogomIPS	14
2.6. Helló, Google!	15
2.7. 100 éves a Brun tétel	17
2.8. A Monty Hall probléma	18
3. Helló, Chomsky!	21
3.1. Decimálisból unárisba átváltó Turing gép	21
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	21
3.3. Hivatkozási nyelv	23
3.4. Saját lexikális elemző	24
3.5. l33t.1	25
3.6. A források olvasása	27
3.7. Logikus	29
3.8. Deklaráció	29

4. Helló, Caesar!	31
4.1. int *** háromszögmátrix	31
4.2. C EXOR titkosító	32
4.3. Java EXOR titkosító	34
4.4. C EXOR törő	35
4.5. Neurális OR, AND és EXOR kapu	37
4.6. Hiba-visszaterjesztéses perceptron	40
5. Helló, Mandelbrot!	42
5.1. A Mandelbrot halmaz	42
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	44
5.3. Biomorfok	46
5.4. A Mandelbrot halmaz CUDA megvalósítása	49
5.5. Mandelbrot nagyító és utazó C++ nyelven	52
5.6. Mandelbrot nagyító és utazó Java nyelven	54
6. Helló, Welch!	57
6.1. Első osztályom	57
6.2. LZW	59
6.3. Fabejárás	64
6.4. Tag a gyökér	65
6.5. Mutató a gyökér	67
6.6. Mozgató szemantika	68
7. Helló, Conway!	69
7.1. Hangyaszimulációk	69
7.2. Java életjáték	73
7.3. Qt C++ életjáték	73
7.4. BrainB Benchmark	77
8. Helló, Schwarzenegger!	81
8.1. Szoftmax Py MNIST	81
8.2. Mély MNIST	84
8.3. Minecraft-MALMÖ	84

9. Helló, Chaitin!	85
9.1. Iteratív és rekurzív faktoriális Lisp-ben	85
9.2. Gimp Scheme Script-fu: króm effekt	86
9.3. Gimp Scheme Script-fu: név mandala	86
10. Helló, Gutenberg!	87
10.1. Programozási alapfogalmak	87
10.2. Programozás bevezetés	88
10.3. Programozás	90
III. Második felvonás	91
11. Helló, Arroway!	93
11.1. OO szemlélet	93
11.2. Homokózó	96
11.3. Gagyí	103
11.4. Yoda	104
11.5. Kódolás from scratch	105
12. „Helló, Berners-Lee!”	108
12.1. Python	108
12.2. C++,java	108
IV. Irodalomjegyzék	110
12.3. Általános	111
12.4. C	111
12.5. C++	111
12.6. Lisp	111

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

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

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

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

DRAFT

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

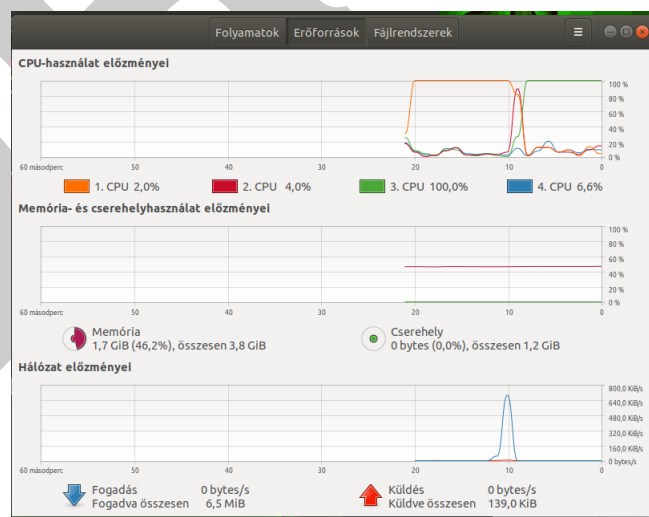
Írj olyan C végtelen ciklusokat, amelyek 0 illetve 100 százalékban dolgoztatnak egy magot és egy olyat, amely 100 százalékban minden magot!

Megoldás forrása:

Kód:

```
#include <stdio.h>

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



Magyarázat: 1.Egy szál 100%-osan futtatása: A main függvényben lévő while ciklus mindaddig lefog futni, amíg a while utáni zárójelben lévő feltétel igaz lesz. Jelen esetben a feltételnek '1' van beírva, ezt a gép mindig igaznak tekinti(a 0-át pedig hamisnak). Ezért ez a ciklus mindaddig futni fog amíg manuálisan ki nem iktatjuk.Ez a ciklus 1 szálát fog folyamatosan 100%-on futtatni. Kód:

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

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

Folyamatnév	Felhasználó	% CPU	Azonosí	Memória	Összes lemez	Összes lemez	Lemez ol
Xorg	foldesizoltan	2	2923	20,7 MiB	3,3 MiB	104,0 KiB	
v2	foldesizoltan	0	4802	64,0 KiB	—	—	
update-notifier	foldesizoltan	0	4235	4,9 MiB	1,6 MiB	5,7 MiB	
systemd	foldesizoltan	0	2773	1,6 MiB	28,4 MiB	420,0 KiB	
sublime_text	foldesizoltan	0	4295	24,9 MiB	2,3 MiB	—	
ssh-agent	foldesizoltan	0	3326	320,0 KiB	—	—	
(sd-pam)	foldesizoltan	0	2774	2,8 MiB	—	—	
pulseaudio	foldesizoltan	0	3461	3,7 MiB	416,0 KiB	8,0 KiB	
plugin_host	foldesizoltan	0	4324	13,2 MiB	6,1 MiB	—	
nautilus-desktop	foldesizoltan	0	3927	32,4 MiB	11,1 MiB	544,0 KiB	
nautilus	foldesizoltan	0	4034	26,8 MiB	4,8 MiB	264,0 KiB	
kdeconnectd	foldesizoltan	0	3946	6,9 MiB	24,8 MiB	—	
ibus-x11	foldesizoltan	0	3649	4,6 MiB	—	—	
ibus-portal	foldesizoltan	0	3653	436,0 KiB	—	—	
ibus-engine-simple	foldesizoltan	0	4051	648,0 KiB	8,0 KiB	—	
ibus-dconf	foldesizoltan	0	3647	652,0 KiB	—	—	
ibus-daemon	foldesizoltan	0	3643	1,7 MiB	8,0 KiB	4,0 KiB	
gvfs-udisks2-volume-monitor	foldesizoltan	0	3678	1,6 MiB	—	—	
gvfs-mtp-volume-monitor	foldesizoltan	0	3756	652,0 KiB	—	—	
gvfs-gphoto2-volume-monitor	foldesizoltan	0	3742	772,0 KiB	748,0 KiB	—	
gvfs-goa-volume-monitor	foldesizoltan	0	3685	552,0 KiB	—	—	
gvfsd-trash	foldesizoltan	0	3989	1,0 MiB	426,0 KiB	—	

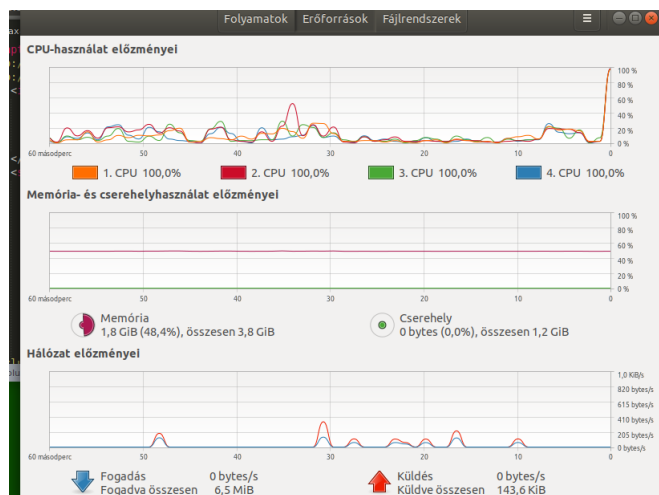
Magyarázat: 2.Egy szál 0%-on való futtatása: Az előzőhöz képest itt is egy végtelen while ciklusunk van a main függvényben, azonban itt megadunk egy sleep nevű utasítást is. A sleep működéséhez include-olnunk kell az unistd.h nevezetű header file-t is. A sleep függvény utáni zárójelben megadhatjuk, hogy az adott szálat hány másodpercig altassa a program. Ezt használhatjuk még programok késleltetésére is. Kód:

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

int main () {

#pragma omp parallel

    while (1) {
}
}
```

Magyarázat: 3. Minden szál futtatása Ahhoz, hogy minden szálát lefoglaljon a programunk, párhuzamossá kell tennünk a program futását. Ehhez include-oljuk az openmp(Open Multi-Processing) alkalmazásprogramozási felületet(API-t). Ezt az omp.h header file-al tehetjük meg. Ezáltal a programunk párhuzamosan fog futni több szálon. A main függvényünk kibővült a #pragma omp parallel sorral. Ez a sor fogja a while ciklusban lévő utasításokat az összes szálra irányítani. Így a végtelen ciklusunk az összes szálon fog egyszerre futni. Ahhoz, hogy ez a program leforduljon használnunk kell a -fopenmp kapcsolót is.

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

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

Megoldás videó:

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

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

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{

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

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

    main(Input Q)
    {
        Lefagy2(Q)
    }

}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

Ha T1000-ben meghívjuk saját magát, és van benne végtelen ciklus akkor a Lefagy függvény igazat ad vissza ha nincs akkor pedig hamisat. A Lefagy2 pedig a Lefagy függvényre épül és ha a Lefagy igaz értéket ad vissza akkor a Lefagy2 igazat ad vissza, ha a Lefagy hamisat, azaz nincs végtelen ciklus a programban akkor a Lefagy2 pedig belép egy végtelen ciklusba, tehát ha nem is volt benne végtelen ciklus mostmár lesz így ellentmondás jön létre. Tehát ilyen program tényleg nem létezhet.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása:

Kód:

```
#include <stdio.h>

int main()
{
    int a=7;
    int b=1;

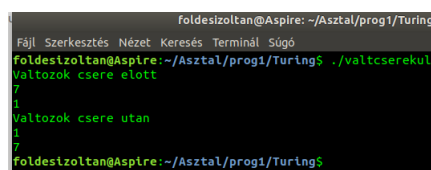
    printf("Valtozok csere előtt\n");

    printf("%d\n",a);
    printf("%d\n",b);

    a=a+b;
    b=a-b;
    a=a-b;

    printf("Valtozok csere után\n");
    printf("%d\n",a);
    printf("%d\n",b);

    return 0;
}
```



```
foldesizoltan@Aspire: ~/Asztal/progi/Turing
Fájl Szerkesztés Nézet Keresés Terminál Súgó
foldesizoltan@Aspire:~/Asztal/progi/Turing$ ./valtcserkul
Valtozok csere előtt
7
1
Valtozok csere után
1
7
foldesizoltan@Aspire:~/Asztal/progi/Turing$
```

Magyarázat: Változók cseréje különbséggel: A main függvényünk elején deklarálunk két int típusú változót(a,b). Jelen esetünkben $a=7$ és $b=1$. A printf függvény segítségével tudunk a konzolra kiírni. A printf utáni zárójelben idézőjelek közé írhatjuk be amit ki szeretnénk írni. A %d kapcsoló azt adja meg, hogy az utána írt 'a' változót egész számként fogja kiírni. A \n kapcsoló pedig a sortörés. Maga a változó értékének cseréje egyszerű matematika. Az 'a' értéke kezdetben 7, 'b' értéke pedig 1. Első lépésként 'a' értékét egyenlővé tesszük 'a' és 'b' összegével, így $a=7+1=8$. Második lépésként 'b' értékét egyenlővé tesszük 'a' és 'b' különbségével, így $b=8-1=7$. Majd utolsó lépésként 'a' értékét egyenlővé tesszük 'a' és 'b' különbségével, azaz $a=8-7=1$. Amint látjuk 'a' értéke 1 lett 'b' értéke pedig 7, tehát megcserélődtek. Kód:

```
include <stdio.h>

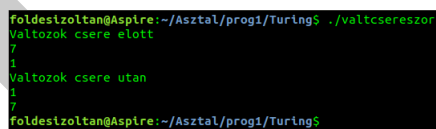
int main()
{
    int a=7;
    int b=1;

    printf("Valtozok csere előtt\n");

    printf("%d\n", a);
    printf("%d\n", b);

    a=a*b;
    b=a/b;
    a=b/a;

    printf("Valtozok csere után\n");
    printf("%d\n", a);
    printf("%d\n", b);
    return 0;
}
```



```
Foldesizoltan@Aspire:~/Asztal/progi/Turing$ ./valtcsereszor
Valtozok csere előtt
7
1
Valtozok csere után
1
7
Foldesizoltan@Aspire:~/Asztal/progi/Turing$
```

Magyarázat: Ez a példa annyiban különbözik az előzőtől, hogy itt az összeadás és különbség helyett szorzást és osztást használunk. Az 'a' értéke kezdetben megint csak 7, 'b' értéke pedig 1. Első lépésként 'a' értékét egyenlővé tesszük 'a' és 'b' szorzatával, $a=7*1=7$. Második lépésként 'b' értékét egyenlővé tesszük 'a' és 'b' hányadosával, $b=7/1=7$. Majd utolsó lépésként 'a' értékét egyenlővé tesszük 'b' és 'a' hányadosával, azaz $a=7/7=1$. Ezek a változó cserék a régebbi időkben voltak hasznosak, ugyanis így nem kellett felesleges erőforrásokat lefoglalni egy segédváltozónak.

2.4. Labdapattogás

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

kon.)

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

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

Kód:

```
#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, "0" );

        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;  
}
```

Magyarázat: A programunk futásához include-olnunk kell a curses.h nevezetű header file-t, melyben a különböző képernyőkezeléshez szükséges függvények találhatóak. A main függvényünk elején a WINDOW paranccsal adjuk meg az ablak ábrázolását, és a későbbiekben ablak néven érjük el. Az initscr függvénnyel "tisztítjuk" le a konzolt. Ezután deklaráljuk az x és y koordinátákat melyek kezdőértékei 0 lesz. Utána az xnov és ynov deklarálása következik melyeknek értékei 1, ezekkel fogjuk növelni a koordinátákat, azaz léptetni a labdánkat. A pattogásunk alapja egy végtelen for ciklus, ezért ez addig fog futni amíg manuálisan ki nem lövjük. A getmaxyx függvény meghatározza a konzolunk maximális koordinátáit. Az mvprintw függvénnyel fogunk x és y koordinátákra jelen esetben egy '0'-t írni. Az usleep mögötti zárójelbe beírt szám fogja meghatározni a pattogás gyorsaságát. Minél nagyobb számot írunk be annál lassabb lesz, ugyanis ez a parancs altatja a ciklusunkat a megadott ideig. Ezután az x és y értékét növeljük xnov-el és ynov-el (jelen esetben 1 el). Az ezután következő 4 db if utasítás segít nekünk abban, hogy labdánk a konzolon belül maradjon. Ha elérte a bal, vagy jobb oldalt, akkor az xnov változó értékét beszorozzuk -1 el, tehát előjelet fog váltani, ezáltal a következő körben csökkeni fog x értéke, tehát visszapattan a labdba. Ha pedig az ablak tetejét, vagy alját érte el akkor ugyanezt csináljuk meg csak az ynov változó értékével.

If nélkül:

Megoldás forrása: 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;  
}
```

A fenti kód ugyanúgy egy labdát fog pattogatni a konzolon, csak ebben az esetben nem használtunk semmilyen logikai utasítást. Ehhez a fordításhoz is szükséges a `-lncurses` kapcsoló. A `WINDOW` utasítással létrehozunk egy mutatót. Az `initscr` függvénnyel előkészítjük az ablakunkat ahol pattogni fog a labdánk. A kód alapja egy végtelen `for` ciklus, vagyis a labdánk mindaddig pattogni fog amíg manuálisan ki nem lövjük a programot. Az `mvprintw` függvényekkel kezeljük a koordinátákat. Először `x=0`-nál kirajzolunk kötőjeleket majd `x=24`-nél is, ezek lesznek az ablakunk határai. Ezután matematikai képletek segítségével kiszámoljuk a koordinátákat, és kiíratjuk őket.

2.5. Szóhossz és a Linus Torvalds féle `BogoMIPS`

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

Megoldás videó:

Megoldás forrása:

Kód:

```
#include <stdio.h>  
  
int main()  
{  
    int a=1;  
    int count=0;  
do  
count++;  
  
    while (a <= 1) ;  
}
```

```
printf("%d", count);
printf("bites a szohossz");
    return 0;
}}
```

Magyarázat: A fenti programunk számítógépünk int típusú változójának méretét fogja megadni. A main függvényben deklarálunk egy 'a' változót és értékét 1-re állítjuk. Ennek segítségével fogjuk mérni a szóhosszt. Valamint deklarálunk egy count nevezetű változót is, melyet majd növelünk és a programunk végén kiíratjuk, ez lesz az int hossza. A main fő része egy do while ciklus. Ez, a sima while-al ellentétben hátultesztelés utasítás, tehát először fogja növelni a count értéket, és csak azután fogja letesztelni a while utáni feltételt. Ez azért fontos mert így tudjuk elérni, hogy az első shift is bele számítsa az eredményünkbe. A szóhosszt az úgy nevezett bitshift operátor segítségével állapítjuk meg. Jelen példánkban a balra shiftelőt használjuk, de van jobbra shiftelő is. Ez az operátor annyit fog tenni, hogy a memóriában eltárolt biteket eggyel balra fogja eltolni és jobbra kiegészíti egy nullával. Tehát például az 'a' értéke egy, ez kettes számrendszerben van eltárolva a memóriában ennek bináris kódja 0001. A bitshift ezt fogja eggyel eltolni tehát 0010 lesz, ami már a 2-es szám. Majd 0100 lesz, ami a 4-es. És minden egyes shiftnél növeljük a count értékét eggyel. Ezt mindaddig fogja így csinálni amíg az adott memóriacímbe csak 0-ások lesznek, tehát az értéke 0 lesz.

2.6. Helló, Google!

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

Megoldás videó:

Megoldás forrása: https://progpater.blog.hu/2011/02/13/bearazzuk_a_masodik_labort#more2657583

Kód:

```
#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;
}
```

Magyarázat:A PageRank a Google keresőmotorjának legfontosabb eleme.Minél többen hivatkoznak egy oldalra, az annál jobb. A fenti C program négy lap PageRank értékét számolja ki.A kódunkhoz szükséges a math.h header file include-olása,ebben matematikai függvények találhatóak, pl.sqrt.A kód elején található egy void típusú függvény.A void annyit takar, hogy a függvény nem fog visszatéríteni semmilyen értéket.A függvény neve után zárójelek között adjuk meg a függvény paramétereit, jelen esetben a függvény egy double típusú tömböt, illetve egy int típusú változót vár. Ez a függvény egy for ciklus segítségével fogja kiírni a konzolra a kapott tömb elemeit.A következő függvény már double típusú,tehát egy double típusú

változót fog visszaadni. Paraméterként pedig két tömböt illetve egy egész számot vár. Ez a függvény fogja számolni a távolságot. A két tömb elemeit egyesével kivonja majd lényegében négyzetre emeli. Majd a függvény végén gyököt vonunk az sqrt segítségével, ezáltal biztos, hogy pozitív eredményt kapunk. A main függvényben lévő 'L' nevezetű mátrix mondja meg, hogy melyik oldalra hányan mutatnak linkekkel. A main függvény fő része egy végtelen for ciklus. A számítást egy másik for ciklussal végezzük el, ez egy sima mátrix szorzás. Ezután az így kapott eredményt átadjuk a tavolsag függvénynek, és ha az így visszakapott érték kisebb mint 0.00000001 akkor megszakítjuk a ciklust és átadjuk az értékeket kiíratásra.

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

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

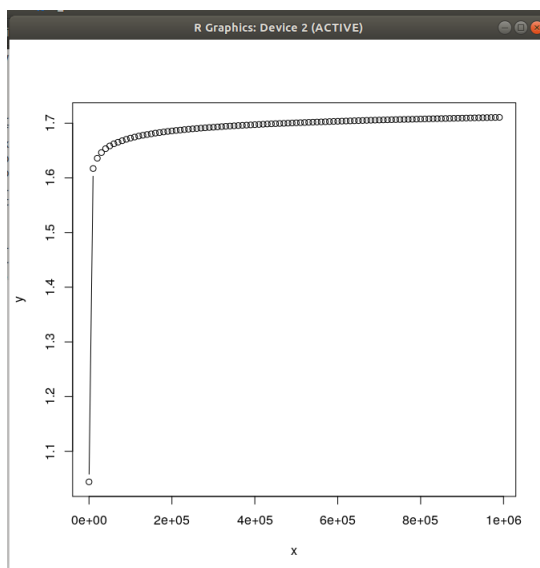
A Brun tétel kimondja, hogy az ikerprímek (olyan prímpárok melyeknek különbsége 2) reciprokösszege egy véges értékhez konvergál. A következő kód ennek a tételnek R-beli megvalósítása.

```
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")
```



Ahhoz, hogy kódunk működjön telepítenünk kell a matlab csomagot, ezt megtehetjük az `install.packages("matlab")` paranccsal. Ezután ezt a csomagot a `library(matlab)` paranccsal be is töltjük. Ezek után `stp` néven létrehozunk egy függvényt amely paraméterül kap majd egy számot. `primes` néven létrehozunk egy vektort amelyben eltároljuk a paraméterként kapott számig a prímszámokat. Ezután a `diff` vektorban kiszámoljuk a prím párok különbségét, ezt úgy tesszük meg, hogy először eltároljuk a `primes` vektor második tagjától a vektor végéig a prímekeket, majd ezekből kivonjuk a `primes` vektor első tagjától az utolsó előtti tagjáig a prímekeket. A következő sorban az `idx` vektorban eltároljuk azoknak a vektoroknak az indexét ahol a különbség kettő (tehát ikerprímek). A `t1primes` vektorban eltároljuk az így kapott prímpárok első tagját, a `t2primes`-ban pedig a párok második tagját. Az `rt1plust2` vektorba kiszámoljuk a két vektor reciprokösszegét, majd `return`-el ezeknek az összegüket írattjuk ki. Az utolsó három sorral pedig ki is rajzoltatjuk az így kapott eredményt.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Tutor: Kiss Máté

Megoldás videó: [https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcsan](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 az USA-ban futó televíziós vetélkedőn alapul. A játékost megkéri a műsorvezető, hogy válasszon 3 csukott ajtó közül. A három ajtó közül az egyik mögött egy nagyon értékes nyeremény van, a másik kettő mögött pedig valami nagyon értéktelen. A játékos választása után a műsorvezető a másik két ajtó közül kinyit egyet ami mögött biztosan nincsen az értékes nyeremény. Ezután a műsorvezető megkérdezi a játékost, hogy szeretné-e a másik ajtót választani, vagy marad az eredeti döntésénél. És itt jön a paradoxon. Ugyanis az első választásnál a játékosnak $\frac{1}{3}$ esélye volt arra, hogy az értékes nyereményt választja. Most a józan ész azt sugallja, hogy mindegy melyik ajtót választja, ugyanis $\frac{50-50\%}{2}$ esélye van a nyereményre, de igazából $\frac{2}{3}$ esélye lenne a nyereményre ha átvált a másik ajtóra. Ezt talán könnyebb elfogadni ha elképzeljük ugyanezt csak 100 ajtóval. Tehát a játékos választ egyet, $\frac{1}{100}$ az esélye arra, hogy eltalálja a nyereményt és $\frac{99}{100}$ az esély arra, hogy valamelyik másik ajtó mögött van. Ezután a játékvezető kinyit 98 ajtót ami mögött nincs semmi és felteszi ugyanazt a kérdést: Váltunk-e ajtót? Talán így könnyebb

belátunk, hogy sokkal nagyobb esélyünk van nyerni ha váltunk a másik ajtóra, ugyanis az az eredeti 99/100 esély oda koncentrálódik.

A következő kód ennek a problémának R szimulációja:

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

for (i in 1:kiserletek_szama) {

  if(kiserlet[i]==jatekos[i]){

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

  }else{

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

  }

  musorvezeto[i] = mibol[sample(1:length(mibol),1)]

}

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

for (i in 1:kiserletek_szama) {

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

}

valtoztatesnyer = which(kiserlet==valtoztat)

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

Kódunk elején a kiserletek_szama vektorban mentjük el, hogy hányszor hajtuk végre a szimulációt. Ezután a kiserlet vektorba tároljuk el, hogy melyik ajtó mögött lesz a nyeremény, 1-3 ig fog egy számot random választani annyszor ahányszor szimuláljuk. A jatekos vektorban ugyanezt csináljuk, ez fogja megadni a játékos választását. A musorvezeto vektor azt az ajtót fogja tárolni amit a műsorvezető fog kinyitni. Ezt úgy határozzuk meg, hogy indítunk egy for ciklust ami végigmegy az összes létrejött eseten. Ezután egy if segítségével először megnézzük azt az esetet amikor a játékos eltalálta a nyereményt rejtő ajtót, azaz a

kísérlet és a játékos vektor egyenlő. Ha ez az eset áll fent akkor a maradék két ajtó közül random választ egyet a program. Ha pedig nem egyezik meg a választása a játékosnak a nyereményt rejtő ajtóval akkor, azt a két ajtó kivesszük a választásból és a maradék egy ajtót fogja kinyitni a műsorvezető. Ezután a nem-változtatásnyer vektorban eltároljuk azokat az eseteket amikor akkor nyerne a játékos ha nem változtat. A változtat vektorban pedig azokat az eseteket amikor akkor nyerne amikor változtat. Ezután kiíratjuk ezeket.

DRAFT

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet grájával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása:

```
#include <stdio.h>

int main() {
    int sz=0;
    int a=0;
    printf("Adj meg egy számot");
    scanf("%d", &sz);
    printf("Unárisba atvaltva:");
    for(int i=0;i<sz;i++)
    {
        printf("%d",1);
    }

    return 0;
}
```

A fent látható program, egy decimális számot fog bekérni a konzolról és azt a számot unáris számrendszerbe fogja kiírni. Az unáris számrendszerbe úgy váltunk át, hogy annyiszor fogunk kiírni egy db egyest amennyi a szám értéke. Tehát pl ha beírjuk, hogy 5 akkor 11111-et fog kiírni. Ezt egy egyszerű for ciklussal tesszük meg. A Turing gép úgy végezné ezt az átváltást, hogy a szám végéről indulva, ha 0-át lát, akkor kilencre vált, és mindig ki vonna egy számot, tehát 8,7,6,5,4,3,2,1,0 és a kivont egyeseket eltárolja, majd a folyamat végén kiírja őket.

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

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

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

S, X, Y legyenek változók
a, b, c legyenek 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 ($aY \rightarrow aa$)
aabbcc

de lehet így is:

S ($S \rightarrow aXbc$)
aXbc ($Xb \rightarrow bX$)
abXc ($Xc \rightarrow Ybcc$)
abYbcc ($bY \rightarrow Yb$)
aYbbcc ($aY \rightarrow aaX$)
aaXbbcc ($Xb \rightarrow bX$)
aabXbcc ($Xb \rightarrow bX$)
aabbXcc ($Xc \rightarrow Ybcc$)
aabbYbccc ($bY \rightarrow Yb$)
aabYbbccc ($bY \rightarrow Yb$)
aaYbbbccc ($aY \rightarrow aa$)
aaabbccc

A, B, C legyenek változók
a, b, c legyenek konstansok

$A \rightarrow aAB$, $A \rightarrow aC$, $CB \rightarrow bCc$, $cB \rightarrow Bc$, $C \rightarrow bc$

A ($A \rightarrow aAB$)
aAB ($A \rightarrow aC$)
aaCB ($CB \rightarrow bCc$)
aabCc ($C \rightarrow bc$)
aabbcc

de lehet így is:

A ($A \rightarrow aAB$)
aAB ($A \rightarrow aAB$)
aaABB ($A \rightarrow aAB$)
aaaABBB ($A \rightarrow aC$)
aaaaCBBB ($CB \rightarrow bCc$)
aaaabCcBB ($cB \rightarrow Bc$)
aaaabCBcB ($cB \rightarrow Bc$)
aaaabCBBc ($CB \rightarrow bCc$)
aaaabbCcBc ($cB \rightarrow Bc$)

```
aaaabbCBcc (CB->bCc)
aaaabbbCccc (C->bc)
aaaabbbbcccc
```

A generatív grammatika különböző transzformációs szabályokat tartalmaz. Ezekhez szükségünk van egy alap szimbólumra, és ezen a szimbólumon tetszés szerint bármelyik szabályt bármennyiszer elvégezhetünk. Az a lényege, hogy egy fajta szimbólumot egy féleképpen tudunk létrehozni, ha több féleképpen is elő tud állni akkor ez a nyelv már nem generatív.

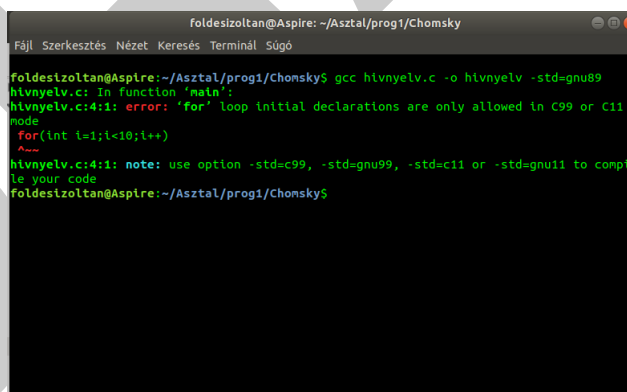
Először megadunk konstansokat és változókat is. Ezután leírjuk a megfelelő nyelvtani szabályokat. Ezt nyilak segítségével tehetjük meg. Bal oldalon megadjuk hogy miből, jobb oldalon meg hogy mi lesz belőle. Ezt rövidebben is láthatjuk alatta. A második generatív nyelv létrehozásánál lényegében elég volt a változók nevét átírni a változók neveit, és máris egy másik generatív nyelvet kapunk amely ugyanazt a nyelvet generálja.

3.3. Hivatkozási nyelv

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

```
#include <stdio.h>

int main() {
    for(int i=1; i<10; i++)
        printf("lefut");
}
```



1989-ben adták ki a C c89-es szabványát és 1999-ben a c99-et. Ezekben a szabványokban különböző szemantikai szabályok érvényesek. A fenti kódban egy ilyen különbségre láthatunk példát. Első ránézésre valószínű fel sem tűnik benne semmi érdekes. Egy sima for ciklus benne egy utasítással. Azonban a for ciklust a c89-es szabvány szerint így nem adhatjuk meg, ugyanis ebben még tilos volt a zárójelek között deklarálni a ciklusváltozót. Ahhoz, hogy ezt lássuk, a megadott szabvány szerint kell lefordítanunk a programot. Ezt a -std=gnu89 kapcsolóval tehetjük meg. Ez a c89-es szabvány szerint fogja lefordítani a programunkat. Így meg is kapjuk azt a hibakódot miszerint c99 szerint kellene fordítanunk a programot mert amit próbálunk csinálni az eszerint a szabvány szerint nem megengedett.

3.4. Saját lexikális elemző

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

Kód:

```
%{
#include <string.h>
int  szamok_szama =0;
}%
%%
[0-9]+      {++szamok_szama;}

%%

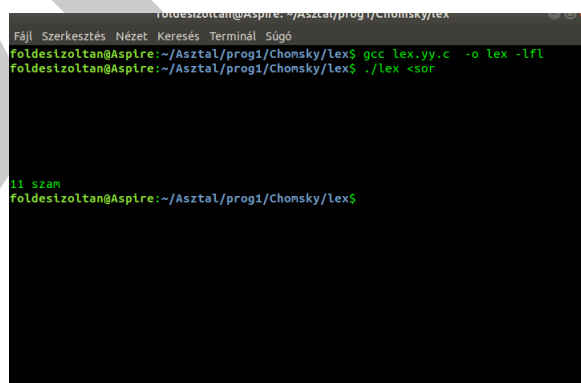
int
main()
{
    yylex();
    printf("%d szam\n",szamok_szama);
    return 0;
}
```

Erre a bemenetre



```
1 2 3 4 5 6
7
8
9 42
10
```

Ezt a kimenetet adja:



```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
foldesztoltan@Aspire:~/Asztal/prog1/Chonsky/lex$ gcc lex.yy.c -o lex -lfl
foldesztoltan@Aspire:~/Asztal/prog1/Chonsky/lex$ ./lex < sor

11 szam
foldesztoltan@Aspire:~/Asztal/prog1/Chonsky/lex$
```

Magyarázat: A fent látható kód, a bemenetén megjelenő számokat fogja összeszámolni egy lexer segítségével. Egy lexer a bemenetén megjelenő szövegeket vizsgálja, és megadhatunk neki különböző mintákat, karaktereket amiket keres. A programot három fő részre lehet bontani, ezeket a fő részeket a % jelek határolják. Az első fő részben adhatjuk meg a különböző definíciókat, jelen esetben egy egész típusú változót

hoztunk létre. A második részben adhatjuk meg, hogy mit keressen a lexer, és ha talált olyan karaktert, mit tegyen vele. Jelen esetünkben az egész számokat fogja keresni, és ha talál egyet, akkor a változó értékét növeli eggyel. A harmadik részben pedig megadjuk, hogy a program végén írassa ki a változó értékét. Ezt a programot először egy .l kiterjesztésbe kell elmentenünk. Fordítása két lépésben történik. Először flex valamivel, ez a parancs létrehoz egy valami.yy.c kiterjesztésű fájlt, majd ezt a fájlt kell c fordítóval lefordítanunk a -lfl kapcsoló segítségével.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás forrás: https://gitlab.com/nbatfai/bhax/blob/master/thematic_tutorials/bhax_textbook_IgyNeveldaProgChomsky/l337d1c7.l

Tutorált: Kiss Máté

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

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

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

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

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

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

};

%}
%%
. {

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

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

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

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

            found = 1;
            break;
        }

    }

    if(!found)
```

```
        printf("%c", *yytext);  
  
    }  
%%  
int  
main()  
{  
    srand(time(NULL)+getpid());  
    yylex();  
    return 0;  
}  
  
}
```

Magyarázat:A leet egy főleg az interneten elterjedt rejtnyelv. Különböző számokkal és jelekkel helyettesítenek betűket, az alapján h mihez hasonlítanak.A fenti kód egy ilyen szöveget alakít olvashatóvá, vagy sima szöveget alakít át.Ennek a programnak is a lexer az alapja csak úgy mint az előző feladatnál.

A kód elején megadtunk egy struktúrát, ez tartalmazni fog egy c változót, ez lesz a betű amit át szeretnénk majd alakítani. Illetve tartalmazni fog még egy négy pointerből álló tömböt.Ezután feltöltjük a tömböt. A kód második részében adjuk meg,hogy mit csináljon a karakterekkel. A pont a bal oldalon azt jelenti, hogy bármilyen karaktert észlel a mellette lévő kód részlet lefog futni.Ha bármilyen karaktert észlel akkor megfogja keresni a tömbben. Ezután egy random számot fog generálni. Ha az a random szám kisebb lesz mint 91 akkor a helyettesítő karakter tömb első karakterével fogja helyettesíteni az adott karaktert. Ha kisebb mint 98 akkor a másodikkal és így tovább. Tehát általában az első helyettesítővel fogja helyettesíteni. A main függvényben csak a random számgenerátort inicializáljuk illetve elindítjuk a karakter beolvasást.

3.6. A források olvasása

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

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

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



Bugok

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

i.

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

Ha a SIGINT jelkezelése nem volt figyelmen kívül hagyva akkor most se legyen. Ha figyelmen kívül volt hagyva akkor a jelkezelő függvény kezelje.

ii.

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

Ez egy for ciklus. A ciklusváltozónk az 'i' melynek kezdőértéke nulla. A ciklus addig fog futni amíg az i kisebb lesz mint 5, és minden ciklus végén az 'i'-t növeljük eggyel.

iii.

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

Ez a for ciklus első ránézésre olyan mint az előző, azonban van egy különbség. Az előző ciklusnál ++i volt most pedig i++. Ennek a for ciklusnál nincs nagy jelentősége, mindkettő növelni fogja eggyel i értékét. Azonban vannak olyan helyzetek amikor nagy jelentősége lehet. Pl i=1 v=i++ és v=++i. Az elsőnél v értéke 1 lesz miközben i értéke kettő, mert hamarabb fogja v felvenni i értékét minthogy növelné eggyel i értékét. A másodiknál pedig előbb növeli, aztán teszi egyenlővé, szóval kettő lesz v értéke.

iv.

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

Egy tömböt tölt fel i értékeivel. Jelen esetben egy öt elemű tömböt hoz létre.

v.

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

Ebben a for ciklusban az i kisebb mint n mellett van még egy feltétel, és a ciklus csak akkor fog lefutni ha mindkét feltétel igaz lesz. A második feltétel nem lesz jó, mert nem logika értéket fog visszaadni (== kellene).

vi.

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

A printf segítségével kiíratunk két egész számot melyeket egy egy függvény fog meghatározni.

vii.

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

Kiíratunk két egész típusú számot, az egyik egy függvény fogja meghatározni.

viii.

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

Ugyanaz mint az előző csak itt az f függvény egy memóriacímet fog kapni

Megoldás forrása:

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat...

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

Tutor: Kiss Máté

Ennek a feladat megoldásához a következő logikai összekötőjeleket kell ismernünk: \neg =negáció(tagadás), \vee =vagy, \wedge =és, \rightarrow =ha...akkor, \leftrightarrow =akkor és csak akkor. A kvantorok \exists =létezik, és a minden \forall . Az első: Bármelyik x számhoz, tartozik egy olyan y prím szám, ami x -nél kisebb. Második: Bármely x szám mellett van olyan y prím és attól 2-vel nagyobb prím ami nagyobb mint az x . Harmadik: Ha van y prím szám, akkor bármelyik x nála kisebb lesz. Negyedik: Ha van olyan y amitől bármelyik x nagyobb akkor az x biztosan nem prím.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciája
- egészek tömbje
- egészek tömbjének referenciája (nem az első elemé)
- egészre mutató mutatók tömbje
- egészre mutató mutatót visszaadó függvény
- egészre mutató mutatót visszaadó függvényre mutató mutató
- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

Mit vezetnek be a programba a következő nevek?

- ```
int a;
```

Egy int típusú, azaz egész számot vezet be, melynek neve a lesz

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

A 'b' tárolni fogja 'a' memóriacímét, azaz 'b' 'a'-ra mutató mutató lesz.

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

Itt az r nem változó hanem egy memóriacím, mégpedig 'a' memóriacíme.

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

Egy egész típusú 5 elemű tömb deklarálása. Az indexelés 0-tól kezdődik.

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

c 5 elemű tömb, minden eleméhez referenciát rendel.

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

d 5 elemű tömb összes tagja egy mutató lesz.

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

int típusú pointert visszaadó függvény.

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

Egész típusra mutató pointert visszaadó függvény

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

Egészlet visszaadó, két egészet kapó és függvényre mutató pointert visszaadó függvény.

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

Függvényt mutatót visszaadó, ami egy olyan függvényre mutat ami egészet ad vissza és két egészet kér be.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

4. fejezet

Helló, Caesar!

4.1. int *** háromszögmátrix

Megoldás videó:

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/thematic_tutorials/bhax_textbook_IgyNeveldaProyCaesar/tm.c?fbclid=IwAR1E0hLaxtbHsS3uRAStdKaeBQPqZY6yht_srjKI4RJzSG-UVQRxAL_vFfA

Kód:

```
#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++) //megint kiíratjuk
{
    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;
}
```

A fent látható kód egy alsó háromszögmátrixot fog kiíratni. Az `nr` változóban adhatjuk meg, hogy hány soros legyen a mátrix, jelen esetünkben 5 lesz. Ezután a `**tm` el egy pointernek foglalunk le helyet. Majd a `malloc` függvény segítségével annyiszor foglalunk le `double`-nyi méretet ahány sorunk van. A `malloc` alapesetben egy pointert ad vissza, de mi típuskényszerítéssel `double **` ot kapunk vissza. Ha a `malloc` valami hiba során nem tudja lefoglalni a megfelelő mennyiségű memóriaterületet akkor egy `NULL` pointert fog visszaadni, ezáltal az `if` igaz lesz és kilép a programunk, ha nem történik ilyen akkor tovább lép. Ezután indítunk egy `for` ciklust amely 0-tól, jelen esetben négyig fog futni. Ez a `for` ciklus fogja lefoglalni minden sornak a megfelelő méretet. Az első sornak 1-et a másodiknak 2-öt és így tovább. Itt megint csak a `malloc` függvényt használjuk. Ezután még két `for` ciklus segítségével feltöltjük a mátrixot. Majd a végén kiíratjuk a mátrixot.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

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

Kód:

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

#define MAX_KULCS 100
#define BUFFER_MERET 256

int main (int argc, char **argv) // argc - stringek száma, ami az argv-re ↔
    mutat, az **argv pedig tárolni fogja a parancssori argumentumokat
{

    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);

}
}
```

A fenti kód a kizáró vagy (EXOR) segítségével fogja titkosítani a megadott tiszta szövegünket. A program fordítása után, a futtatási parancs után megadjuk a kulcsunkat (ez alapján fogja titkosítani a szöveget) majd "" segítségével átadjuk a fájlt amit titkosítani szeretnénk és ">" segítségével kiíratjuk egy másik fájlba a titkosított szöveget. A kód elején a #define segítségével MAX_KULCS és BUFFER_MERET néven megadunk két konstanszt. A main elején létrehozuk a szükséges változókat. A kulcs_meretet egyenlővé tesszük az strlen függvény segítségével az argv[1] méretével. Az argv tömb tárolja a parancssori argumentumokat, az első tagja igazából a 2. tagja lesz mert 0-tól indul az indexelés, tehát ez a kulcsunk mérete lesz. A strncpy függvény a kulcs változóba másolja át a megadott kulcsunkat. Ezután egy while ciklus következik, ez addig fog futni amíg a fájlunkból adatot tud olvasni, közben eltároljuk az olvasott bajtok számát. Ezek után indítunk egy for ciklust ami az olvasott bajtok számáig fog futni. Ebben a ciklusban zajlik maga a titkosítás. A bufferben lévő bajtokat össze exorálja a kulcs bajtjaival és így egy másik karaktert kapunk. A végén pedig kiíratjuk a titkosított szövegünket.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Forrás: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html?fbclid=IwAR0z9V_UxdHR3gu

```
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();
            }

        }
    }
}
```

Ez a program az eddigiekkal ellentétben java nyelven íródott. Java nyelvű kódot a javac paranccsal tudunk fordítani. A java egy erősen objektumorientált magasszintű programozási nyelv. Amint láthatjuk sokkal rövidebb és könnyebben olvashatóbb mint C-s társa. Javában minden class-nak tekinthető, még a main függvény is. Vannak public és private osztályok. A public osztályokat a programon belül bármi el tud érni, a private classokat pedig csak az adott class tudja. Az exortitkosito class fogja a titkosítást végezni. Három argumentuma van, a kulcs, és a kimenő és bemenő csatorna. Ha kevesebbet kap akkor a throw segítségével hibát fog dobni. Ezután beolvassa a bajtokat és össze exorozza a kulccsal, majd ez kiírja. A mainben lévő try és catch függvénnel hibákat kaphatunk el.

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

4.4. C EXOR törő

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

Megoldás videó:

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

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) //tiszta szöveg ←
    lehet, függvény nézi, kap egy állandó karakter tömböt, meg egy méretet
{
    //tiszta szöveg valószínű, hogy tartalmazza a gyakori magyar szavakat
    return strcasestr(titkos,"hogy") && strcasestr(titkos,"nem") && ←
        strcasestr(titkos,"az") && strcasestr(titkos,"ha"); //megkeresi hogy ←
        van e benne hogy, nem, az, ha
}

void exor(const char kulcs[], int kulcs_meret, char titkos[], int ←
    titkos_meret)
//exor eljárás, kap egy kulcs tömböt, egy kulcs méretet, a titkos tömböt és ←
    a titkos méretét
{
    int kulcs_index=0; //deklaráció

    for(int i=0; i<titkos_meret; ++i) //a ciklus a titkos méretig fut
    {
        titkos[i]=titkos[i]^kulcs[kulcs_index]; //exorral nézi a kulcsokkal a ←
            dolgokat, ya know this bad boi
        kulcs_index=(kulcs_index+1)%kulcs_meret;
    }
}

int exor_tores(const char kulcs[], int kulcs_meret, char titkos[], int ←
    titkos_meret)
//exortores kulcs tömb, kulcs méret, titkos tömb, méret
```

```
{
    exor(kulcs, kulcs_meret, titkos, titkos_meret); //exortörés go for it

    return tiszta_lehet(titikos, titkos_meret); //tiszta lehet megnézi
}

int main(void)
{
    //deklarációk
    char kulcs[KULCS_MERET];
    char titkos[MAX_TITKOS];
    char *p=titkos;
    int olvasott_bajtok;

    //titkos fajt puffereelt berantasa //HÜLP
    while((olvasott_bajtok=
        read(0, (void *) p,
            (p-titikos+OLVASAS_BUFFER<
                MAX_TITKOS)? OLVASAS_BUFFER:titkos+MAX_TITKOS-p)))
        p+=olvasott_bajtok;

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

    //osszes kulcs eloallitasa
    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-titikos))
                                        printf("Kulcs: [%c%c%c%c%c%c%c%c]\nTiszta szoveg: [%s]\n", ii, ji, ki, li ←
                                            , mi, ni, oi, pi, titkos);
                                    //ujra EXOR-ozunk, így nem kell egy második buffer
                                    exor(kulcs, KULCS_MERET, titkos, p-titikos);
```

```
}  
return 0;  
}
```

A kód elején define segítségével vannak deklarálva konstansok, ezekre a későbbiekben könnyebb lesz hivatkozni. Ezután a main függvény előtt vannak deklarálva különböző eljárások és függvények, amiket majd a programunk során használni fogunk. A `tiszta_lehet` függvény az első. Ez paraméterként kap egy karakter tömböt és egy méretet, majd ebben a tömbben a 'hogya', 'nem', 'az' és 'ha' szavakat keresi és ezt adja vissza is. Alatta az `exor` nevezetű eljárás található, azért eljárás és nem függvény mert nincs visszatérítési értéke. Ez az eljárás paraméterként kap egy kulcs tömböt, kulcs méretet illetve a titkos tömböt és titkos méretet. Ezután egy for ciklus segítségével a titkos szöveget bájtonként össze exorálja a kulccsal. Ez alatt található az `exor_tores` függvény amely meghívja az `exor` eljárást. Ezután a `tiszta_lehet` függvény és ennek fogja visszatéríteni az értékét. Ezután következik a main függvény. A deklarálások után egy while ciklussal beolvassuk a bufferbe a titkos szöveget, ezután egy for ciklussal a bufferben lévő maradék helyet nullákkal töltjük fel. Ezután nyolc egybeágyazott for ciklussal melyek mindegyike 0-tól 9-ig fut, előállítjuk az összes lehetséges kulcsot. Ez a program maximum 8 számjegű kulcsot tud feltörni. Ezután meghívjuk az `exor_tores` függvényt és ha igazat ad vissza kiíratjuk a kulcsot és a tiszta szöveget.

4.5. Neurális OR, AND és EXOR kapu

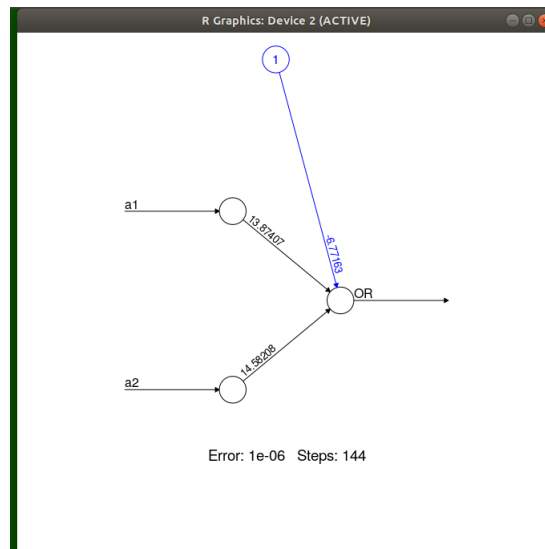
R

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

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

A következő kódokkal neurális hálókat fogunk betanítani az OR, AND és EXOR műveletekre. Kezdjük az OR-al:

```
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])
```



Először is szükségünk lesz a neuralnet package-re, ezt betöltjük. Majd meghatározunk 3 vektort. Itt megadjuk hogyha a1 0 és a2 is 0 akkor az OR is 0 lesz..és így tovább. Ezután ezekből adatot hozunk létre, majd feltöltjük a neurális hálót. Ezután a háló megtanulja és pontosan kiszámolja az OR értékeit. A megadott értékekből megállapítja a súlyokat. És a program végén a compute al számolja ki a végleges számokat.

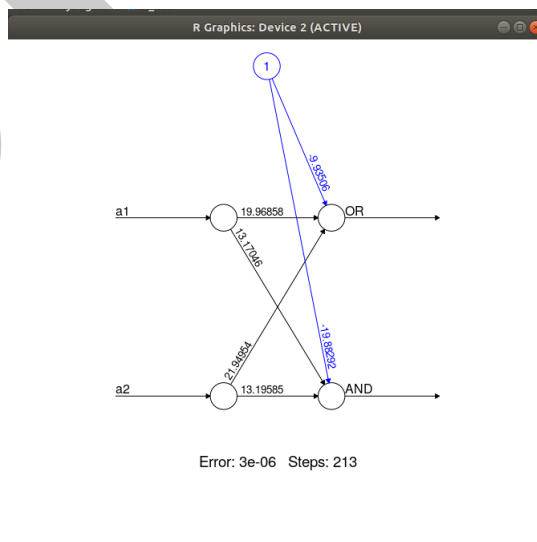
```
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
OR      <- c(0,1,1,1)
AND     <- c(0,0,0,1)

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])
```



Ez a kód pedig az AND logika utasítást fogja megtanulni. Lényegében ugyanaz mint az előző, csak az

OR mellé megadtuk az AND művelet eredményeit is, és ez alapján adja ki az eredményeket. Láthatjuk azt is, hogy nyugodtan ábrázolhatunk több kimenetet is egy ábrán.

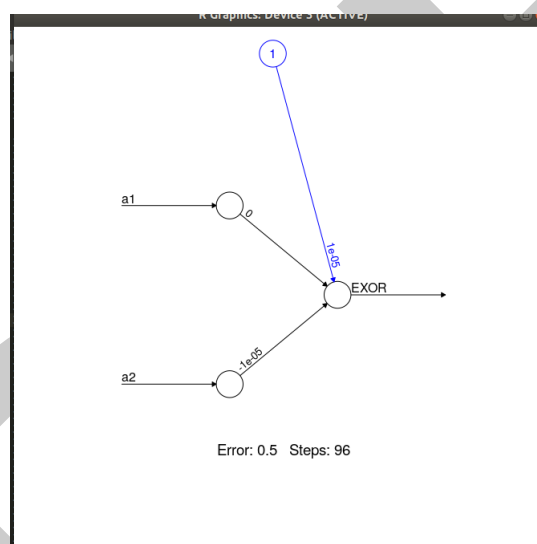
```
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
EXOR    <- c(0,1,1,0)

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])
```



Ebben a kódban az EXORT szeretnénk megtanítani a neuronnak. EXOR-nál 0,1,1,0 kellene kapnunk azonban, láthatjuk, hogy mindenhol 0,5-öt kaptunk, azaz a program nem tudta megtanulni.

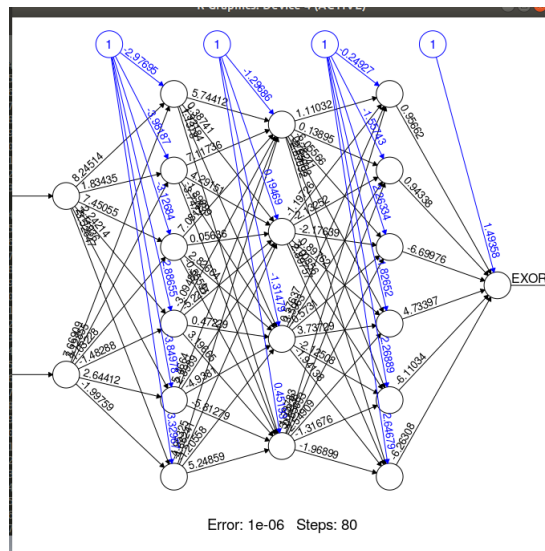
```
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
EXOR    <- c(0,1,1,0)

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])
```

Amint látjuk e fenti program már jól működik. Ugyanis a tanításnál megadtunk pár rejtett hálózatot és így sokkal hatékonyabban és pontosabban tud tanulni.

4.6. Hiba-visszaterjesztéses perceptron

C++

Kód:

```
#include <iostream>
#include "mlp.hpp"
#include "png++/png.hpp"
//g++ mlp.hpp main.cpp -o perc -lpng -std=c++11

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;  
}
```

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

A program futásához szükségünk lesz a libpng csomagra. Ezzel a programmal a mandel.png RGB kódjának a piros részét adjuk át, és a program egy három rétegű hálót alkot belőle. A futáshoz szükségünk van az mlp.hpp fájlra is, ugyanis ez tartalmazza a perceptron classt. A main-ben először lefoglalunk a szükséges helyeket. Ezután a megadott kép méretét eltároljuk egy változóban. Alatta pedig létrehozunk egy perceptron típusú változót, amiben megadjuk, hogy hány rétegű legyen a háló. Majd két egybe ágyazott for ciklussal betöltjük a kép piros részét és ráállítunk egy pontert. Ezután kiíratjuk a megfelelő eredményt, majd felszabadítjuk a memóriaterületet.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

```
// Copyright - no copyright

#include <png++/png.hpp>

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

void GeneratePNG(int tomb[N][M]) {
    png::image<png::rgb_pixel> image(N, M);

    for (int x = 0; x < N; ++x) {
        for (int y = 0; y < M; ++y) {
            image[x][y] = png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y]);
        }
    }
    image.write("kimenet.png");
}

struct Komplex {
    double re, im;
};

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

    int i, j, k;
    double dx = (MAXX - MINX) / N;
    double dy = (MAXY - MINY) / M;
```

```
struct Komplex C, Z, Zuj;

int iteracio;

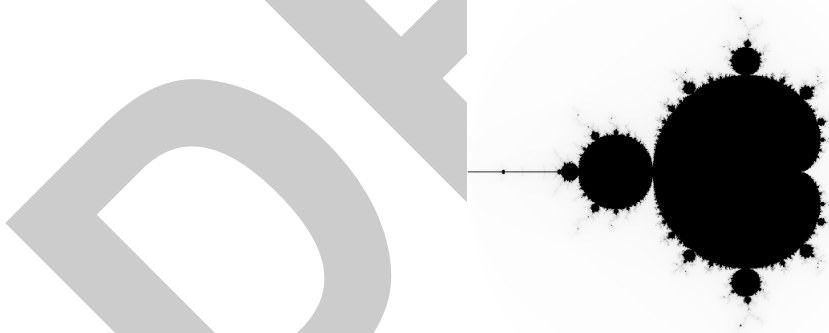
for (i = 0; i < M; ++i) {
    for (j = 0; j < N; ++j) {
        C.re = MINX + j * dx;
        C.im = MAXY - i * dy;

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

        while (Z.re * Z.re + Z.im * Z.im < 4 && iteracio++ < 255) {
            Zuj.re = Z.re * Z.re - Z.im * Z.im + C.re;
            Zuj.im = 2 * Z.re * Z.im + C.im;
            Z.re = Zuj.re;
            Z.im = Zuj.im;
        }
        tomb[i][j] = 256 - iteracio;
    }
}
GeneratePNG(tomb);

return 0;
}
```

A fenti kód a következő png képet fogja generálni:



Ez a mandelbrot halmaz síkbeli ábrázolása. A mandelbrot halmaz komplex számokból áll. Ezekre a számokra igaz az, hogy $X(n+1) = X_n^2 + c$. És ez nem tart a végtelenbe, azaz korlátos. A forráskód fordításához szükség van egy -lpng kapcsolóra, ugyanis a kód tartalmazza a png++ header file-t, ez szükség a png generáláshoz. A kódunk elején deklarálunk pár konstans számot, majd a png generáló eljárást is. Ez paraméterül kap egy NxM-es két dimenziós tömböt. Egy 500x500 -as képet fog létrehozni, két for ciklus segítségével és a tömbben lévő értékek szerint vagy fehér lesz az adott pixel vagy fekete. Az eljárás végén write-al hozzuk létre a png-t. Ezután létrehozunk a Komplex struktúrát amiben egy valós és egy komplex szám lesz. A main elején deklaráljuk a szükséges változókat, és létrehozunk 3 Komplex osztályú változót is. Ezután a megfelelő képletet használva feltöltjük a mátrixot a megfelelő értékekkel és átadjuk a GeneratePNG eljárásnak a

mátrixot.

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

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

int
main ( int argc, char *argv[] )
{
    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double a = -1.9;
    double b = 0.7;
    double c = -1.3;
    double d = 1.3;

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

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

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

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

    for ( int j = 0; j < magassag; ++j )
```

```
{

for ( int k = 0; k < szelesseg; ++k )
{

    reC = a + k * dx;
    imC = d - j * dy;
    std::complex<double> c ( reC, imC );

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

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

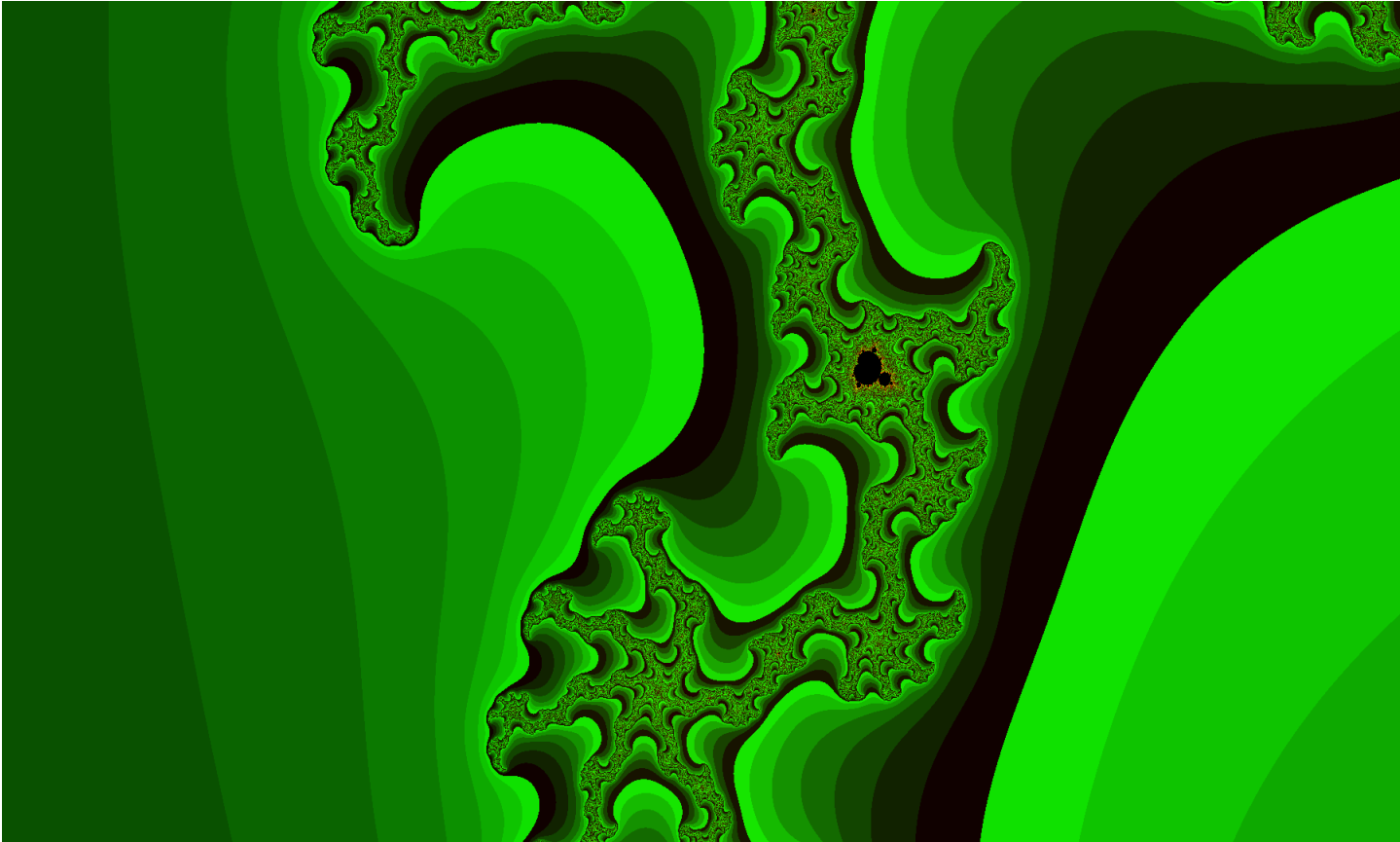
        ++iteracio;
    }

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

}

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

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



Ez a kód nem sokban különbözik az előzőtől. Bekerült a complex header file. Ezáltal nem kell struktúrát használnunk, hogy kezeljük a komplex számokat, hanem ez a header file fogja kivitelezni azt. A kód elején deklaráljuk a szükséges változókat. Futtatáskor meg kell adnunk a kép magasságát és szélességét, azt hogy milyen néven hozza létre a képet, és az n a b c d értékeit. Ezt követően a program beállítja a megfelelő értékekre a változókat. Ha viszont rosszul adtuk meg a dolgokat (pl kevés számot adtunk meg) akkor kiírja a használatot. Ezután a szükséges méretet lefoglaljuk a majd létrejövő képünknek. Maga a számolás a while ciklusban történik. A program futása közben folyamatosan tájékoztat arról, hogy hány százaléknál jár a generálás és arról is ha sikeresen lementette a képet.

5.3. Biomorfok

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

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 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";

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

    for ( int x = 0; x < szelesseg; ++x )
    {

        double reZ = xmin + x * dx;
        double imZ = ymax - y * dy;
        std::complex<double> z_n ( reZ, imZ );
```

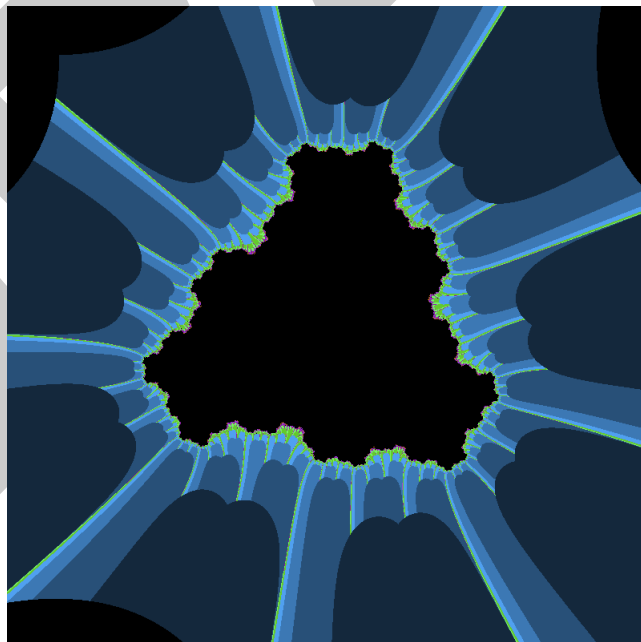


```
int iteracio = 0;
for (int i=0; i < iteraciosHatar; ++i)
{
    z_n = std::pow(z_n, 3) + cc;
    //z_n = std::pow(z_n, 2) + std::sin(z_n) + cc;
    if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
    {
        iteracio = i;
        break;
    }
}

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

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

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



Ez a program lényegében Julia halmazokat fog létrehozni. Julia halmazokból végtelen mennyiségű van. Ennek a programnak az alapja az előző mandelbrotos program. A mandelbrot halmaz tartalmazza az összes Julia halmazt. A mandelbrotos programmal ellentétben itt a `c` nem változó lesz, hanem állandó, és a konzolról kérjük be. A kód elején megint csak létrehozzuk a megfelelő változókat. Majd bekérjük azokat és ha megfelelőek akkor a megfelelő változóhoz rendeljük azokat. Lefoglaljuk a helyet a képünknek. For ciklu-

sukkal végig megyünk az összes pixelen és a megfelelő egyenlet segítségével kiszámoljuk az értékeket. Ez a program is jelzi, hogy hány százaléknál jár és hogy lementette-e már a képet.

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása: https://progpater.blog.hu/2011/03/27/a_parhuzamossag_gyonyorkodtet?fbclid=IwAR0m3eiY

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

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

#define MERET 600
#define ITER_HAT 32000

_device_ int
mandel (int k, int j)
{

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 < iteraciosHatar)
{
    // z_{n+1} = z_n * z_n + c
    ujreZ = reZ * reZ - imZ * imZ + reC;
    ujimZ = 2 * reZ * imZ + imC;
    reZ = ujreZ;
    imZ = ujimZ;
```

```
        ++iteracio;
    }
    return iteracio;
}

/*
_global_ void
mandelkernel (int *kepadat)
{

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

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

}
*/

_global_ void
mandelkernel (int *kepadat)
{

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

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

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

}

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

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

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

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

    cudaMemcpy (kepadat, device_kepadat,
```

```
        MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost);
cudaFree (device_kepadat);

}

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

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

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

    int kepadat[MERET][MERET];

    cudamandel (kepadat);

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

    for (int j = 0; j < MERET; ++j)
    {
        //sor = j;
        for (int k = 0; k < MERET; ++k)
        {
            kep.set_pixel (k, j,
                png::rgb_pixel (255 -
                    (255 * kepadat[j][k]) / ITER_HAT,
                    255 -
                    (255 * kepadat[j][k]) / ITER_HAT,
                    255 -
                    (255 * kepadat[j][k]) / ITER_HAT));
        }
    }
    kep.write (argv[1]);

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

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

    delta = clock () - delta;
```

```
std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << std::endl;
}
```

Ebben a feladatban az első mandelbrotos feladatot fogjuk megoldani a CUDA segítségével. A CUDA segítségével el tudjuk azt érni, hogy a számításokat ne csak a processzerünk végezze, hanem besegítsen a videokártyánk is. A CUDA működéséhez NVIDIA videokártyára lesz szükségünk, ugyanis a CUDA-t az NVIDIA fejlesztette ki. Így a programunk sokkal gyorsabban fogja legenerálni a 600x600-as képünket.

A program fordításához szükségünk lesz az nvidia-cuda-toolkit-re. Ezután a telepítés után nvcc paranccsal tudjuk majd a fordítást végrehajtani. A kód elején a DEFINE segítségével meghatározzuk két állandót, az egyik a kép mérete lesz a másik pedig az iterációs határ. A mandel függvényben végezzük el a szükséges számításokat melyek majd létrehozzák a mandelbrot halmazt. Itt nem használjuk a complex header file-t, ami magától tudná kezelni a komplex számokat. A függvény előtt észrevehetjük, hogy nem simán a visszatérítési típus van megadva, hanem már megadtuk a device szócskát, amivel jelezzük, hogy a GPU befog szállni a számolásba. Mind a device illetve a global szócska arra lesz jó, hogy megadjuk, hogy ezekhez a számításokhoz szálljon be a GPU is. Ezt majd a fordítóprogram intézi el nekünk. A mandelkernel függvény fogja kiszámolni a z értékeket. A cudamandel függvényben lefoglalunk egy 600x600-as tömbnek memóriát. Majd a grid-ben megadjuk, hogy hány blokkot szertnénk létrehozni, a tgridben pedig, hogy egy blokkhoz hány szál tartozzon. Ezután ezeket átadjuk a mandel függvénynek. Ha ez megvan, felszabadítjuk a területet. A main függvény nem sokban különbözik az előző feladathoz képest, csupán annyiban, hogy futási időt is mérünk.

Ez a technológia sokszor használatos képfeldolgozásnál illetve videórenderelésnél.

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!

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

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

    FrakAblak w1;
    w1.show();

    /*
    FrakAblak w1,
    w2(-.08292191725019529, -.082921917244591272,
        -.9662079988595939, -.9662079988551173, 600, 3000),
    w3(-.08292191724880625, -.0829219172470933,
        -.9662079988581493, -.9662079988563615, 600, 4000),
    w4(.14388310361318304, .14388310362702217,
        .6523089200729396, .6523089200854384, 600, 38655);
    w1.show();
    w2.show();
```

```
w3.show();  
w4.show();  
*/  
return a.exec();  
}
```



A fent látható kód működéséhez szükséges telepíteni a libqt4-dev-et. Valamint szükséges összesen 5 fájl. A fájlokat az előbb említett csomaggal fogjuk összefűzni. Először qmake -project paranccsal létrehozunk hozni egy .pro kiterjesztésű fájlt, ennek a fájlnek az utolsó sorába be kell írunk a QT += widgets mondatot. Ezek után qmake *.pro paranccsal létrejön egy makefile. Majd a make paranccsal létrehozzuk a futtatható fájlt. Ezután futtatva a kódot megjelenik a Mandelbrot halmaz, amiben kijelölve egy területet bele tudunk nagyítani szinte a végtelenségig. Ez látható az első és a második képen.

Megoldás forrása:

5.6. Mandelbrot nagyító és utazó Java nyelven

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

```
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("MandelbrotHalmazNagyítás_");
sb.append(++pillanatfelvételSzámláló);
sb.append("_");
sb.append(a);
sb.append("_");
sb.append(b);
sb.append("_");
sb.append(c);
sb.append("_");
sb.append(d);
sb.append(".png");
try {
    javax.imageio.ImageIO.write(mentKép, "png",
        new java.io.File(sb.toString()));
} catch(java.io.IOException e) {
    e.printStackTrace();
}
}
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);
}
}
```

Ebben a feladatban az előző Qt-s projektet írtuk meg java-ban. A lényege annyi, hogy létrehozza a mandelbrot halmazt és bele tudunk nagyítani, illetve a nagyított képet az n betű megnyomásával élesíteni tudjuk. Ebben a programban már be van importálva egy másik program a MandelBrothalmaz.java, ezt az extend szócskával

érjük el, ez olyan mintha C-ben vagy C++-ban include segítségével hozzacsatoltuk volna. A program figyel, amikor megnyomjuk az egeret, és akkor lekérdezi a koordinátákat, majd azt is figyel, mikor engedjük fel, és a két koordináta különbségéből kiszámolja, hogy hova szeretnénk volna belenagyítani. A kódnak van egy kis hibája, ugyanis minden nagyításkor új ablakot nyit. Az új koordinátáknál, számol szélességet és magasságot és újra rajzolja a képet.

DRAFT

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

```
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;
}
```

A fenti kóddal random számokat tudunk generálni. Ezt egy osztállyal valósítottuk meg. Ennek az osztálynak van egy public illetve egy private része. A public részt a programon belül bárhonnán el tudjuk érni, a private rész viszont csak az osztályon belül lesz elérhető. A class elején van megadva a konstruktor és a destruktork. A konstruktor akkor fut le ha egy új példányt szeretnénk létrehozni ilyen típusú osztályként. A destruktork pedig törlésnél fog lefutni, ám ez az esetünkben nem lesz fontos. A következő függvénnyel fogjuk a random számokat generálni, azonban ennek a matematikai háttérével most nem fogunk foglalkozni. A konstruktor meghívásakor a nincsTarolt változó igaz lesz illetve az srand függvény meghívódik. A main függvényben létrehoztunk egy pg nevű változót amely PolarGen típusú lesz, tehát itt meghívódik majd a konstruktor. Valamint egy for ciklussal 10 random számot generálunk le.

A java kód a következőképpen néz ki:

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

Láthatólag sokkal rövidebb lett a kódunk mint c++-ban. Annyiban különbözik, hogy javában minden egy nagy class része, még a main függvény is. Ezenkívül maga a generáló függvény teljesen ugyanaz, és maga a program is ugyanúgy működik.

6.2. LZW

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

https://progpater.blog.hu/2011/03/05/labormeres_otthon_avagy_hogyan_dolgozok_fel_egy_pedat?fbclid=IwAR0p5MQomtcQIdfTeZvPInhgRxu-CCsxGOx453MSrGk

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

typedef struct binfa
{
    int ertek;
    struct binfa *bal_nulla;
    struct binfa *jobb_egy;
```

```
} BINFA, *BINFA_PTR;

BINFA_PTR
uj_elem ()
{
    BINFA_PTR p;

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

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);
    }
}
```

A fent látható program a bemenetére kapott 0-ásokból és egyesekből egy bináris fát fog alkotni. A bináris fára igaz az, hogy mindegyik csomópontjának maximum 0,1 vagy kettő gyermeke lehet. A kód fordításához szükségünk lesz a `-lm` kapcsolóra, ennek segítségével tudjuk majd a matematikai függvényeket használni. Kódunk elején deklaráljuk a `binfa` struktúrát, a `typedef` segítségével. Ez tartalmazni fog egy egész értéket, illetve két mutatót. Az egyik mutató a bal oldali gyermekre fog mutatni a másik pedig a jobb oldalra. Az alatta lévő függvénnyel az új elemeknek fogjuk lefoglalni a megfelelő nagyságú memóriát, ezt a `malloc` segítségével tesszük meg. Ez a lefoglalt területre mutató mutatót fog visszaadni, itt vizsgáljuk azt is ha `NULL` pointer-t ad vissza, ugyanis ekkor nem tudta lefoglalni a területet és ekkor egy `error`-t dobunk. Ezután négy függvényprototípust láthatunk. Ezután következik a `main` ahol először lefoglaljuk a gyökérnek a területet értékét pedig egy `/` jelle állítjuk. Valamint a bal és jobb mutatóját `NULL` pointerre állítjuk. Majd visszaállunk a gyökérre. Ezután elkezdjük beolvasni a számokat a bemenetről. Ha nullást kapunk a bemenetről

akkor először megnézzük, hogy van-e a fának baloldali gyermeke, ha nincs akkor létrehozunk egyet, értéke nullás lesz a pointerek pedig NULL pointerek, és a gyökér baloldali pointerét ráállítjuk. Ha azonban már van baloldali gyermeke akkor addig lépkedünk balra, ameddig nem lesz egy üres hely. Ha egyest kapunk a bementről akkor ugyanezeket a lépéseket tesszük meg csak a jobb oldalra. A main végén a kiir függvénnyel fogjuk kiírni a fánkat. Ez a függvény preorder bejárás szerint fogja kiírni a fát, a fabejárásokról a következő feladatnál beszélünk részletesebben. A szabadit függvénnyel rekurzívan fogjuk felszabadítani az összes lefoglalt területet.

6.3. Fabejárás

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

Tutorált: Kiss Máté

Az előző programunk tökéletes lesz a fabejárások bemutatására. Csak a kiir függvényt fogjuk változtatni. Egy fát három féleképpen tudunk bejárni. Az előző kiir függvényben inorder szerint jártuk be a fát. Az az először a bal oldali részfát jártuk be inoderesen majd a gyökeret és azután a jobb oldalt inoderesen. Most nézzük meg preoderesen a kiir függvényt.

```
void kiir(BINFA_PTR elem)
{
    if (elem !=NULL)
    {
        ++melyseg
        if (melyseg>max_melyseg)
            max melyseg = melyseg;
        for (int i=0; i<melyseg;i++)
            printf("---");
        printf("%c(%d)\n",elem->ertek<2 ? '0' + elem->ertek : elem-> ←
            ertek, melyseg-1);
        kiir(elem->bal_nulla);
        kiir(elem->jobb_egy);

        --melyseg
    }
}
```

Itt láthatjuk, hogy először a gyökeret járjuk be, majd a baloldali részfát preoderesen, majd utoljára a jobb oldali részfát preoderesen. Az utolsó bejárasi mód a postorder bejárás. Nézzük a kiir függvényt :

```
void kiir(BINFA_PTR elem)
{
    if (elem !=NULL)
```

```

    {
        ++melyseg
        if (melyseg>max_melyseg)
            max_melyseg = melyseg;
        kiir(elem->bal_nulla);
        kiir(elem->jobb_egy);
        for (int i=0; i<melyseg;i++)
            printf("---");
        printf("%c(%d)\n",elem->ertek<2 ? '0' + elem->ertek : elem-> ←
            ertek, melyseg-1);
        --melyseg
    }
}

```

Itt először a baloldali részfat járjuk be postorderesen majd a jobb oldali részfat ugyancsak postorderesen, és majd a legvégén a gyökeret. Észrevehetjük, hogy az összes fabejárás rekurzívan történik.

6.4. Tag a gyökér

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

```

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

```

```
        if (!fa->egyenesGyermek ())
        {
            Csomopont *uj = new Csomopont ('1');
            fa->ujEgyesGyermek (uj);
            fa = &gyoker;
        }
        else
        {
            fa = fa->egyenesGyermek ();
        }
    }
}

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

if (b == '0')
{

    if (!fa->nullasGyermek ())
    {
        Csomopont *uj = new Csomopont ('0');
        fa->ujNullasGyermek (uj);
        fa = &gyoker;
    }
}

int main ()
```

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

Ez a program lényegében ugyanaz mint az előző faépítő algoritmus, csak ez c++-ban íródott. A struktúra helyett osztályt, az az class-t használunk. A kód elején megadjuk az LZWBinfa osztályunkat. Ebben beállítjuk a gyökeret illetve a gyermekeit melyek eleinte NULL pointerek lesznek. Majd ezután nézzük meg, hogy egyes vagy nullás jön-e a bemenetről és attól függően, hogy melyik jön ugyanazt csináljuk mint az előző programunkban. Itt annyi a különbség, hogy egyből a fába kerül az adott elem. Ha új csomópontot kell létrehoznunk akkor a new szóval tudjuk azt megtenni. Ugyanis létrehoztunk még egy Csomopont nevezetű osztályt. A nullasGyermek és egyesGyermek függvények az adott gyermekre mutató pointert fognak visszatérési értéként adni. az ujEgyesgyermek és ujNullasgyermek pedig paraméterként kap egy gyermeket és arra fog mutatót állítani. A main függvényünkben csak beolvassuk az adatokat a bemenetről és átadjuk az osztályoknak. Ezután kiíratjuk és a végén az összes helyet felszabadítjuk.

6.5. Mutató a gyökér

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

Az előző bináris fákban a gyökerelem, tagként szerepelt. Azonban most ezt átfogjuk írni egy mutatóra. Első lépésként megkeressük a protected részben a Csomopont gyökeret és mutatóvá írjuk át azt, ezt megtehetjük úgy, hogy egy csillagot teszünk elé.

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

Ezután ha fordítjuk a programot, nagyon sok hibát fog kiírni. Ezeket kijavíthatjuk, ha minden elem valamilyen gyermeknél a pontokat kicseréljük nyilakra. Valamint a gyokernel a referenciajeleket ki kell törölnünk. Ha ezután próbáljuk fordítani a programunkat, le fog fordulni, azonban az első futtatásnál szegmentálási fault-ot fog kidobni, ugyanis olyan memóriacímre szeretnénk hivatkozni amihez nincs jogunk. Szóval le kell foglalnunk a helyet a konstruktorral, illetve a destruktorral ki is kell majd azt törölnünk.

```
class LZWBinFa
{
```

```
public:

    LZWBinFa ()
    {
gyoker= new Csomopont('/');
fa = gyoker;
    }
    ~LZWBinFa ()
    {
        szabadit (gyoker->egyenesGyermekek ());
        szabadit (gyoker->nullasGyermekek ());
        delete(gyoker);
    }
```

Ezután a programunk ugyanúgy fog működni mint ahogy eddig is működött.

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása:

7. fejezet

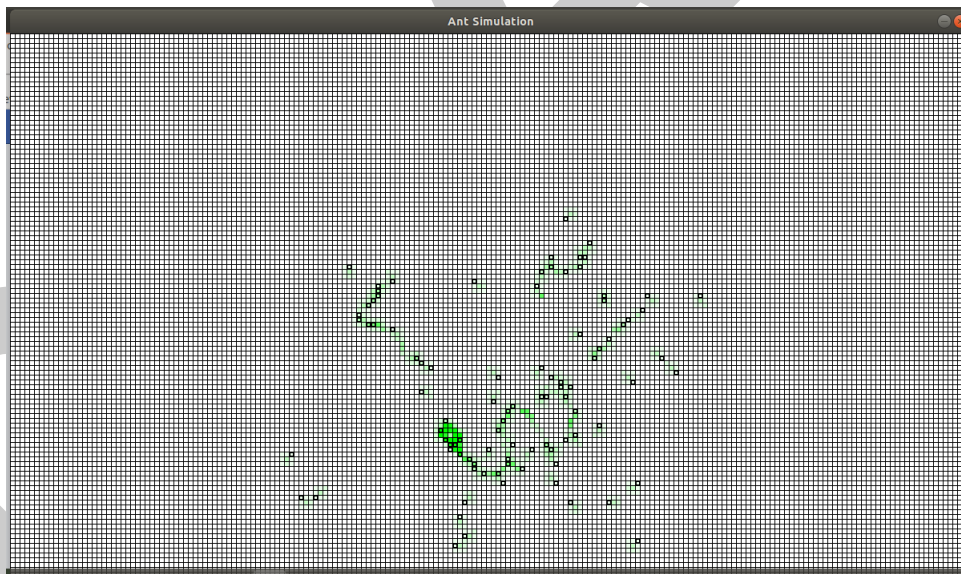
Helló, Conway!

7.1. Hangyaszimulációk

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

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

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/Myrmecologist<https://bhaxor.blog.hu/2018/10/10/myrmecologist>



Ezt a feladatot a Qt segítségével oldjuk meg. Több fájlunk lesz, ezekből majd összeáll a programunk. A program egy hangyaszimuláció lesz. Egy képernyőt felosztunk kis négyzetekre és minden kis négyzet ami ki lesz színezve egy hangyának felel meg.

Az ant.h fájlban lesz megadva az Ant osztály, ezek lesznek a hangyák. Minden hangyának lesz egy x és egy y koordinátája, illetve egy iránya, amit egy random számgenerálással fogunk megadni. Fontos megjegyezni, hogy ezek az adatok publikusak lesznek, azaz más függvények is elfogják érni ezeket.

```
#include "antwin.h"
```

```
#include <QDebug>

AntWin::AntWin ( int width, int height, int delay, int numAnts,
                int pheromone, int nbhPheromon, int evaporation, int ←
                cellDef,
                int min, int max, int cellAntMax, QWidget *parent ) : ←
                QMainWindow ( parent )
{
    setWindowTitle ( "Ant Simulation" );

    this->width = width;
    this->height = height;
    this->max = max;
    this->min = min;

    cellWidth = 6;
    cellHeight = 6;

    setFixedSize ( QSize ( width*cellWidth, height*cellHeight ) );

    grids = new int**[2];
    grids[0] = new int*[height];
    for ( int i=0; i<height; ++i ) {
        grids[0][i] = new int [width];
    }
    grids[1] = new int*[height];
    for ( int i=0; i<height; ++i ) {
        grids[1][i] = new int [width];
    }

    gridIdx = 0;
    grid = grids[gridIdx];

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

    ants = new Ants();

    antThread = new AntThread ( ants, grids, width, height, delay, numAnts, ←
                                pheromone,
                                nbhPheromon, evaporation, min, max, ←
                                cellAntMax);

    connect ( antThread, SIGNAL ( step ( int ) ),
              this, SLOT ( step ( int ) ) );

    antThread->start();
}
```

```
}

void AntWin::paintEvent ( QPaintEvent* )
{
    QPainter qpainter ( this );

    grid = grids[gridIdx];

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

            double rel = 255.0/max;

            qpainter.fillRect ( j*cellWidth, i*cellHeight,
                                cellWidth, cellHeight,
                                QColor ( 255 - grid[i][j]*rel,
                                          255,
                                          255 - grid[i][j]*rel ) );

            if ( grid[i][j] != min )
            {
                qpainter.setPen (
                    QPen (
                        QColor ( 255 - grid[i][j]*rel,
                                255 - grid[i][j]*rel, 255),
                        1 )
                );

                qpainter.drawRect ( j*cellWidth, i*cellHeight,
                                    cellWidth, cellHeight );
            }

            qpainter.setPen (
                QPen (
                    QColor (0,0,0 ),
                    1 )
            );

            qpainter.drawRect ( j*cellWidth, i*cellHeight,
                                cellWidth, cellHeight );

        }
    }

    for ( auto h: *ants) {
        qpainter.setPen ( QPen ( Qt::black, 1 ) );

        qpainter.drawRect ( h.x*cellWidth+1, h.y*cellHeight+1,
```



```
        cellWidth-2, cellHeight-2 );

    }

    QPainter painter;
    painter.begin( &antWin );
    painter.drawRect( 0, 0, cellWidth-2, cellHeight-2 );
    painter.end();
}

AntWin::~AntWin()
{
    delete antThread;

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

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

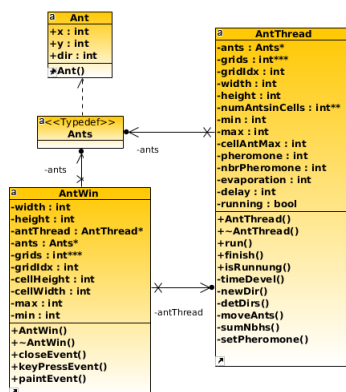
    delete ants;
}

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

Az antwin.cpp-ben hozzuk létre magát az ablakot. Minden kis négyzet 6x6 pixel lesz. For ciklusok segítségével kirajzoltatjuk ezeket a négyzeteket és elkezdjük feltölteni őket hangyákkal. A kód végén felszabadítjuk a memóriát.

Az antthread.cpp fájlban adjuk meg a hangyák viselkedését. A sumNBHS függvénnyel tudjuk számonkövetni az egyes hangyák szomszédszámát. A newDir-el új irányt adunk a hangyáknak. Itt figyeljük a szomszédsági viselkedését, ugyanis a hangyák feromon alapján követik egymást. A feromot láthatjuk is futás közben először nagyon élesen majd egyre halványodik. A moveAnts függvénnyel tudjuk mozgatni a hangyákat.

Az osztálydiagrammban az osztályokat és azok kapcsolatait lehet ábrázolni. Ennek a programnak a következő az osztálydiagrammja.



7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

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

Ebben a feladatban ismét egy Qt-s projektet készítünk el. Most egy sejtautomatát fogunk készíteni C++-ban. Ezeket a sejtautomatákat modellezésre használják. Egy ilyen program leggyakoribb formája két dologból áll. Egy négyzetrács illetve a négyzetrácsban találhatóak a sejtek. A program futása során, ahogy telik az idő a sejtek változtatják állapotukat.

Mi ebben a feladatban az egyik legismertebb ilyen sejtautomatával fogunk foglalkozni, a John Conway által kifejlesztett életjátékkal. Ennek a programnak háttere olyan lesz mint egy négyzetrácsos füzet, és minden sejtnak nyolc darab szomszédja lesz. Ebben a modellben a sejteknek két állapota lehet, vagy élők vagy pedig halottak. A sejtek az idő múlásával megadott szabályok szerint fognak állapotot változtatni. Ezek a szabályok a következők:

- Egy olyan helyre ahol halott sejt van, de van 3 élő szomszédja, egy új sejt születik
- Egy olyan sejt amely él, és van legalább két élő szomszédja, az továbbra is élni fog
- Az összes többi sejt halott lesz

A program több fájlból fog állni.

```
#include "sejtablak.h"
```

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

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

    QDesktopWidget *d = QApplication::desktop();

    int sz = d->width();
    int m = d->height();

    cellaSzelesseg = sz/this->szelesseg;
    cellaMagassag = m/this->magassag;

    setFixedSize(QSize(this->szelesseg * cellaSzelesseg, this->magassag * ←
        cellaMagassag));

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

    racsIndex = 0;
    racs = racsok[racsIndex];

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

    sikloKilovo(racs, 5, this->magassag-20);

    eletjatek = new SejtSzal(racsok, this->szelesseg, this->magassag, 80, ←
        this);
    eletjatek->start();
}

void SejtAblak::paintEvent(QPaintEvent*) {
    QPainter qpainter(this);

    cellaSzelesseg = size().width()/this->szelesseg;
    cellaMagassag = size().height()/this->magassag;
    bool **racs = racsok[racsIndex];
    for(int i=0; i<magassag; ++i) { // végig lépked a sorokon
        for(int j=0; j<szelesseg; ++j) { // s az oszlopok
```

```
// Sejt cella kirajzolása
if(racs[i][j] == ELO)
    QPainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                      cellaSzelesseg, cellaMagassag, Qt::black) ←
    ;
else
    QPainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                      cellaSzelesseg, cellaMagassag, Qt::white) ←
    ;
QPainter.setPen(QPen(Qt::gray, 1));

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

QPainter.end();
}

SejtAblak::~SejtAblak()
{
    delete eletjatek;

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

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

A fenti kód a sejtAblak.cpp. Ezzel hozzuk létre magát a sejtteret. Beállítjuk az ablak nevét, ezt fogja kiírni a program futásakor. A kód elején hivatkozunk a sejtAblak.h file-ra, ebben található a sejtAblak osztály. Ebben a kódban létrehozuk a 100x75-ös ablakot. Valamint létrehozuk a 6x6-os cellákat. Kezdetben minden cella üres lesz. Majd elindítjuk az életjátékot és ha egy sejt élő lesz akkor feketére festi az adott cellát, ha halott akkor pedig fehér marad. A siklikilovo függvény segítségével fognak a sejtek mozogni.

```
#include "sejtszal.h"

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

    racsIndex = 0;
}
```

```
}

int SejtSzal::szomszedokSzama(bool **racs,
                              int sor, int oszlop, bool allapot) {
    int allapotuSzomszed = 0;

    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)

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

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

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

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

    return allapotuSzomszed;
}

void SejtSzal::idoFejlodes() {

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

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

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

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

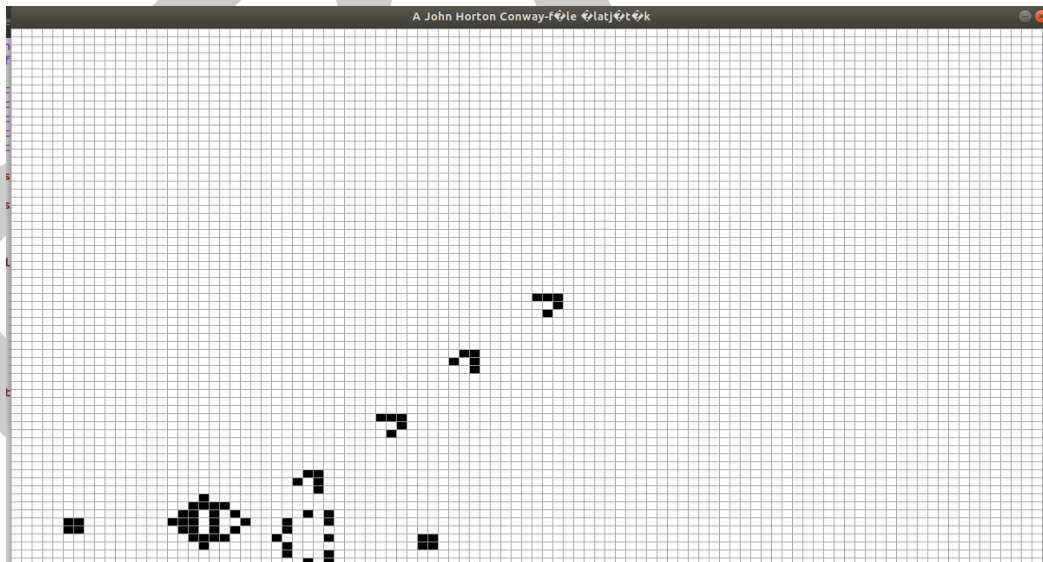
                if(elok==2 || elok==3)
                    racsUtana[i][j] = SejtAblak::ELO;
                else
                    racsUtana[i][j] = SejtAblak::HALOTT;
            } else {
```

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

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

Ez a másik fontos fájl, a sejtszal.cpp. Végigmegyünk for ciklusokkal az adott sejt összes szomszédján (magát kihagyva) és a Conway szabályok szerint figyeljük az állapotukat. Ha két vagy több élő szomszédja van akkor élő marad, ha három élő szomszédja van akkor pedig halott marad.

A main.cpp ezeket a fájlokat fogja össze, igazából minden lényeges rész ezekben található meg. A fordítást a Qt-vel végezzük.



7.4. BrainB Benchmark

Megoldás videó:

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

Tutor: Kiss Máté

Ez a projekt is egy Qt-s projekt lesz. Most egy kis koncentráció teszteléssel foglalkozunk. Ez a program azt fogja vizsgálni, hogy mennyi ideig tudjuk az egerünket egy megadott dolgon tartani, miközben az az adott dolog mozog illetve folyamatosan jelennek meg ugyanúgy kinéző dolgok a képernyőn. Tehát még arra is figyelniünk kell, hogy jó kis kört kövessünk. A kód az eredményt egy külön mappába fogja elmenteni, így figyelemmel kísérhetjük a fejlődésünket.

Ez a program is több file-ból fog állni. Nézzük először a BrainBWin.cpp-t. Először is kiíratjuk az ablakot, ezen belül lesz verziószámunk, illetve egy órát is jelzünk. A kód folyamatosan figyeli a signalokat, illetve követi az egérmozgását. Az egérmozgását csak akkor fogja figyelni ha leván nyomva a bal egérgomb. Ha nagyon messze kerül a kurzorunk a karaktertől akkor egy változóhoz hozzáfog adni egyet, ezt akkor fogja elmenteni ha több mint 12-szer fordul ilyen elő, ez majd beleszámít a végső pontunkba. A programban az S betű lenyomásával menteni tudunk, a P mint pause betűvel pedig szüneteltetni tudjuk a futását.

```
#include "BrainBThread.h"

BrainBThread::BrainBThread ( int w, int h )
{

    dispShift = heroRectSize+heroRectSize/2;

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

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

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

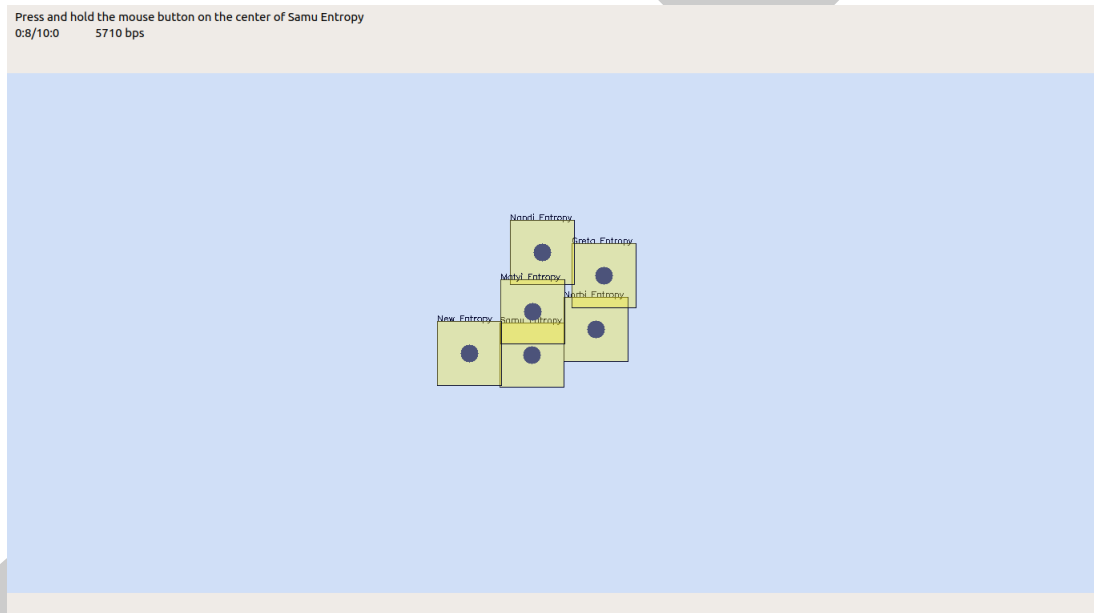
    Hero other1 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - 100,
                  this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), 5, "Norbi Entropy" );
    Hero other2 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - 100,
                  this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), 3, "Greta Entropy" );
    Hero other4 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - 100,
                  this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), 5, "Nandi Entropy" );
    Hero other5 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - 100,
```

```
        this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ),  
        7, "Matyi Entropy" );  
  
    heroes.push_back ( me );  
    heroes.push_back ( other1 );  
    heroes.push_back ( other2 );  
    heroes.push_back ( other4 );  
    heroes.push_back ( other5 );  
  
}  
  
BrainBThread::~BrainBThread()  
{  
  
}  
  
void BrainBThread::run()  
{  
    while ( time < endTime ) {  
  
        QThread::msleep ( delay );  
  
        if ( !paused ) {  
  
            ++time;  
  
            devel();  
  
        }  
  
        draw();  
  
    }  
  
    emit endAndStats ( endTime );  
  
}  
  
void BrainBThread::pause()  
{  
  
    paused = !paused;  
    if ( paused ) {  
        ++nofPaused;  
    }  
  
}  
  
void BrainBThread::set_paused ( bool p )
```



```
{  
  
    if ( !paused && p ) {  
        ++nofPaused;  
    }  
  
    paused = p;  
  
}
```

A másik fontos fájl a BrainBThread.cpp. Ebben létrehozuk az 5 hőst, az egyik mi leszünk. Ezeknek mindnek meghatározzuk a koordinátájukat. A run eljárással fogjuk indítani a program futását, valamint ebben a kódban figyelünk arra is, hogy az adott képfrissítések ne legyenek túl gyorsak az emberi szemnek. A BrainBThread.h-ban van megadva a mi hősünk, Samu néven, nekünk ezen a hősön kell tartani a program futása során a kurzorunkat. A mérést nehezíti, hogy a másik négy hőst a mi hősünkhöz nagyon közel tesszük le, ezért nehéz lehet követni.



8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

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

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

A fenti feladat megoldásához szükségünk lesz a tensorflow telepítésére. Ezt a következőképpen tehetjük meg:

```
sudo apt install python3-dev python3-pip
sudo pip3 install -U virtualenv # system-wide install
virtualenv --system-site-packages -p python3 ./venv
source ./venv/bin/activate # sh, bash, ksh, or zsh
pip install --upgrade pip
pip install --upgrade tensorflow
#ellenőrizzük, hogy helyesen települt-e
python -c "import tensorflow as tf; tf.enable_eager_execution(); print(tf. ↵
    reduce_sum(tf.random_normal([1000, 1000])))"
```

A tensorflow egy ingyenes és nyílt forráskódú könyvtár. Különböző dolgokra lehet használni, köztük gépi tanulásra, azon belül neurális hálókhoz is. Mi ebben a feladatban arra fogjuk használni, hogy segítségével kézzel írt számokat felismerjen a gépünk. Ehhez szükségünk van egy adatbázisra, ez az MNIST lesz. Az MNIST-ben 60.000 kép van és 10.000 teszt kép, melyek kézzel írt arab számokat ábrázolnak.

```
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
```

[illegible]

```
print("-----")

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

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

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

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

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

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

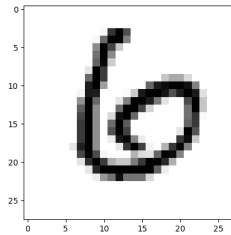
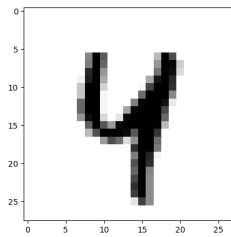
img = mnist.test.images[11]
image = img
matplotlib.pyplot.imshow(image.reshape(28,28), cmap=matplotlib.pyplot.cm. ←
      binary)
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()

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

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

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

A kód elején beimportáljuk az MNIST adatbázisból a képeket, majd a neurálishálózhoz létrehozuk a megfelelő változókat, és az `y`-ban tároljuk az egyenletet, amely 0 vagy 1 értéket vehet fel. Az `x` változóban fogjuk tárolni a képeket. A cross entropy függvényvel azt vizsgáljuk, hogy a becslt illetve a valós értéknek mennyi a különbsége, tehát ez a szám minél kisebb annál pontosabb lesz a program. Ezután lépésekben megtörténik a tanítás, majd a végén teszteljük két képpel a létrejött programot.



```
foldesizoltan@Aspire: ~  
Fájl Szerkesztés Nézet Keresés Terminál Súgó  
-- A hálózati tanítása  
0.0 %  
10.0 %  
20.0 %  
30.0 %  
40.0 %  
50.0 %  
60.0 %  
70.0 %  
80.0 %  
90.0 %  
-----  
-- A hálózati tesztelése  
-- Pontosság: 0.9166  
-----  
-- A MNIST 42. tesztkepek felismerése, mutatom a számot, a továbblépéshez csak  
d be az ablakot  
-- Ezt a hálózati ennek ismerte fel: 4  
-----  
-- A MNIST 11. tesztkepek felismerése, mutatom a számot, a továbblépéshez csak  
d be az ablakot  
-- Ezt a hálózati ennek ismerte fel: 6  
-----  
(veny) foldesizoltan@Aspire:~$
```

Amint látjuk a program futásakor százalékosan jelzi a tanítás állapotát, illetve, hogy az első képet ténylegesen 4-esnek a másodikat pedig ténylegesen hatosnak ismerte fel.

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Minecraft-MALMÖ

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Tutorált: Kiss Máté

Megoldás videó:

Megoldás forrása: [https://hu.wikipedia.org/wiki/Lisp_\(programoz%C3%A1si_nyelv\)#P%C3%A9ldaprogramok](https://hu.wikipedia.org/wiki/Lisp_(programoz%C3%A1si_nyelv)#P%C3%A9ldaprogramok)

Ebben a feladatban a lisp programozási nyelvvel fogunk megismerkedni. Egy rekurzív illetve egy nem rekurzív faktoriális számláló függvényt fogunk létrehozni. A lisp telepítéséről számos videót találunk az interneten. A lisp nyelvet egy általános célú programnyelvnek szánták, azonban a mesterséges intelligencia kutatás nyelveként is használták. A lisp legfőbb adatstruktúrája a láncolt lista. Szintaxisa a prefix jelölés jellemző. A teljesen zárójeles forma könnyíti a programkódok könnyen elemezhetőek, azonban emberi olvasásánál könnyen el lehet veszni a sok zárójelben. Nézzük is a függvényeket. Kezdjük a rekurzív függvénnyel.

```
(defun faktorial (n)
  (if (= n 0)
      1
      (* n (faktorial (- n 1)))))
```

Amint látjuk az összes kifejezés teljesen zárójelezett. A defun szócskával tudjuk deklarálni a függvényt, mögötte zárójelben pedig láthatjuk, hogy egy n változót fog kapni paraméterként. Ezután egy if - el megnézzük, hogy a kapott szám 0-e, ha igen akkor egyet fog visszaadni a függvény. Ezt is prefix módon adjuk meg tehát a $(= n 0)$ azt jelenti, hogy $n=0$. Ha nem 0 akkor pedig n -t megszorozzuk a faktorial $n-1$ -el, ezeket is prefix módon írjuk be. A faktorial szóval hívjuk meg saját magát, és ez mindaddig fogja magát meghívni amíg nem nulla lesz, és akkor már tudjuk az eredményt, hogy 1. Ezután visszafele fog dolgozni a függvény és a végén megkapjuk az eredményt. Most nézzük meg ugyanezt nem rekurzívan.

```
(defun faktorial (n)
  (loop for i from 1 to n
        for fakt = 1 then (* fakt i)
        finally return fakt))
```

Ugyanúgy létrehozzuk a függvényt. Majd a loop for szócskával egy ciklust indítunk ami 1-től fog a kapott számig menni. Majd ezután egy változót indítunk ami minden egyes körben egyenlő lesz az akkori érté-

ke*az adott i értékével.Így kapjuk meg a faktoriális értékét, majd a függvény végén a return-el visszaadjuk ennek a változónak az értékét,ez lesz az eredmény.

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

Az SMNIST kísérletet megcsináltam lvl.9-ig ezért ezt a feladatot passzoltam.

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Az SMNIST kísérletet megcsináltam lvl.9-ig ezért ezt a feladatot passzoltam.

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

[?]

A könyv elején megismerkedünk az alapfogalmakkal. A számítógépek programozási nyelvének 3 szintjét különböztetjük meg: gépi nyelv, assembly szintű nyelv illetve magas szintű nyelv. Mi a magas szintű nyelvekkel foglalkozunk részletesebben. Az ezen a nyelveken megírt programot forrásprogramnak nevezzük. Ezeknek vannak szintaktikai szabályai, ezek az adott kód nyelvtani szabályai. Valamint vannak a szemantikai szabályok a tartalmi, értelmezési illetve jelentésbeli szabályok tartoznak ide. Ezeket a forrásszövegeket kétféleképpen tudjuk gépi kóddá alakítani: fordítóprogramos vagy interpreteres módszerrel. A fordítóprogram lexikális, szintaktikai majd szemantikai elemzést végez, és ez után generál kódot. Ezzel létrejön egy tárgy kód, ez már gépi nyelvű, de még nem futtatható. Ez a kód átkerül egy kapcsolatszerkesztő programnak, és így áll elő egy futtatható program. Az interpreteres módszer anniban különbözik, hogy az nem alkot tárgykódot hanem egyből végrehajtja a forráskód feladatait, így egyből kapunk eredményt. Egy program saját szabványát hivatkozási nyelvnek nevezzük. Ebben vannak definiálva a szemantikai illetve a szintaktikai szabályai az adott nyelvnek. Napjainkban már programok írásához grafikus integrált környezetet használnak.

A programnyelveket három féleképpen tudjuk csoportosítani: Imperatív, deklaratív illetve máselvű nyelvek. Az imperatív nyelvek algoritmikusak, utasításokból állnak a változó a legfőbb programozói eszköz és szorosan kötődnek a Neumann-architektúrához. Ezeknek alcsoportjai az eljárásorientált illetve az objektumorientált nyelvek. A deklaratív nyelvek nem kötődnek annyira a Neumann-architektúrához, nem algoritmikusak, a programozó csak a problémát adja meg, nincs lehetőség memóriaműveletre. Alcsoportjai Funkcionális illetve logikai nyelvek. Minden más nyelv a máselvű nyelvek közé sorolható.

A kifejezések segítségével, különböző értékekből, új értéket tudunk meghatározni. Ezeknek két összetevőjük van, érték és típus. A kifejezések operandusból, operátorokból illetve kerek zárójelekből állhatnak. Az operátorok a műveleti jelek, a zárójelek segítségével a végrehajtás sorrendjén tudunk változtatni, az operandusok pedig az értékek. Vannak egyoperandusú, kétoperandusú és háromoperandusú kifejezések. Operátor sorrend szerint van infix, postfix és prefix ábrázolás. Az a folyamat amikor a kifejezés értéke és típusa kiszámolásra kerül kiértékelésnek nevezzük. Az infix alak nem egyértelmű, az operátorok erősségét egy precedencia táblázatban adják meg. Az infix teljesen bezárójelezett alakja teljesen egyértelmű. A kifejezés típusát két féleképpen lehet meghatározni. Ha ugyan olyan típusú operandusok vannak a kifejezésben akkor típus egyenértékűség lesz, ha nem akkor pedig típuskényszerítés. A konstansok értéke fordítási időben dőlnek el. A C egy kifejezésorientált nyelv. Típuskényszerítő elvét vallja. A könyv itt bemutatja az operátorokat.

Az adattípusok konkrét programozási eszközök, mindegyiknek van egy neve, ez az azonosítójuk. Egy adattípust három dolog határoz meg: tartomány, műveletek, reprezentáció. Az adattípusok tartománya azokat az értékeket tartalmazza amit felvehet. A reprezentáció megadja, hogy hány bájtos lehet az adott típus. Minden típusos nyelv rendelkezik alap típusokkal, de van olyan is ahol a programozó definiálhat saját típust. Ezzel jobban lehetővé téve a modellezést. Saját típus létrehozásakor a fentebb említett három dolgot kell definiálnia a programozónak. Vannak egyszerű illetve összetett adattípusok. Az egyszerűeket már nem lehet tovább bontani, azonban az összetett összes eleme valamilyen egyszerű típusra bontható. Egyszerű típusok: Egész és valós (lebegőpontos ábrázolás), ezek numerikus típusok ezeken numerikus és összehasonlító műveleteket lehet végrehajtani. Karakteres típusok tartományába a karakterláncok tartoznak, a sztring pedig karaktersorozatból áll, szöveges és hasonlító műveleteket lehet végezni. Egyes nyelvek ismerik a logikai típust, ez igaz vagy hamis lehet, logikai illetve összehasonlító műveletek végezhetőek el vele. Összetett típusok: Idetartozik a tömb és a rekord. A tömb elemei ugyanolyan típusúak. A tömböt mint típust meghatározza: dimenzióinak száma, indexkészletének típusa és tartománya és elemeinek a típusa. A C nem ismeri a több dimenziós tömböt ezért úgy képzeljük el, hogy egy tömb elemei tömbök lesznek. A rekord elemei különböző típusúak lehetnek. A mutató egy adott tárterületre fog mutatni. A nevesített konstans három dologból áll: név, típus és érték. Ezt olyankor használjuk ha egy értéket sokszor szeretnénk használni a programban. Az egész program futása során állandó lesz. A változók négy részből állnak: név, attribútumok, cím és érték. A név egy azonosító, a legfőbb attribútum a típus. Többféleképpen lehet deklarálni: Explicit (a program végzi valami eredményeként), implicit (programozó végzi), automatikus (a fordítóprogram végzi). A cím a tárterületet adja meg ahol tárolva van. Többféleképpen lehet tárterületet hozzárendelni, ezt a könyv részletesen taglalja. Értéket adhatunk értékadó utasítással, kezdőértékadással.

Ezután megismerkedünk a C-ben lévő típusokkal majd láthatunk pár példát deklarációra. A typedef-et is taglalja a könyv valamint a struktúra megadását is.

Ezekután az utasításokról olvashatunk részletesen. A fordító ezek segítségével készíti el a tárgyprogramot. Két csoportja van: deklarációs és végrehajtható, az utóbbit használja a tárgykód elkészítéséhez. Ezeknek 9 csoportja van: értékadó utasítás, üres utasítás, ugró utasítás (go to), elágaztató utasítás (if, if else, switch), ciklusszerv utasítások (while, for, do while), vezérlésátadó (continue, break return), I/O illetve egyéb utasítások. Ezeket a könyv részletesen írja le illetve példákat is mutat rá.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

A könyv vezérlési szerkezetek című fejezetében a különböző ilyen szerkezeteket írja le a könyv részletesen. Ezekkel a vezérlési szerkezetekkel határozhatjuk meg, hogy a program a különböző műveleteket milyen sorrendben hajtsa végre. Egy kifejezés, pl egy deklaráció, vagy függvény meghívás akkor válik utasítássá ha egy pontos vesszőt teszünk utána. Kapcsos zárójelek segítségével ún. blokkokat hozhatunk létre. Ezekben a blokkokban több utasítás lehet. Ezt pl az if-nél használjuk. Az if-else utasítással döntéseket hozhatunk. Az if után zárójelek között kell megadnunk a vizsgált kifejezést, és ha az igaz lesz (nem nullát kapunk vissza) akkor az if utáni utasítást végrehajtja a program, ha hamis lesz akkor pedig tovább ugrik az else ágra. Az else ágat el lehet hagyni, illetve az if után blokk is állhat. Az else-if segítségével több if-et is megadhatunk, és a program futása során az összeset megfogja vizsgálni, és ha egyik sem lesz igaz, csak akkor ugrik majd az else ágra. A switch utasítással többirányú elágazást hozhatunk létre a programban. A switch szó után zárójelek között megadjuk a vizsgált kifejezést, majd case szó után megadunk még egy kifejezést, és ha az

a kifejezés igaz lesz akkor az utána lévő utasítást hajtja végre a program. Akármennyi case-t létrehozhatunk. Tehát például ha egy dolgot szeretnénk osztályozni akkor a dolgozat pontszámát beírjuk a switch mögé, a case szavak után pedig a különböző pontszámokat és azután, hogy az hanyas érdemjegynek felel meg. A program ki fogja választani, hogy melyik intervallumba illik be a pontszám és kiírja az érdemjegyet. Az utasítás végén megadhatunk egy default értéket, ez akkor fog végrehajtódni ha egyik esetre se igaz az adott kifejezés. Tehát például ha valakinek több pontja lenne mint a max pontszám akkor a default utasítás hajtódna végre. A break utasítással bármelyik utasításból kiléphetünk manuálisan. A while illetve for utasításokkal ciklusokat hozhatunk létre a programban. A while után zárójelek között kell megadnunk egy kifejezést, ez a ciklus mindaddig fog futni amíg a kifejezés igaz lesz (akkor áll meg ha hamissá válik). Ha igaz, akkor a megadott utasítást végrehajtja. A for utasítással is ciklusokat tudunk létrehozni a programban. Ilyenkor a for szócska után zárójelben 3 kifejezést kell megadnunk. A zárójelben az első kifejezésben megadjuk azt ahonnan indulunk (vagy értékadással vagy egy függvénnyel), a második kifejezés általában egy relációs kifejezés, ezt fogja vizsgálni a program (ha igaz akkor fog lefutni) a harmadik kifejezéssel általában a ciklusváltozóval csinálunk valamit (vagy növeljük vagy csökkentjük). A for ciklust nagyon egyszerűen át tudjuk írni while ciklusra. Az első kifejezést kell a while elé írunk, a második kifejezés a while utáni zárójelbe kerül, a 3. kifejezés pedig utasításként a while utáni blokkba kerül. Az, hogy melyik ciklust használjuk az adott programban, teljesen a programozóra van bízva. Általában azt használjuk amelyik kézenfekvőbbnek tűnik. Ha a for ciklus utáni zárójelek közé két pontosvesszőt írunk, akkor egy végtelen ciklust hozunk létre. Ilyenkor a 2. kifejezést mindig igaznak fogja kiértékelni a program és a break utasítással vagy a return utasítással tudunk manuálisan kilépni a ciklusból. A while és for ciklusok feltételeit a program mindig a ciklus futása előtt ellenőrzi le, így előfordulhat az, hogy az adott ciklus egyszer sem fut le (ha a feltétel kezdetektől fogva hamis). Ezekkel ellentétben a do-while ciklus ún. hátultesztelésű ciklus, azaz egyszer biztosan lefut, mivel csak a ciklus futása végén ellenőrzi a feltétel igazságát. Ezt a ciklust sokkal ritkábban használják, de ritkán nagyon hasznos. A break utasítással az előbb már átbeszélt utasításokból idő előtt ki tudunk lépni. A continue utasítás a switch utasításban nem használható. A while esetén ez az utasítás azt fogja eredményezni, hogy egyből végrehajtja a feltételvizsgálatot. Általánosan pedig a következő utasításra ugrik. A goto utasítással egy címkére ugorhatunk, általában nem használják. De ha igen akkor a leggyakoribb felhasználási módszere, ha több egybeágyazott ciklusból szeretnénk kilépni, ugyanis itt nem használható a break utasítás.

Ezek az utasítások leírásuk sorrendjében hajtódnak végre illetve különböző sorrendbe sorolhatóak. Hat fő csoportba tudjuk sorolni az utasításokat. A címkézett utasításoknál előtagként egy címkét írhatunk az utasítás elé. Ez a címke egy azonosítóból áll. A következő csoport a kifejezés utasítás, a legtöbb utasítás ilyen. Ide tartozik a függvény hívás vagy az értékadás, pontosvesszővel zárjuk le. Az összetett utasítások csoportjába tartozik a blokk. Ezáltal több utasítást egy utasításként tudunk kezelni. Ezt a kapcsos zárójelekkel tehetjük meg. A kiválasztó utasítások közé tartozik az if illetve a switch utasítás. Az ebbe a csoportba tartozó utasítások már megváltoztatják a végrehajtási sorrendet. Az if-el található else utasítás mindig a blokkban lévő utolsó if-el van párban. A switch utasítás után bármennyi case állhat, default viszont csak egy. Az iterációs utasítások csoportjába tartozik a while, do-while illetve for utasítások. Ezek egy ciklust definiálnak. A vezérlésátadó csoportba a goto, continue, break illetve return utasítások tartoznak. Ezekkel feltétel nélkül átadhatjuk a vezérlést. A goto működéséhez szükséges egy címke, amire át fog térni a vezérlés. A continue utasítással egy ciklusban újra megvizsgálásra kerül a feltétel. A break-el a következő utasításra ugordhatunk. A returnnel visszatérítési értéket adhatunk meg.

10.3. Programozás

[BMECPP]

A könyv részletesen fejti ki az objektumorientáltságot. A C++ mint már tudjuk objektumorientált nyelv, ezáltal összetebb problémákat lehet vele megoldani azonban ez teljesítménycsökkenéssel járhat. A C++-ban osztályokat használunk, aminek minden elemét egyednek hívunk. Fontos megemlíteni az adatrejtést, ez azt jelenti, hogy az osztályon belüli bizonyos adatok más programrésznek nem elérhetőek. A könyv tárgyalja a tagfüggvények különböző megadását illetve a tagváltozókról is beszél. Az adatrejtést is kifejti ezután a szerző, elmagyarázza, hogy azért fontos a különböző adatok védelme, hogy azokat más ne tudja megváltoztatni, ugyanis lehet olyan függvény ami azzal az értékkel számol. Ezután szó van a konstruktorról illetve a destruktorról. A konstruktor egy új példány létrehozásakor hívódik meg, és ez foglalja le számára a memóriát. A destruktor pedig egy példány törlésekor felszabadítja a memóriát. A dinamikus adatkezelésről is olvashatunk. Memóriát lefoglalni a new operátorral, a delete operátorral pedig a lefoglalt memóriát törölhetjük. A másolókonstruktorról is olvashatunk, ami egy adott példányt fog lemásolni, ez akkor fog meghívódni ha egy függvényt értékszerinti paraméterátadással hívunk meg, így nem fog változni az eredeti példány. Létezik sekély(bitenként másol) illetve mély másolás. Majd a friend függvényeket írja le a könyv, a friend függvények ugyanúgy hozzáférnek a tagfüggvényekhez és a tagváltozók, mintha az osztály tagjai lennének. Friend osztályok is léteznek, ezeknek is ugyanaz a joguk lesz mintha az adott osztályba tartoznának. A statikus tagváltozók az osztály minden objektumában ugyanazt az értéket veszik fel. Az operátorok kiértékelési sorrendje az ún. precedenci táblázatban van leírva, ezt a sorrendet zárójelek segítségével tudjuk befolyásolni. C++-ban az operátorok túlterhelésével, saját operátorokat hozhatunk létre. Itt a szerző különböző példákkal magyarázza el az operátor túltöltését.

A 10. fejezetben a kivételkezelésről olvashatunk. C-ben a hibákról, a függvények által visszatérített hibakódokkal illetve globális változókkal tudunk tájékozódni. C++-ban azonban létrehozhatunk ún. kivételkezelést. Ezáltal sokkal átláthatóbb illetve könnyebben kezelhető lesz a hibakezelésünk. Ebben a fejezetben megismerjük a C-ben lévő hibakezelés problémáit. Illetve arra is kapunk segítséget, hogy hogyan kereljük el a memóriszivárgást a kivételkezelés során. C-ben ha egy függvényhívás láncot nézünk és valamelyik függvény hibát észlel(például nem tud megnyitni egy fájlt) akkor a visszatérési értékéből tudjuk meg a hibát, ezt a kódot visszaadja egész az első láncszemig, a main függvényig. A main függvényben pedig kezeljük azt. Ez nagyon körülményes illetve nagyon könnyű átsiklani a hibákon, így könnyen előfordulhat az, hogy hibás adattal fogunk dolgozni. Ezeket a hibákat tudjuk kiküszöbölni a C++-os kivételkezeléssel. Kivételkezeléskor ugyanis a futás azonnal a hibakezeléses részlegre fog kerülni. Ezzel a módszerrel nem csak hibát, hanem bármilyen kivételes esetet tudunk kezelni. A 190. oldali példában egy try-catch blokkot találunk. Ha megfelelő számot írunk be akkor a try blokkban lévő majdnem összes utasítás lefog futni, majd a végén kiírja a program, hogy done. Ilyenkor csak a throw utasítás nem fut le. Ha azonban nem megfelelő értéket írunk be akkor a throw utasítással egy kivételt dobunk, jelen esetben const char* típusú stringet, ilyenkor a vezérlés azonnal átkerül az ugyan ilyen kivételt 'elkapó' catch-re, és végrehajtódik a megadott utasítás. Ha egy függvény hívási láncot nézünk akkor ilyenkor egyből a megfelelő catch ágra fog kerülni a vezérlés, és nem a main függvényre. Ha egy kezeletlen kivétellel találkozunk a program akkor az abort függvény hívódik meg amely a programból való kilépést eredményezi. Több try-catch blokkot egybe tudunk ágyazni illetve egy kivételt többször is dobhatunk. A 197. oldalon a verem visszacsévézésére látunk egy példát. Ez az a jelenség mikor egy kivétel dobásakor és elkapásáig a lokális változók felszabadulnak. Ebből az a tanulság, hogy kivétel dobás és elkapás között kód futhat le.

III. rész

Második felvonás

DRAFT

**Bátf41 Haxor Stream**

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

DRAFT

11. fejezet

Helló, Arroway!

11.1. OO szemlélet

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

Ebben a feladatban a polártranszformációs generátort fogjuk megírni javában illetve C++-ban. A teljes programok forrásként megtalálható githubon, ide csak programrészletek lesznek beszúrva.

Ez az algoritmus 10 db random számot fog generálni polártranszformáció segítségével. Kezdjük a javas kóddal. A PolarGenerator osztályban két darab változó található, egy logikai illetve egy double típusú. A logikai nincsTarolt változóval azt jelezzük, hogy van-e tárolt változó vagy sem, a tárolt double változóban pedig a tárolt változó értékét fogjuk tárolni. Ezután található az osztály konstruktora amely a logikai változónkat igazra fogja állítani, ami azt jelenti majd, hogy nincs tárolt változónk.

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

Ezután egy metódus következik következő néven, ez fogja a random számokat generálni. Ez a metódus azt fogja nézni, hogy van-e tárolt változó. Ha van tárolt változó akkor azt fogja visszaadni értékül. Ha nincs akkor pedig két értéket fog kiszámolni, és az egyiket elfogja tárolni a másikat pedig értékül fogja visszaadni.

Ezután van leírva a main függvényünk, ezzel indul a programunk. Először is létrehozunk egy objektumot a PolarGenerator classból, majd egy for ciklus segítségével 10-szer meghívjuk a következő metódust, ezáltal létrehozva a 10 darab random számot.

```
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());
    }
}
}

```

Nézzük meg ugyanezt a kódot C++ nyelven. Különbségek fognak adódni ugyanis C++ nyelven nem minden osztály míg javában mindent osztálynak tekintünk. C++-ban is osztályokkal fogjuk megoldani a feladatot. Név szerint a PolarGen osztállyal. Ennek az osztálynak lesz egy public illetve egy private része. A public részhez a kódon belül bármi hozzá tud érni, míg a private részhez csak az adott osztályon belül tartozó dolgok tudnak hozzáférni. A public részhez három dolog tartozik. Az első rész a konstruktor, melynek ugyanaz a neve mint magának az osztálynak. Ez a konstruktor akkor fog meghívódni amikor az osztályt példányosítjuk és jelen esetben a nincsTarolt változót igazra fogja állítani. A második rész a destruktort amit a ~ jellel jelölünk, ez akkor hívódik meg amikor egy példányt törölünk. Ez a javas kódunkban nem volt, mivel javában egy automatikus szeméthyűjtő van, ami felszabadítja a területet ha egy példányra már nincs több hivatkozás. Ezen kívül a public részhez tartozik még a kovetkezo metódus prototípusa, ezt azért adjuk meg, hogy tudja a program, hogy a későbbiekben lesz majd egy ilyen metódus, csak nem itt írjuk le részletesen.

```

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

```

```
    }  
    double kovetkezo();
```

A private részben a változóink találhatóak amik megegyeznek a javas kódunknál megadott változókkal, ezek azért privátok mert így más programrészlet nem tudja majd megváltoztatni őket.

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

Ezután megadjuk a kovetkezo metódust, ami megint csak megegyezik a javas kódunkban megadott kovetkezo-vel. Majd következik a main amiben létrehozunk egy PolarGen objektumot majd egy for ciklus segítségével létrehozunk 10 random számot.

```
    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;  
}
```


Kimenet:



```
foldesizoltan@Aspire: ~/Asztal/prog2/arroway/polar
Fájl Szerkesztés Nézet Keresés Terminál Súgó
foldesizoltan@Aspire:~/Asztal/prog2/arroway/polar$ java PolarGenerator
-0.40541083242284065
-0.36224767080949155
0.704848912359715
-0.08873189133532974
-3.2392055735183245
0.3773398846938243
-1.2496289845400096
0.4889969279099216
0.9099878142390746
1.4959480369242457
foldesizoltan@Aspire:~/Asztal/prog2/arroway/polar$
```

11.2. Homokózó

Írjuk át az első védési programot (LZW binfa) C++ nyelvről Java nyelvre, ugyanúgy működjön! Mutasunk rá, hogy gyakorlatilag a pointereket és referenciákat kell kiírtani és minden máris működik (erre utal a feladat neve, hogy Java-ban minden referencia, nincs választás, hogy mondjuk egy attribútum pointer, referencia vagy tagként tartalmazott legyen). Miután már áttettük Java nyelvre, tegyük be egy Java Servletbe és a böngészőből GET-es kéréssel (például a böngésző címsorából) kapja meg azt a mintát, amelynek kiszámolja az LZW binfáját!

Ebben a feladatban a már az első fejezetben megírt C++-os binfát fogjuk javában megírni. A binfa működéséről beszéltünk a könyv első fejezetében, most csak a C++ és java különbségeiről fogunk

```
language="java">

public class LZWBinFa {

    public LZWBinFa() {

        fa = gyoker;

    }

    public void egyBitFeldolg(char b) {

        if (b == '0') {

            if (fa.nullasGyermek() == null)
            {

                Csomopont uj = new Csomopont('0');

                fa.ujNullasGyermek(uj);

                fa = gyoker;
            } else
            {

                fa = fa.nullasGyermek();
            }
        }
    }
}
```

```
    }
    else {
        if (fa.egyenesGyermek() == null) {
            Csomopont uj = new Csomopont('1');
            fa.ujEgyenesGyermek(uj);
            fa = gyoker;
        } else {
            fa = fa.egyenesGyermek();
        }
    }
}

public void kiir() {

    melyseg = 0;

    kiir(gyoker, new java.io.PrintWriter(System.out));

}

public void kiir(java.io.PrintWriter os) {
    melyseg = 0;
    kiir(gyoker, os);
}

class Csomopont {

    public Csomopont(char betu) {
        this.betu = betu;
        balNulla = null;
        jobbEgy = null;
    }

    ;

    public Csomopont nullasGyermek() {
        return balNulla;
    }

    public Csomopont egyenesGyermek() {
        return jobbEgy;
    }

    public void ujNullasGyermek(Csomopont gy) {
```

```
        balNulla = gy;
    }

    public void ujEgyesGyermeke(Csomopont gy) {
        jobbEgy = gy;
    }

    public char getBetu() {
        return betu;
    }

    private char betu;

    private Csomopont balNulla = null;
    private Csomopont jobbEgy = null;

};

private Csomopont fa = null;

private int melyseg, atlagosszeg, atlagdb;
private double szorasosszeg;

public void kiir(Csomopont elem, java.io.PrintWriter os) {

    if (elem != null) {
        ++melyseg;
        kiir(elem.egyesGyermeke(), os);

        for (int i = 0; i < melyseg; ++i) {
            os.print("---");
        }
        os.print(elem.getBetu());
        os.print("(");
        os.print(melyseg - 1);
        os.println(")");
        kiir(elem.nullasGyermeke(), os);
        --melyseg;
    }
}

protected Csomopont gyoker = new Csomopont('/');
int maxMelyseg;
double atlag, szoras;

public int getMelyseg() {
```

```
melyseg = maxMelyseg = 0;
rmelyseg(gyoker);
return maxMelyseg - 1;
}

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

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

    rszas(gyoker);

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

    return szoras;
}

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

        rmelyseg(elem.nullasGyermekek());
        --melyseg;
    }
}

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

```
    }  
  }  
}  
  
public void rszoras(Csomopont elem) {  
    if (elem != null) {  
        ++melyseg;  
        rszoras(elem.egyenesGyermeke());  
        rszoras(elem.nullasGyermeke());  
        --melyseg;  
        if (elem.egyenesGyermeke() == null && elem.nullasGyermeke() == null) {  
            ++atlagdb;  
            szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));  
        }  
    }  
}  
  
public static void usage() {  
    System.out.println("Usage: lzwtree in_file -o out_file");  
}  
  
public static void main(String args[]) {  
  
    if (args.length != 3) {  
  
        usage();  
  
        System.exit(-1);  
    }  
  
    String inFile = args[0];  
  
    if (!"-o".equals(args[1])) {  
        usage();  
        System.exit(-1);  
    }  
  
    try {  
  
        java.io.FileInputStream beFile =  
            new java.io.FileInputStream(new java.io.File(args[0]));  
  
        java.io.PrintWriter kiFile =  
            new java.io.PrintWriter(  
                new java.io.BufferedWriter(  
                    new java.io.FileWriter(args[2])));  
    }  
}
```

```
byte[] b = new byte[1];

LZWBinFa binFa = new LZWBinFa();

while (beFile.read(b) != -1) {
    if (b[0] == 0x0a) {
        break;
    }
}

boolean kommentben = false;

while (beFile.read(b) != -1) {

    if (b[0] == 0x3e) {
        kommentben = true;
        continue;
    }

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

    if (kommentben) {
        continue;
    }

    if (b[0] == 0x4e) // N betű
    {
        continue;
    }

    for (int i = 0; i < 8; ++i) {

        if ((b[0] & 0x80) != 0)
        {
            binFa.egyBitFeldolg('1');
        } else
        {
            binFa.egyBitFeldolg('0');
        }
        b[0] <<= 1;
    }
}
```

```
binFa.kiir(kiFile);

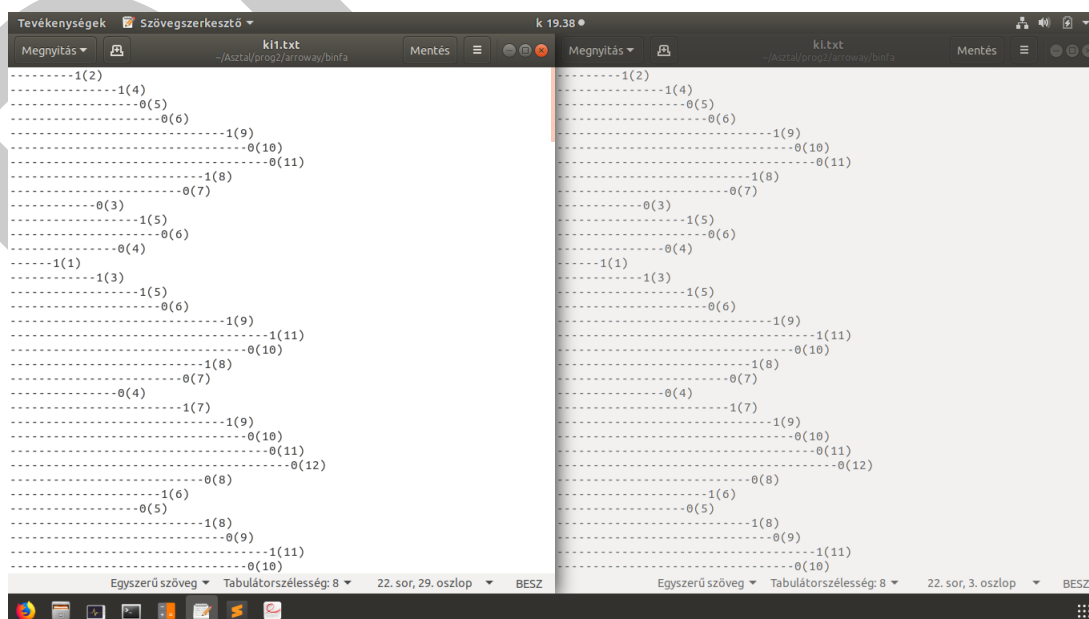
kiFile.println("depth = " + binFa.getMelyseg());
kiFile.println("mean = " + binFa.getAtlag());
kiFile.println("var = " + binFa.getSzoras());

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

} catch (java.io.FileNotFoundException fnfException) {
    fnfException.printStackTrace();
} catch (java.io.IOException ioException) {
    ioException.printStackTrace();
}

}
}
```

Amint látjuk nem sokban különbözik a kódunk a C++-os változathoz képest. Először is a pointerek illetve a referenciajelek tűntek el. Ez azért van mert a java nem támogatja a pointereket. Javában minden referencia szerint működik. Mikor egy osztályban létrejön egy objektum akkor a referenciáját kapjuk meg. Javában operátor túlterhelés sincs, ezért ezt egy függvénnyel helyettesítettük. Ezenkívül ebben a kódban nincs szükségünk destruktorra, ugyanis javában működik az automatikus szemétygyűjtő, ami automatikusan felszabadítja a nem használt memóriaterületet. Classon belül nincs külön public illetve private rész, ezért minden függvény és eljárás elé oda kell írunk, hogy public vagy private. Ezeken kívül még pár szintaktikai dolgot kellett átírni és minden működik úgy mint C++-ban. A képen látható, hogy ugyanaz a kimenet, tehát az átírás sikeres volt.



```
-----1(2)
-----1(4)
-----0(5)
-----0(6)
-----1(9)
-----0(10)
-----0(11)
-----1(8)
-----0(7)
-----0(3)
-----1(5)
-----0(6)
-----0(4)
-----1(1)
-----1(3)
-----1(5)
-----0(6)
-----1(9)
-----1(11)
-----0(10)
-----1(8)
-----0(7)
-----0(4)
-----1(7)
-----1(9)
-----0(10)
-----0(11)
-----0(12)
-----0(8)
-----1(6)
-----0(5)
-----1(8)
-----0(9)
-----1(11)
-----0(10)
```

A feladat második része, hogy létrehozzunk egy java servletet és a binfa a böngésző címsorából kapja meg a bemeneti adatokat és ezekből készítsen bináris fát. A java servlet egy olyan objektum ami HTTP kéréseket kap és készít. A feladat megoldásához szükség van az apache tomcat telepítésére. Ez a feladat még folyamatban....

11.3. Gagy

Az ismert formális2,,

```
while (x <= t && x >= t && t != x);
```

tesztkérdéstípusra adj a szokásosnál (miszerint x, t az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciája) „mélyebb” választ, írd Java példaprogramot mely egyszer végtelen ciklus, más x, t értékekkel meg nem! A példát építsd a JDK Integer.java forrására³, hogy a 128-nál inkluzív objektum példányokat poolozza!

JDK forrás:

```
public static Integer valueOf(int i) {  
    if (i >= IntegerCache.low && i <= IntegerCache.high)  
        return IntegerCache.cache[i + (-IntegerCache.low)];  
    return new Integer(i);  
}
```

A java eltárolja a -128 és 127 intervallumba tartozó integer típusú számokat. Ha egy ebben a tartományban található számot akarunk egy változóhoz rendelni, akkor meg fog hívódni az Integer.valueOf függvény és hozzá fog rendelni egy címet ami már alaphól benne volt az úgynevezett poolba, ahol ezek az előre elmentett számok vannak.

Azonban ha egy az intervallumon kívül eső számot rendelünk egy változóhoz akkor a függvény létre fog hozni egy teljesen új objektumot. A java az == és != operátorok esetén a címeket hasonlítja össze. Tehát ha két változóhoz ugyanazt az értéket rendeljük és ezek az értékek benne vannak a -128 és 127 intervallumba, akkor a két változónak meg fog egyezni a címük, tehát egy összehasonlításkor megegyeznek majd. Azonban ha ezen az intervallumon kívül választunk számot akkor két új objektum fog létrejönni és egy összehasonlítás során nem fog megegyezni a címük.

Most pedig nézzünk két programot, ahol a feladatban megadott feltételt megadva egyik esetben végtelen ciklust, másik esetben pedig hamis feltételt fogunk kapni.

```
class vegtelen{  
    public static void main(String args[])  
    {  
        Integer t = 130;  
        Integer x = 130;  
  
        while(x <= t && x>=t && t != x)  
            System.out.println("Vegtelen lesz");  
    }  
}
```


Látható,hogy ebben a kódban a t és x értékek egy olyan számot adtunk meg ami kívül esik az eltárolt intervallumon,ezáltal új objektuok jönnek létre,melyeknek nem ugyanaz lesz a címük.Így a while-ban lévő kifejezés mindig igaz lesz,így egy végtelen ciklust kapunk.

```
class igaz{
    public static void main(String args[])
    {
        Integer t = 100;
        Integer x = 100;

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

Ebben a kódban pedig láthatjuk,hogy t és x értékeként egy olyan számot adtunk amely benne van a pool-ba,így a java,ugyanazt a címet fogja mindkét változónak adni. Így a while ciklus feltétele hamis lesz,nem lép végtelen ciklusba.

11.4. Yoda

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

A programozásban a yoda conditions egy programozási stílus.Ezt összehasonlításakor használjuk.A megszokott feltétel úgy néz ki,hogy először a változót írjuk amit hasonlítani akarunk valamilyen konstanshoz azonban a yoda conditions esetében a konstansot fogjuk a kifejezés bal oldalára írni.A nevét a Star Wars-ban szereplő Yoda karakterről kapta,ugyanis ő sem a megszokott szintaxis szerint beszél a filmben. Ez a stílus az elírásokból következő hibákat próbálja orvosolni,ugyanis gyakran előfordul,hogy valamit össze-akarunk hasonlítani azonban csak egy egyenlőségjelet írunk kettő helyett ami által felülírjuk a változó értékét.Hátránya azonban,hogy a kód olvasása nehezebb lesz.

Egy nem yoda szerint írt kifejezés:

```
if (number == 14){do something}
//Itt látszik,hogy a változó van a kifejezés bal oldalára írva a ←
konstans pedig a jobb oldalon
```

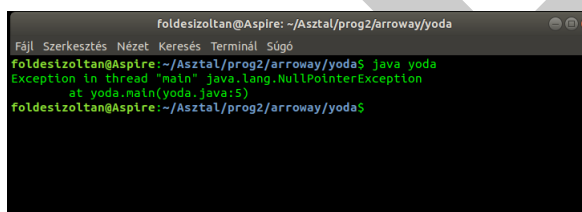
Ugyanez yoda szerint:

```
if (14 == number) {do something}
//Látszik,hogy a konstans a bal oldalra került míg a változó a jobb ←
oldalra.
```

A fenti példákból jól látható ha csak egy egyenlőségjelet írunk a kifejezésbe, akkor yoda condition nélkül a number változóhoz 14-et rendelne a program,azonban ha yoda condition szerint írunk egy egyenlőségjelet

akkor hibát dobna a fordítás során. Tehát a yoda stílussal kevesebb a hiba lehetőségünk. A yoda conditionnel elkerülhetőek a nem kívánatos null viselkedések is. Például ha egy string változónak null értéket adunk és ezt a stringet hasonlítjuk egy másik stringhez akkor nullpointerexceptiont okoz. Azonban ha yoda condition szerint először a stringet írjuk le és csak utána a változót akkor a kifejezés megfelelően fog működni. Erre írtunk egy programot ami egy nullpointerexceptiont dob:

```
class yoda {  
  
    public static void main(String args[]){  
        String mine = null;  
        if ( mine.equals("yoda")){ System.out.println("nem fog működni");  
    }  
}  
}
```



11.5. Kódolás from scratch

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

Ebben a feladatban a BBP algoritmust fogjuk megírni. A Bailey-Borwein-Plouffe vagy röviden BBP féle algoritmus 1995-ben lett felfedezve. Ezzel az algoritmussal a Pi egy általunk megadott számjegyétől tudunk további jegyeit számolni hexadecimális formában. Magát az algoritmust is egy gép fedezte fel.

A kód elején létrehozunk a "d16PiHexaJegyek" változót amiben a hexadecimális jegyeket fogjuk tárolni. A kódban négy darab függvény fogja a számításokat végezni. Nézzük őket egyenként.

```
public PiBBP(int d) {  
  
    double d16Pi = 0.0d;  
  
    double d16S1t = d16Sj(d, 1);  
    double d16S4t = d16Sj(d, 4);  
    double d16S5t = d16Sj(d, 5);  
    double d16S6t = d16Sj(d, 6);  
  
    d16Pi = 4.0d*d16S1t - 2.0d*d16S4t - d16S5t - d16S6t;  
  
    d16Pi = d16Pi - StrictMath.floor(d16Pi);  
}
```

```

StringBuffer sb = new StringBuffer();

Character hexaJegyek[] = {'A', 'B', 'C', 'D', 'E', 'F'};

while(d16Pi != 0.0d) {

    int jegy = (int)StrictMath.floor(16.0d*d16Pi);

    if(jegy<10)
        sb.append(jegy);
    else
        sb.append(hexaJegyek[jegy-10]);

    d16Pi = (16.0d*d16Pi) - StrictMath.floor(16.0d*d16Pi);
}

d16PiHexaJegyek = sb.toString();
}

```

Ez a függvény paraméterül kap egy integert, ettől az integer+1-től fogjuk számolni a jegyeket. Ezután négy double változót hozunk létre, ezek szükségesek a képlethez, ezeket majd a d16Sj függvény fogja kiszámolni. Majd következik maga a képlet. Ennek az eredménynek az egész részét fogjuk kapni a StrictMath.floor-nak köszönhetően, ugyanis ez egy nem egész számnak a nála kisebb egész részét fogja visszaadni. Ezekután megadjuk a hexajegyeket majd a kapott értékeket átváltjuk, és az sb.append függvény segítségével összefűzzük őket.

```

public double d16Sj(int d, int j) {

    double d16Sj = 0.0d;

    for(int k=0; k<=d; ++k)
        d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);

    /* (bekapcsolva a sorozat elején az első utáni jegyekben növeli pl.
       a pontosságot.)
    for(int k=d+1; k<=2*d; ++k)
        d16Sj += StrictMath.pow(16.0d, d-k) / (double)(8*k + j);
    */

    return d16Sj - StrictMath.floor(d16Sj);
}

```

Ez a második függvény a programban, ez hívódik meg az előző függvényben és ez fogja számolni az S1 S4 S5 és S6 értékeket. A függvény két értéket kap paraméterül. A for ciklus után itt is történik egy másik függvényhívás, az n16modk függvény fog meghívódni. A függvény ennek az értékeknek is az egész részét fogja visszaadni.

```

public long n16modk(int n, int k) {

```

```
int t = 1;
while(t <= n)
    t *= 2;

long r = 1;

while(true) {

    if(n >= t) {
        r = (16*r) % k;
        n = n - t;
    }

    t = t/2;

    if(t < 1)
        break;

    r = (r*r) % k;

}

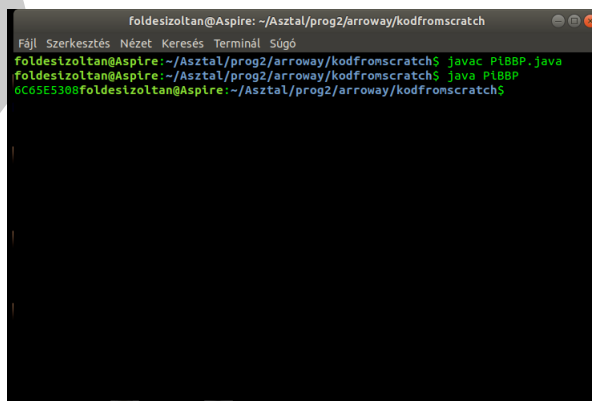
return r;
}
```

Ez a függvény bináris hatványozást fog végezni. Ez $16^n \bmod k$ -t fogja kiszámolni. Ezután a toString függvény fogja visszaadni a kiszámolt hexajegyeket.

```
public static void main(String args[]) {
    System.out.print(new PiBBP(1000000));
}
}
```

Itt látható a main függvény ahol példányosítjuk a PiBBP osztályt. A 1 millióodik számjegytől fogjuk kiírni a Pi-t, vagyis az 1 millió+1 től fogja kiírni a hexadecimális számokat.

Eredmény:



```
foldesizoltan@Aspire: ~/Asztal/prog2/arroway/kodfromscratch
foldesizoltan@Aspire:~/Asztal/prog2/arroway/kodfromscratch$ javac PiBBP.java
foldesizoltan@Aspire:~/Asztal/prog2/arroway/kodfromscratch$ java PiBBP
0C05E5308f
```

12. fejezet

„Helló, Berners-Lee!”

12.1. Python

A könyv megadott oldalain a Python nyelv megismerésével foglalkozunk. A Python egy magasszintű, dinamikus, objektumorientált és platformfüggetlen nyelv. Leginkább prototípus készítésére és különböző tesztelésekre használják. A Python egy köztes nyelv, azaz nincs szükség fordításra, ezért használják előszeretettel tesztelésekre, ugyanis kevesebb időt vesz igénybe. Ettől függetlenül nagyon bonyolult problémákat lehet leírni vele. Sokkal rövidebb kódok hozhatóak létre ugyanarra a problémára mint például C++ vagy Java nyelveken. A kódot sorbehúzással szerkesztjük, nincs szükség zárójelekre vagy kulcsszavakra. Egy utasítás a sor végéig tart, nincs szükség pontosvesszőre. A behúzást tabulátorral vagy szóközökkel is létrehozhatjuk, azonban ezekkel konzisztensnek kell lennünk, mert megzavarhatjuk az értelmezést. Az értelmező a kódot soronkénti tokenekre bontja. Ezek a tokenek lehetnek azonosítók, kulcsszavak, operátorok, delimiter-ek vagy literálok. Ezek után a könyvben egy táblázatot láthatunk ami felsorolja a nyelvben lefoglalt kulcsszavakat. Pythonban az összes adatot objektumok ábrázolnak és a rajtuk végezhető műveletek függnek az adott objektumok típusától. Nincs szükség megadni az adatok típusát ezt a rendszer futási időben kitalálja az adott adathoz rendelt értékek alapján. A számoknak van egész, lebegőpontos illetve komplex ábrázolása is. Az ennesek vagy angolul tuples objektumok gyűjteményei. Ezeknek az objektumoknak nem muszáj ugyanolyan típusúaknak lenniük. Ezeket sima zárójelek közé írjuk. A szögletes zárójelekkel megadott dolgo pedig a lista. Ebben különböző típusú elemek lehetnek. Kapcsos zárójelekkel a szótárakat adhatjuk meg. Ez egy rendezetlen halmaz. Pythonban a NULL neve None. A változók az egyes objektumokra mutató referenciák. Ha egy változó már más objektumra mutat akkor a az automatik garbage collector hívódik meg. Ez a felesleges memóriaterületet fogja felszabadítani.

12.2. C++, Java

Ebben az olvasónaplóban a megadott két könyv szerint fogjuk összehasonlítani a Java és a C++ programozási nyelveket. A két nyelv szintaxisa nagyon hasonló. Az objektumorientált fogalmakkal kezdjük. Javában az objektumok a legkisebb önálló egységek, tehát javában minden objektum. C++-ban az objektumok adatstruktúrák. Az osztályon belül az adott dolog összes jellemzője megtalálható. Szinte minden leírható egy osztállyal pl. bankszámla, vállalat dolgozói stb. Az osztályokat lehet példányosítani, egy adott példány vagy egyed rendelkezni fog az adott osztály tulajdonságaival. Ezeket az egyedeket vagy példányokat objektumoknak is nevezzük. Javában az osztályokat egy két részből álló osztálydefiníció írja le. Az

első rész azokat a változókat tartalmazza mellyel az objektum állapota írható le. A második rész pedig az osztályon elvégezhető metódusokat tartalmazza. Mind javában és C++-ban írhatunk konstruktorokat ezek automatikusan meghívódnak ha létrehozunk egy új egyedet. A konstruktor nevének ugyanannak kell lennie mint magának az osztálynak. Olyan mint egy metódus, viszont nincs visszatérítési értéke. Ha egy objektum törlődik, akkor meghívódik az adott osztály destruktora, ez a memóriát fogja felszabadítani. Ezt C++-ban nekünk kell megírni, azonban javában van egy automatikus szemétygyűjtő mechanizmus. Azonban vannak olyan esetek ahol tudnunk kell, hogy egy objektum megsemmisül, ezt a javában a `finalize` metódus adja meg. Ez akkor hívódik meg mielőtt még a szemétygyűjtő felszabadítaná a memóriát. Az osztályváltozók olyan változók amelyek magához az osztályhoz tartoznak. Ezek az osztályváltozók az összes példány osztozik. Az objektumok példányosítással jönnek létre, ezt a `new` operátorral tudjuk elvégezni. Az operátor mögé megadjuk, hogy melyik osztályt szeretnénk példányosítani és mögé zárójelek közé megadjuk az adott példány paramétereit. Azonban a paraméter szerinti példányosítás csak akkor tud működni ha saját konstruktort hoztunk létre. Ez a konstruktor a megfelelő metódusokat meghívva beállítja a megfelelő értékeket. Javában létezik az öröklődés aminek legegyszerűbb este amikor egy osztályt egy másik létező osztály kiterjesztésével definiálunk. Ez a kiterjesztés több dolgot is jelenthet pl. új műveletek vagy új változók bevezetését. Az osztályban ezt az `extends` szóval jelezzük. Ebben az esetben az alap osztály lesz a szülőosztály a bővített pedig a gyermekosztály. Így létrejönnek leszármazottak illetve ősök. A bővített osztály rendelkezni fog az újonnan megadott tagokkal illetve a szülő tagjaival is, tehát örökli a tagokat (innen származik a kifejezés). A gyermek azonban a szülő konstruktorait nem örökli.

IV. rész

Irodalomjegyzék

DRAFT

12.3. Általános

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

12.4. C

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

12.5. C++

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

12.6. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf , 2004.

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPROG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEACHackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésükért és hasznos észrevételeikért.

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