

Univerzális programozás

**Tanulj meg programozni 2^{23}
másodperc alatt!**

Ed. 1. kiadás

KÖZREMÜKÖDTEK

	<i>CÍM :</i>	
	Univerzális programozás	
<i>HOZZÁJÁRULÁS</i>	<i>NÉV</i>	<i>DÁTUM</i>
ÍRTA	Bátfai Norbert és Molnár Antal Albert	2019. április 25.

VERZIÓTÖRTÉNET

VERZIÓ	DÁTUM	LEÍRÁS	NÉV
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	2019-02-18	Perszonalizáció. Saját fork GitHubon: https://github.com/-krook1024/textbook	krook1024
0.0.5	2019-02-18	Meglévő kódsnittek behúzása. Magyarázatok megírásának elkezdése.	krook1024

VERZIÓTÖRTÉNET

VERZIÓ	DÁTUM	LEÍRÁS	NÉV
0.0.6	2019-02-20	Egy plusz olvasandó doksi (The Unix Koans of Master Foo). Magyarázat a legtöbb első csokorbeli feladathoz.	krook1024
0.0.7	2019-02-26	Az első csokor teljesen kész.	krook1024
0.0.8	2019-02-26	A második csokor elkezdve.	krook1024
0.0.9	2019-02-26	A második csokor nagyjából készen van.	krook1024
0.1.0	2019-03-07	Munkálkodás a második csokron. Harmadik csokor elkezdése.	krook1024
0.1.1	2019-03-11	Harmadik csokor befejezve. Negyedik csokor elkezdése.	krook1024
0.1.9	2019-03-15	Caesar csokor kész. Mandelbrot elkezdve.	krook1024
0.2.1	2019-03-23	Travis build beállítva. Welch elkezdve.	krook1024

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. A Battlestation	3
II. Tematikus feladatok	4
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	7
2.3. Változók értékének felcserélése	9
2.4. Labdapattogás	11
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	13
2.6. Helló, Google!	13
2.7. 100 éves a Brun tétele	15
2.8. A Monty Hall probléma	18
3. Helló, Chomsky!	19
3.1. Decimálisból unárisba átváltó Turing gép	19
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	20
3.3. Hivatkozási nyelv	21
3.4. Saját lexikális elemző	23
3.5. l33t.l	25
3.6. A források olvasása	26
3.7. Logikus	29
3.8. Deklaráció	30

4. Helló, Caesar!	34
4.1. double ** háromszögmátrix	34
4.2. C EXOR titkosító	35
4.3. Java EXOR titkosító	36
4.4. C EXOR törő	37
4.5. Neurális OR, AND és EXOR kapu	38
4.6. Hiba-visszaterjesztéses perceptron	42
5. Helló, Mandelbrot!	44
5.1. A Mandelbrot halmaz	44
5.2. A Mandelbrot halmaz a std::complex osztállyal	45
5.3. Biomorfok	46
5.4. A Mandelbrot halmaz CUDA megvalósítása	47
5.5. Mandelbrot nagyító és utazó C++ nyelven	52
5.6. Mandelbrot nagyító és utazó Java nyelven	52
6. Helló, Welch!	57
6.1. Első osztályom	57
6.2. LZW	58
6.3. Fabejárás	60
6.4. Tag a gyökér	61
6.5. Mutató a gyökér	61
6.6. Mozgató szemantika	62
7. Helló, Conway!	66
7.1. Hangyszimulációk	66
7.2. Java életjáték	69
7.3. Qt C++ életjáték	77
7.4. BrainB Benchmark	78
8. Helló, Schwarzenegger!	80
8.1. Szoftmax Py MNIST	80
8.2. Mély MNIST	83
8.3. Deep dream	83
8.4. Minecraft-MALMÖ	86

9. Helló, Chaitin!	87
9.1. Iteratív és rekurzív faktoriális Lisp-ben	87
9.2. Weizenbaum Eliza programja	88
9.3. Gimp Scheme Script-fu: króm effekt	88
9.4. Gimp Scheme Script-fu: név mandala	89
10Helló, Guttenberg!	90
10.1Juhász István – Magas szintű programozási nyelvek	90
10.2Kernighan & Ritchie – A C programozási nyelv	92
10.3BME könyv	94
III. Második felvonás	95
11Helló, Arroway!	96
11.1A BPP algoritmus Java megvalósítása	96
11.2Java osztályok a Pi-ben	96
IV. Irodalomjegyzék	97
11.3Általános	98
11.4C	98
11.5C++	98
11.6Lisp	98

Ábrák jegyzéke

2.1. A B_2 konstans közelítése	17
3.1. Példa ilyen Turing-gépre	19
3.2. Lexikális elemző	23
3.3. LaTeX forrás lefordítva	30
4.1. Neurális OR kapu	39
4.2. Neurális OR, AND kapu	40
4.3. Neurális EXOR kapu	41
4.4. Neurális XOR kapu	42
5.1. A Mandelbrot halmaz a komplex síkon	45
5.2. A generált biomorf	47
5.3. Mandelbrot halmaz CUDA-val	51
5.4. Mandelbrot nagyító és utazó Java nyelven	56
6.1. A 011010 bináris fája	59
7.1. Hangyszimulációk futás közben	67
7.2. Hangyszimulációk UML osztálydiagram	68
7.3. Qt C++ életjáték	77
7.4. Qt C++ életjáték	78
7.5. BrainB Benchmark	78
8.1. 2*2 számítási ábra	81
8.2. Példák az MNIST adatbázisból	81

Előszó

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.

A könyv írása során igyekeztem minden forrást beágyazni, annak érdekében, hogy ne kelljen folyamatosan új ablakokat megnyitni a böngészőben.

Milyen nyelven nyomjuk?

- C (mutatók)
- C++ (másoló és mozgató szemantika)
- Java (lebutított C++)
- R (vektoros szemlélet)
- Python
- Lisp, Prolog (lássunk illet is)

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

A programozás problémamegoldás. Sokféle és sokszínű. Trial and error. És abból nagyon sok.

A programozás ötlete vonzónak tűnik, de nem való mindenkinél. Amennyiben számodra jól hangzik ugyan azt a kódcsipetet bámulni óráig, hibákat javítani, akkor a legjobb helyen jársz. Ellenkező esetben előfordulhat, hogy a programozás (és ez a könyv) mégsem neked való.

Elméletben a programozók szigorú, lefektetett szabályok és nyelvtanok alapján alkotnak kódot. Ezzel szemben a valóságban ők is csak emberek. Rengeteg időt töltnek saját, vagy mások hibáinak javításával, tanulással, stb. Értetlenül ülnek a számítógép felett, mikor három napnyi hibakeresés után még mindig nem találtak megoldást, vagy éppen mikor az általuk hibásnak feltételezett kód (látszólag) jól működik.

1.2. Milyen doksikat olvassak el?

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



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.

1.3. A Battlestation

A könyv egy ThinkPad X230 ultrabookon íródott, Arch Linux alatt. C fordító: gcc, szövegszerkesztő: vim. Ez a setup megengedte számomra, hogy soha ne kelljen el-hagynom a terminált és egy kellemes, a saját igényeimnek megfelelően kialakított környezetben dolgozhassak.

Az Arch Linux egy bleeding-edge operációs rendszer, melynek használata nagyon egyszerű, és hosszútávon rengeteg időt spórol meg, amennyiben elsajátítjuk. A bleeding-edge azt jelenti, hogy a fejlesztők igyekeznek minden csomagból a legfrissebbet elérhetővé tenni a felhasználók számára a lehető leggyorsabban.

Az Arch Linux sok ablakkezelőt támogat, az én választásom pedig az i3-ra esett. Ez egy billentyűzet-orientált ablakkezelő rendszer, melynek előnye, hogy minden teljesen testreszabható. Az én konfigurációm ról beágyazok egy képet lentebb.

```

b1@desktop ~/Documents
[1] 0 [empty]
b1@desktop ~% ls -lh
total 5.0M
drwxr-xr-x 2 b1 wheel 0 6.17M sum, 311G free 1/11 All

1 15.8% 4 10.7%
2 20.9% 5 14.9%
3 20.8% 6 14.3%
Mem[2.03G/7.76G] Tasks: 66, 293 thr; 1 running
Swp[0/0K] Load average: 1.25 1.04 0.89
Uptime: 13:33:00

PID USER PRIO NI VIRT RES SHR S CPU% MEM% TIME+ COMMAND
2100 b1 20 0 14.0M 75620 51768 S 32.3 0.9 31:59.07 /usr/lib/
18824 b1 20 0 12.2M 31640 22908 S 22.2 0.4 0:01.80 main ->s p
28220 b1 20 0 1856M 329M 143M S 12.8 4.1 13:09.94 /usr/lib/
29882 b1 20 0 2449M 535M 182M S 10.1 6.7 50:30.19 /usr/lib/
23903 b1 20 0 2482M 511M 169M S 5.0 6.4 52:37.56 /usr/lib/
1657 b1 20 0 24716 432M 2668S 4.0 0.5 0:00.00 /usr/lib/python-
18223 b1 20 0 24716 13624 8280 S 1.0 0.2 0:02.15 et.
18456 b1 20 0 526M 692 624 S 2.7 0.0 0:01.27 lolcat
725 b1 9 -1 1064M 13272 9720 S 2.0 0.2 7:14:12 /usr/bin/
29887 b1 20 0 2449M 535M 182M S 6.7 1:26.35 /usr/lib/
5308 b1 20 0 1856M 329M 143M S 1.3 4.1 0:45.24 /usr/lib/
18793 b1 20 0 1856M 329M 143M S 1.3 4.1 0:00.11 /usr/lib/
787 b1 20 0 1856M 329M 143M S 0.2 2:33.44 /usr/lib/
18797 b1 20 0 1856M 329M 143M S 0.7 1.1 0:00.12 /usr/lib/
18455 b1 20 0 3152 244M 1908 S 0.7 0.0 0:00.08 cmatrix
10792 b1 20 0 1856M 329M 143M S 0.7 4.1 0:00.12 /usr/lib/
F1Help F2Setup F3Search F4Allterm F5Tree F6SortbyP F7Nice F8Kill

F1Help F2Setup F3Search F4Allterm F5Tree F6SortbyP F7Nice F8Kill

15 import javafx.application.Application;
14 import javafx.scene.Scene;
13 import javafx.scene.control.Label;
12 import javafx.scene.layout.StackPane;
11 import javafx.stage.Stage;
10 ...
9 public class HelloFX extends Application {
8 ...
7 @Override
6 public void start(Stage stage) {
5 String javaVersion = System.getProperty("java.version");
4 String javafxVersion = System.getProperty("javafx.version");
3 Label l = new Label("Hello, JavaFX " + javafxVersion + ", running on Java " + javaVersion + ".");
2 Scene scene = new Scene(new StackPane(l), 640, 480);
1 stage.setScene(scene);
stage.show();
}
1 ...
2 ...
3 ...
4 ...
5 ...
6 ...
7 }

NORMAL HelloFX.java
java utf-8[unx] 69% 16/23 la : 1

~ + neofetch
-----+
OS: Arch Linux x86_64
Kernel: 5.0.4-1-MANJARIS-ARCH
Uptime: 13 hours, 33 mins
Packages: 679 (pacman)
Shell: bash 5.0.2
Resolution: 1920x1080, 1920x1080
WM: i3
Theme: Adwaita [GTK2/3]
Icontheme: Adwaita [GTK2/3]
Terminal: st
Terminal Font: mono
CPU: AMD FX-6300 (6) @ 3.800GHz
GPU: NVIDIA GeForce GTX 660
Memory: 2060MiB / 7949MiB
F V J N S : & & v 1 3 3 4 g l o m 9 x M 7 k

```

II. rész

Tematikus feladatok

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus



Tutorált

Ebben a feladatban tutoráltam Füleky Ladislavot.

Dolgoztassunk 1 szálat megközelítőleg 0 százalékban:

```
#include <unistd.h> // sleep()

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

    return 0;
}
```

Amennyiben egy végtelen ciklusban `sleep(1)` rendszerhívással jelezük az OS számára, hogy a következő 1 milliszekundumban nem kívánunk processzor időt kapni, a processzor használatunk (megközelítőleg) 0 százalék lesz.

Dolgoztassunk 100 százalékban egy szálat:

```
#define EVER ;;

int main()
{
    for (EVER) ;
    return 0;
}
```

Termézeszen itt semmi extra, egy végtelen ciklussal elérhetjük a kívánt hatást.

Dolgoztassuk meg az összes szálat 100 százalékban:

```
#include <pthread.h>
#include <unistd.h> // sleep()

#define NO_OF_THREADS 4

void *func()
{
    for (;;) ;
}

int main()
{
    pthread_t thread[NO_OF_THREADS];

    for (int i = 0; i < NO_OF_THREADS; i++)
        pthread_create(&thread[i], NULL, func, NULL);

    for (;;)
        sleep(1);

    return 0;
}
```

Ebben az esetben a `pthreads.h` adta előnyöknél fogva, és bízva abban, hogy thread-safe függvényt sikerült írnunk, benne a végtelen ciklussal, egyszerűen megoldhatjuk, hogy az alkalmazásunk magához vegye a fennmaradó szabad processzor idő 100 százalékát.

Egy másik megoldásban az OpenMP függvénykönyvtárat használva is megoldható a feladat, lényegesen kevesebb, minden össze 5 sornyi kóddal.

```
#include <omp.h>

int main()
{
#pragma omp parallel
    for (;;) ;
    return 0;
}
```

A különbség a kettő megoldás között az, hogy a `pthreads` egy alacsony szintű POSIX API, melyben jóval több irányításunk van a szálak fölött, míg az OMP egy magasabb szintű könyvtár mely szinte minden megcsinál helyettünk, ezzel kivéve az irányítást a kezünkön.

A feladat érdekességét számomra az adta, hogy C-ben egy üres ciklusmag nagyság-rendekekkel több processzor időt igényel, mint egy olyan ami tegyük fel `sleep(1)`-t tartalmaz. Meglehetősen kíváncsi voltam hogy mi is történik ilyenkor a háttérben.

Némi utánajárás után meg is találtam a választ, mi szerint az történik, hogy míg egy üres ciklusmag esetén az operációs rendszer azt hiszi, hogy rengeteg dolga van a programnak, így ki is osztja neki az összes rendelkezésre álló processzor időt. Ezzel szemben a `sleep(1)` jelzi az operációs rendszer számára, hogy a következő 1 millisekundumban nem tart igényt processzor időre, ennek megfelelően az OS nem is oszt számára időt.

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 pszeudokódjára:

```
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-est, amelyben a Lefagy-ra épülő Lefagy2 már nem tartalmaz feltételezett, csak csak konkrét kódot:

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

Mit fog 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.

A lefagyó program problémája majdnem egyidős a programozással. Annak idején Alan Turing (aki megalkotta a Turing-gép fogalmát) matematikai eszközökkel keresztül bizonyította meg, hogy ilyen programot nem lehet írni. Míg egyszerű programok esetében ránézésére meg lehet mondani, hogy le fog-e fagyni, bonyolultabb programok esetében ezt eldönteneti is nehezebb.

Tehát amennyiben a bizonyítás ellenére nekünk sikerülne egy ilyen programot írnunk, valószínű, hogy nem kellene dolgoznunk soha életünkben, hiszen rengeteg pénzt érne. Egy ilyen program tömérdek problémát megoldana, amelyekért szép summát kaphatnánk.

2.3. Változók értékének felcserélése

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

Megoldás forrása:

```
#include <stdio.h>

void swap(int *a, int *b)
{
    *a -= *b;
    *b += *a;
    *a = *b - *a;
}

int main()
{
    int x = 3, y = 7;

    printf("x=%d y=%d\n", x, y);
    swap(&x, &y);
    printf("x=%d y=%d\n", x, y);

    return 0;
}
```

Logikai utasítás nélkül például összeadással oldható meg két változó cseréje. A fenti programot kipróbálva láthatjuk, hogy a programunk működik.

```
$ ./a.out
x=3 y=7
x=7 y=3
```

```
#include <stdio.h>

void swap(int *a, int *b)
{
    if (a != b) {
        *a ^= *b;
        *b ^= *a;
        *a ^= *b;
    }
}

int main()
{
    int x = 3, y = 7;

    printf("x=%d y=%d\n", x, y);
```

```
    swap(&x, &y);
    printf("x=%d y=%d\n", x, y);

    return 0;
}
```

Két változó értékét megcserélhetjük még az *XOR* operátorral, segédváltozó használata nélkül. Az *XOR* operátor bitenként végzi el a kizárá vagy logikai műveletet a bemeneteken. A fentebbi kód értelmet nyer, hogyha megnézzük, hogy mi történik a memóriában. Ehhez szerkesszük egy kicsit a programunkat, a következőképp: (ne ijedjünk meg a sok `#define`-tól, ezeket makrónak hívják, és ahhoz kellenek, hogy a megértést segítendő kettés számrendszerben láthassuk a számokat)

```
#include <stdio.h>

#define BYTE_TO_BINARY_PATTERN "%c%c%c%c%c%c%c%c"
#define BYTE_TO_BINARY(byte) \
    (byte & 0x80 ? '1' : '0'), \
    (byte & 0x40 ? '1' : '0'), \
    (byte & 0x20 ? '1' : '0'), \
    (byte & 0x10 ? '1' : '0'), \
    (byte & 0x08 ? '1' : '0'), \
    (byte & 0x04 ? '1' : '0'), \
    (byte & 0x02 ? '1' : '0'), \
    (byte & 0x01 ? '1' : '0')

void swap(int *a, int *b)
{
    if (a != b) {
        printf("a=" BYTE_TO_BINARY_PATTERN ", b="
            BYTE_TO_BINARY_PATTERN "\n",
            BYTE_TO_BINARY(*a), BYTE_TO_BINARY(*b));

        *a ^= *b;
        printf("a=" BYTE_TO_BINARY_PATTERN ", b="
            BYTE_TO_BINARY_PATTERN "\n",
            BYTE_TO_BINARY(*a), BYTE_TO_BINARY(*b));

        *b ^= *a;
        printf("a=" BYTE_TO_BINARY_PATTERN ", b="
            BYTE_TO_BINARY_PATTERN "\n",
            BYTE_TO_BINARY(*a), BYTE_TO_BINARY(*b));

        *a ^= *b;
        printf("a=" BYTE_TO_BINARY_PATTERN ", b="
            BYTE_TO_BINARY_PATTERN "\n",
            BYTE_TO_BINARY(*a), BYTE_TO_BINARY(*b));
    }
}
```

```
int main()
{
    int x = 3, y = 7;

    swap(&x, &y);

    //printf ("x=%d y=%d\n", x, y);
    return 0;
}
```

Erre a kimenet a következőképp alakul:

```
a=00000011, b=00000111
a=00000100, b=00000111
a=00000100, b=00000011
a=00000111, b=00000011
```

Ebben a formában könnyebben nyomon követhető, hogy mi történik a háttérben. Az első kimeneti sor a kiinduló állapotot tükrözi. A második sorban már a kizárt vagy eredménye jelenik meg. Ezek után még egyszer elvégezzük ezt a lépést, csak fordított sorrendben, aztán még egyszer az eredeti sorrendben. Ekkor az eredeti *a* és az eredeti *b* helyet cserélnek.

2.4. Labdapattogás

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

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

int main()
{
    int xj = 0, xk = 0, yj = 0, yk = 0, mx = 0, my = 0, h = 0, w =
    0;

    initscr();
    cbreak();
    noecho();
    nodelay(stdscr, TRUE);
    getmaxyx(stdscr, h, w);
    mx = w * 2;
    my = h * 2;

    while (true) {
        xj = (xj - 1) % mx;
        xk = (xk + 1) % mx;
```

```
yj = (yj - 1) % my;
yk = (yk + 1) % my;

clear();

mvprintw(abs((yj + (my - yk)) / 2),
         abs((xj + (mx - xk)) / 2), "o");

refresh();
usleep(30000);
}

return 0;
}
```

Ez a példa minta példája lehetne a programozás és a matematika felbonthatatlan kapcsolatának. Itt a lényeg, hogy olyan függvényt találunk, ami leírja a labda patto-gását, de mégsem vesz fel olyan értékeket, amik kívül esnek a konzolunkon (azaz a megjeleníthető koordinátákon).

A fenti kód egy megoldása a problémának. Itt az `ncurses` függvény könyvtárat vesszük használatba, mely terminálos interfészek (TUI-k) létrehozására lett elkészítve.

A program lényeges része a ciklusmagban van, ezt nézzük meg sorról sorra.

```
xj = (xj - 1) % mx;
```

Az `xj` változó értékét beállítjuk úgy, hogy az a maradéka legyen a saját értékénél egyel kisebb értéknek és a terminál szélességének a kétszeresének a hányadosával.

```
xk = (xk + 1) % mx;
```

Ugyanezt elvégezzük az `xk` változóval is.

```
yj = (yj - 1) % my;
yk = (yk + 1) % my;
```

Itt ugyanezt elvégezzük az `y` tengelyen is.

```
clear();
```

Letöröljük a képernyőt, hogy ne maradjon fent a labdánk az előző kirajzolásból adódóan. Ezzel készülünk fel a következő képkocka kirajzolására.

```
mvprintw(abs((yj + (my - yk)) / 2),  
abs((xj + (mx - xk)) / 2), "o");
```

Itt rajzoljuk ki a labdát magát, ehhez meghívjuk az `mvprintw` függvényt. Ez a függvény kiírat egy karaktert a megadott x és y koordinátáknak megfelelően a képernyőn valahol.

```
refresh();  
usleep(30000);
```

A `refresh` függvény szükséges, hogy kimenetet lássunk a terminál ablakunkban, majd várunk 30000 mikroszekundumnyi időt.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

Megoldás videó:

```
#include <stdio.h>  
  
int main()  
{  
    unsigned long int a = 1, count = 0;  
  
    do  
        count++;  
    while (a <= 1);  
  
    printf("A szóhossz ezen a számítógépen: %d.\n", count);  
    return 0;  
}
```

A szóhossz megállapítására használhatunk egy `int` típusú változót, melyet inicializálunk 1-el, majd addig shifteljük balra, amíg csak tudjuk. Ha megszámoljuk, hogy hányszor tudtuk balra shiftelni, megkapjuk a szóhosszt az adott számítógépen.

2.6. Helló, Google!



Tutorált

Ebben a feladatban tutorált Duszka Ákos Attila.

Megoldás videó:

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

void kiir(double tomb[], int db);
double tavolsag(double pagerank[], double pagerank_temp[], int db);

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 };

    for (;;) {
        for (int i = 0; i < 4; i++)
            PR[i] = PRv[i];

        for (int i = 0; i < 4; i++) {
            double tmp = 0.0;

            for (int j = 0; j < 4; j++) {
                tmp += L[i][j] * PR[j];
                PRv[i] = tmp;
            }
        }

        if (tavolsag(PR, PRv, 4) < 0.000001)
            break;
    }

    kiir(PR, 4);

    return 0;
}

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

```
double tavolsag(double pagerank[], double pagerank_temp[], int db)
{
    double tav = 0.0;

    for (int i = 0; i < db; i++) {
        tav += (pagerank[i] - pagerank_temp[i]) * (pagerank[i] -
            pagerank_temp
            [i]);
    }

    return sqrt(tav);
}
```

Ezt az algoritmust a Googlenél fejlesztette ki Larry Page (innen a neve; PageRank) és Sergey Brin. Vélték felfedezni, hogy minél több oldal mutat egy oldalra, annak az oldalnak az értéke annál nagyobb. Persze ez nem ilyen egyszerű, a nagyobb értékű weboldalak "voksaik" többet érnek. Egy oldal fontossága az alapján dől el, hogy hány másik oldal mutat ő rá.

Természetesen a Googlenél ennek az algoritmusnak egy jócskán módosított változata fut, mely nem is publikus, hiszen akkor mindenki kereső szolgáltatást indíthatna szabadidejében. Azt is fontos megjegyezni, hogy a fent látott példa az algoritmus egyszerűsített változata, ez már magában hordoz hibákat, és némi eltérést. Például a zsákutca hiba, miszerint zsákutcába érkezünk, amennyiben olyan oldalt találunk, amely nem mutat másikra.

2.7. 100 éves a Brun téTEL

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

```
# Copyright (C) 2019 Dr. Norbert Bátfai, nbatfai@gmail.com
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.

library(matlab)

stp <- function(x){
  primes = primes(x)
  diff = primes[2:length(primes)]-primes[1:length(primes)-1]
  idx = which(diff==2)
  t1primes = primes[idx]
  t2primes = primes[idx]+2
  rt1plust2 = 1/t1primes+1/t2primes
```

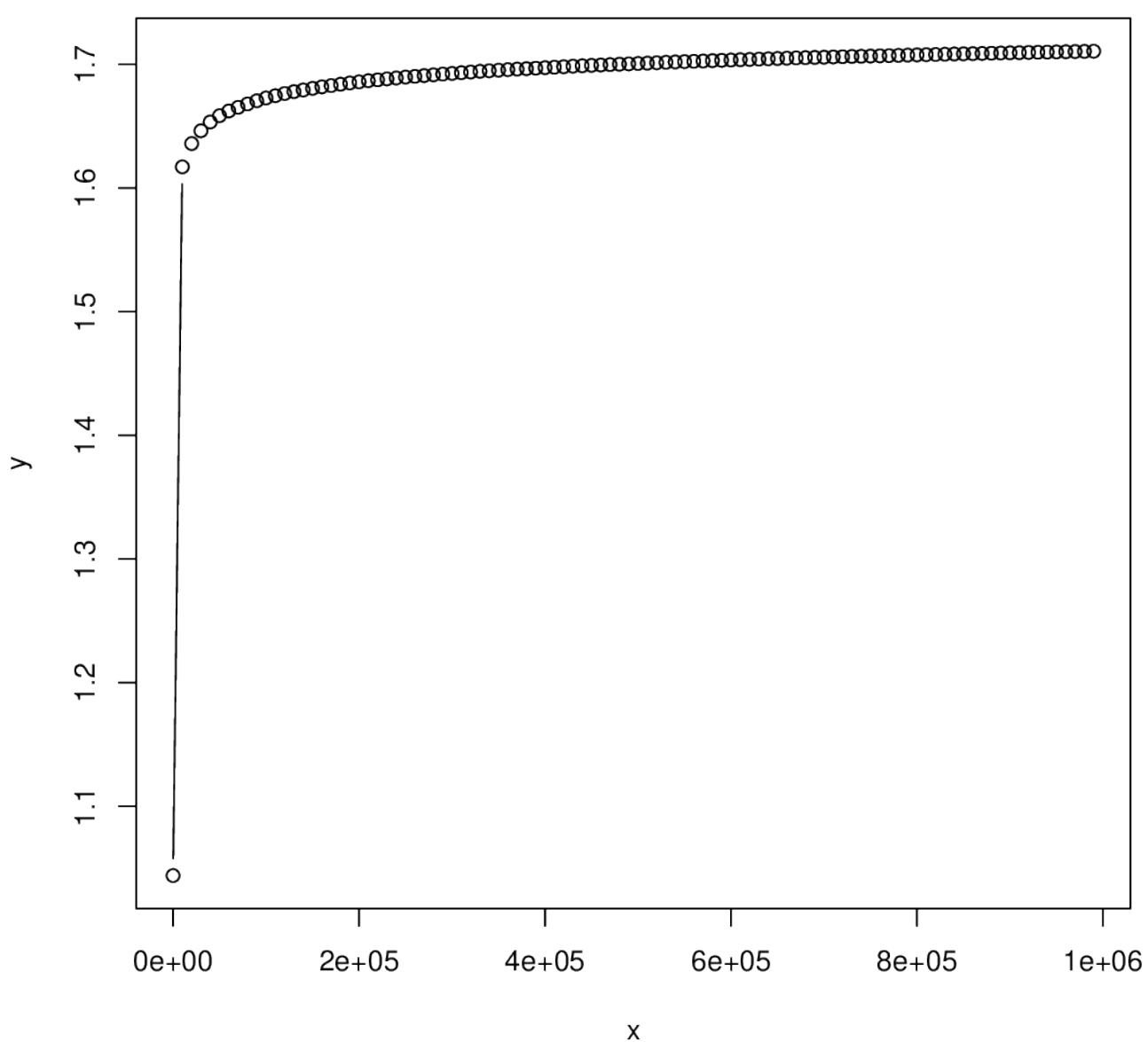
```
    return(sum(rt1plust2))
}

x=seq(13, 1000000, by=10000)
y=sapply(x, FUN = stp)
plot(x,y,type="b")
```

A prímszámok olyan számok, melyek csak önmagukkal és eggyel osztva nem adnak maradékot. Az ikerprímek pedig olyan príszmámok, melyek különbsége kettő.

A Brun téTEL azt mondja ki, hogy ha vesszük az ikerprímek reciprokát, majd elkezdjük összeadni őket, pl. $(\frac{1}{3} + \frac{1}{5}) + (\frac{1}{5} + \frac{1}{7}) + \dots$, akkor ez a sor egy B_2 (Brun-)konstanshoz fog konvergálni. Ez a felfedezés viszont nem oldja meg az ikerprímek számának problémáját, hiszen arról nem nyilatkozik, hogy a sor véges, vagy végtelen.

Ez a program az ikerprímeket ábrázolja egy koordináta rendszerben, melyen jól megfigyelhető hogy valóban a B_2 konstanshoz tartunk.



2.1. ábra. A B_2 konstans közelítése

2.8. A Monty Hall probléma



Tutorált

Ebben a feladatban tutoráltam Duszka Ákos Attilát.

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

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

Ebben a példában megismerkedhettünk az R programozási nyelvvel, amely egy főleg statisztikai számításokra alkalmas, interpreteres nyelv. Ennek köszönhetően a Monty Hall problémára alkotott R program forrása (is) könnyen olvasható és értelmezhető.

A Monty Hall probléma egy TV showból ered, melyben három ajtó közül kell választanunk, az egyik kinyitásával megtalálhatjuk a kincset, a másik kettővel pedig nem. Itt a szimulációban azt vizsgáljuk, hogy érdemes-e váltogatni, hogy mikor melyik ajtót nyitjuk ki, vagy érdemesebb ugyanannál az ajtónál maradni a játék körei során.

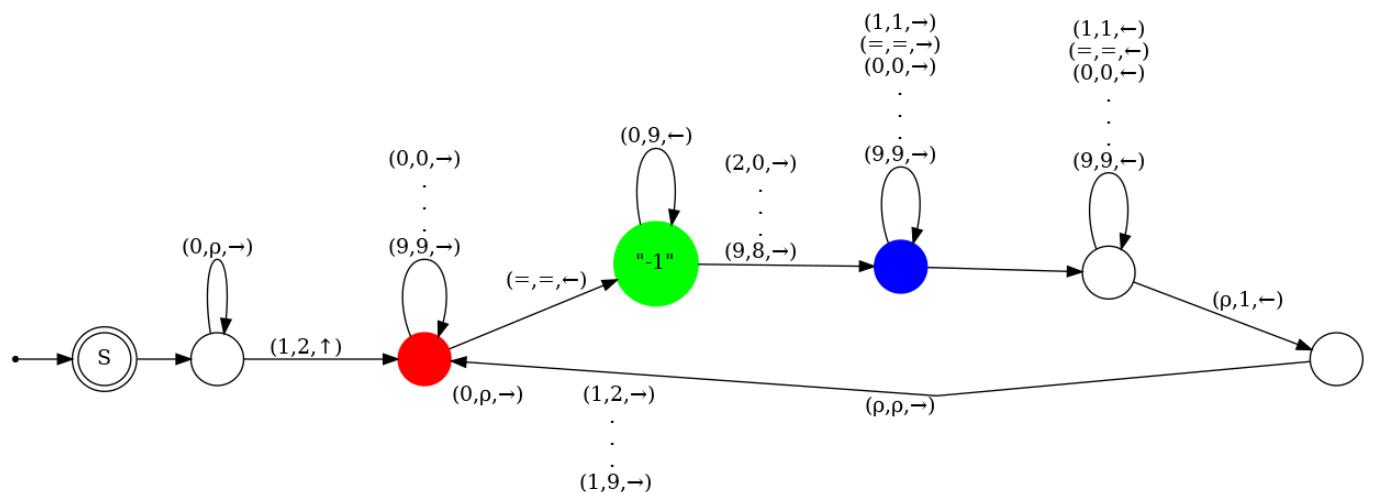
```
$ Rscript montyhall.r
[1] "Kiserletek szama: 10000000"
[1] 3330958
[1] 6669042
[1] 0.4994657
[1] 10000000
```

A program kimenetét elemezve azt láthatjuk, hogy 100000000 iteráció után az eredmény az, hogy mindenkor érdemes váltogatni az ajtóinkat, ugyanis így megközelítőleg 50 százalékos javulást érhetünk el. Itt a 3330958 azt mutatja, hogy az összes esetből ennyiszer nyerünk úgy, ha nem változtatunk az ajtónkon, amin benyitunk. A 6669042 pedig a nyertes esetek száma. A 0.4994657 pedig az előbbi kettőnek a hárnyadosa.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép



3.1. ábra. Példa ilyen Turing-gépre

Ránézésre a téma elégé száraznak és unalmasnak hangozhat, azonban hogyha jobban beleássunk magunkat a témaiba, rájöhetünk, hogy meglehetősen izgalmas tud lenni olykor. Programozóként nem szabad megijednünk a Turing-gépektől valamint alkalmazásuktól.

Az unáris számrendszer egy olyan egyszerű számrendszer, ahol vonalakkal, húzásokkal ábrázoljuk a számokat. Egy vonal egyet jelent, és összeadva kell értelmezni ezeket (pl. ||| =3). Ez a Turing-gép ezt az átváltást végzi el.

C-ben a következőképpen nézne ez ki:

```
#include <stdio.h>

int main(void)
```

```
{
    printf("Please enter a decimal number you'd like to convert to unary: ");

    unsigned int in = 0;
    scanf("%d", &in);

    for (int i = 0; i < in; ++i)
        printf((i % 5) ? "|" : " ");

    printf("\n");

    return 0;
}
```

Itt minden összesen annyi történik, hogy bekérünk egy számot, majd ötös csoportokban kiíratjuk a vonalkákat.

```
$ ./a.out
Please enter a decimal number you'd like to convert to unary: 50
||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| |||||
$ ./a.out
Please enter a decimal number you'd like to convert to unary: 51
||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| |
```

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

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

I. Legyenek S, X, Y változók. Legyen a, b, c konstansok.

$S \rightarrow abc, S \rightarrow aXbc, Xb \rightarrow bX, Xc \rightarrow Ybcc, bY \rightarrow Yb, Ay \rightarrow aax, Ay \rightarrow aa$

```
S
aXbc
abXc (Xc -> Ybcc)
abYbcc
```

- S ($S \rightarrow aXbc$)
- $aXbc$ ($Xb \rightarrow bX$)
- $abXc$ ($Xc \rightarrow Ybcc$)
- $abYbcc$ ($bY \rightarrow Yb$)

- $aYbbcc$ ($aY \rightarrow aa$)
- $aabbcc$

II. Legyenek S, X, Y változók. Legyen a, b, c konstansok.

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

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

Noam Chomsky az 50-es években élt és alkotott, nyelvészket végzett, ami azért lás-suk be, nem annyira meglepő, hiszen a mai napig sok programozó kerül ki a nyelvész szakmából, hiszen sok a hasonlóság a két szakma között.

Chomsky a Chomsky-féle nyelvosztályok megalkotója. A fent látottak pedig bizonyítékok arra, hogy az $a^n b^n c^n$ nyelv nem környezetfüggetlen, hiszen kettő nyelvtant is megadtunk, amik ezt a nyelvet generálják (valamint példákat rájuk).

3.3. Hivatkozási nyelv

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

C-ben az utasítás fogalmát Bachus-Naur formában megadhatjuk a következőképpen. Ezt a következő logika mentén olvassuk; Mi az utasítás? címkézett_utasítás VAGY összetett_utasítás VAGY stb... Ezen logika mentén olvassuk a következő szabálysorozatot! A szabályokat érdemes hátulról (visszafelé) olvasni.

```
utasítás ::=  
    címkézett_utasítás  
    | kifejezésutasítás  
    | összetett_utasítás  
    | kiválasztó_utasítás  
    | iterációs_utasítás  
    | vezérlésátadó_utasítás  
címkézett_utasítás ::=  
    azonosító : utasítás  
    case állandó_kifejezés : utasítás  
    default : utasítás  
kifejezésutasítás ::= kifejezés ;
```

```
összetett_utasítás ::= { deklarációs_lista utasítás_lista }
deklarációs_lista ::= deklaráció | deklarációs_lista deklaráció
utasítás_lista ::= utasítás | utasítás_lista utasítás
kiválasztó_utasítás ::=
    if ( kifejezés ) utasítás
    | if ( kifejezés ) utasítás else utasítás
    | switch ( kifejezés ) utasítás
iterációs_utasítás ::=
    while ( kifejezés ) utasítás
    | do utasítás while ( kifejezés ) ;
    | for ( kifejezés ; kifejezés ; kifejezés ) utasítás
vezérlésátadó_utasítás ::=
    goto azonosító ;
    | continue ;
    | break ;
    | return kifejezés ;
```

Előfordul olyan kódcsipet, mely az egyik C verzióval hibátlanul lefordul, míg más C verzió fordítója hibát dob rá. Nézzünk meg erre egy konkrét példát, ahol a csipetünk C99-el fordul, azonban C89-el nem.

```
#include <stdio.h>

int main()
{
    for (int i = 0; i < 5; i++) ;
    return 0;
}
```

A fenti csipet talán a leglátványosabb példája ennek. Próbáljuk meg lefordítani ezt C89 verzióval.

```
$ gcc -std=c89 c99.c
c99.c: In function 'main':
c99.c:4:2: error: 'for' loop initial declarations are only allowed in C99 ←
      or C11 mode
  for(int i=0; i < 5; i++);
  ^~~
c99.c:4:2: note: use option -std=c99, -std=gnu99, -std=c11 or -std=gnu11 to ←
      compile your code
```

Látható hogy a fordító a barátunk, megoldást is kínál a problémánkra. Amennyiben az általa javasolt opciókkal fordítunk, semmi probléma nincs.

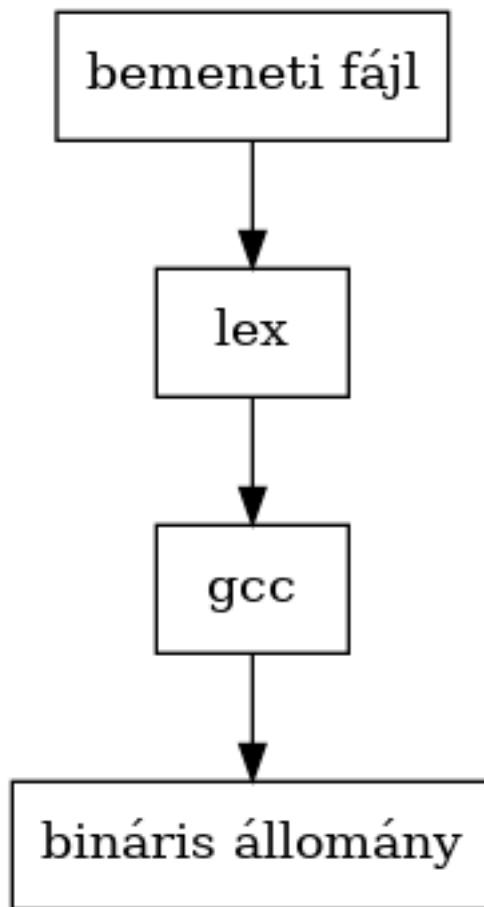
```
$ gcc -std=c99 c99.c
$
```

3.4. Saját lexikális elemző

**Tutorált**

Ebben a feladatban tutoráltam Duszka Ákos Attilát.

A program működési elve a következő:



3.2. ábra. Lexikális elemző

Látható, hogy az általunk írt állomány (*lex.l*) először bekerül a lex nevű programba, majd annak a kimenetét tudjuk lefordítani a fordítóval, esetünkben GCC-vel. Az ebből kapott bináris fájlt tudjuk majd futtatni is.

```
%{  
// C kod, amit meghagyunk 1-az-1-ben  
#include <stdio.h>  
%}
```

```
%option noyywrap

%%
[-+]?([0-9]*\.[0-9]+|[0-9]+) {
    printf("Valos talaltam: %s\n", yytext);
}
.\n {}%
%%

int main(void)
{
    yylex();
    return 0;
}
```

Itt a lex által ránk ruházott előnyöknél fogva egyszerűen csak megadjuk a definíciót, majd hagyjuk, hogy az óriások, akiknek a vállán állunk, dolgozzanak helyettünk.

Amit fent látunk, az nem más, mint a lex definícióink. A programnak megmondjuk, hogy mi értelmes, esetünkben ezt a [-+]?([0-9]*\.[0-9]+|[0-9]+) sor jelenti. Természetes nyelven olvasva ezalatt annyit értünk, hogy bármely szám nullától kilencig, akárhányszor előfordulása érvényes. A következő sorban az .|\n {} utasítás alapján minden más bemenetet ignorálunk.

Kicsit részletesebben a valós szám regex definíóján:

- [-+]?
'-' vagy '+', egyszer, vagy nullaszor
- (
Csoport kezdete
- [0-9]*
Szám nullától kilencig, akárhányszor (ez a pont előtti rész)
- \.
Szó szerint vett '.' (pont)
- [0-9]+
Szám nullátok kilencig, akárhányszor
- |
Logikai vagy
- [0-9]+
Szám nullától kilencig, akárhányszor
-)
Csoport vége

Látszik, hogy a reguláris kifejezésünk felépítése egyszerű. Elsőnek keresünk előjelet (negatív vagy pozitív) **ÉS** keressük az egészrészt valamint a törtrészt **VAGY** csak egészrészt.

Ebből az egyszerűnek tűnő kódból majd a lex farag nekünk egy értelmes C kódot, amit lefordítva láthatjuk is, hogy a programunk működik.

```
$ lex lex.l
$ gcc lex.yy.c -o lex
$ ./lex
123ad
Valost talaltam: 123
123ad45
Valost talaltam: 123
Valost talaltam: 45
{1}{2}
Valost talaltam: 1
Valost talaltam: 2
3.1415 1.4142
Valost talaltam: 3.1415
Valost talaltam: 1.4142
...
```

3.5. 133t.l



Tutorált

Ebben a feladatban tutoráltam Duszka Ákos Attilát.

Megoldás forrása:

```
$ lex 1337.l && gcc lex.yy.c && ./a.out
hello, world!
h3ll0, w0rld!
what's up?
wut's up?
this is so cool!
thls ls s0 kewl!
what's up mate?
wut's up m8?
hey dude!
h3y d00d!
```

```
loveu
10/3u
Using a lexer is fun!
Us1ng 4 l3x3r 1s fvn!
```

Fedezzük fel, hogy az óriások vállán állás nagyon jóvedelmező. Ezt a programot megadtuk a következő egyszerű szabályokkal, hogy specifikus bemenetre mit adjon, minden más pedig írjon le úgy, ahogy kapta. Ezzel a módszerrel sokkal egyszerűbb volt a feladatot megoldani, mint magunknak olvasni a bemenetet, és keresni benne a kicsérélendő karakterláncokat.

```
...
%%
"a"    { printf("4"); }
"c"    { printf("k"); }
"e"    { printf("3"); }
"o"    { printf("0"); }
"i"    { printf("1"); }
"t"    { printf("7"); }
"q"    { printf("kw"); }
"A"    { printf("/-\\" ); }
"B"    { printf("13"); }
"C"    { printf("K"); }
"E"    { printf("3"); }
"I"    { printf("1"); }
"V"    { printf("\\\\/"); }
"bye"  { printf("bai"); }
"and"  { printf("nd"); }
"dude" { printf("d00d"); }
[...] 
.|\\n  { printf("%s", yytext); }
%%
...
```

3.6. A források olvasása



Tutorált

Ebben a feladatban tutoráltam Füleky Ladislav-ot.

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

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

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



Bugok

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

i.

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

Ha a SIGINT jelkezelése nincs ignorálva, akkor innentől fogva **jelkezelo** végezze a jelkezelést.

ii.

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

Végezzük el ötször, hogy... (prefix increment)

iii.

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

Végezzük el ötször, hogy... (postfix increment)

iv.

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

Nem lehet megmondani, hogy mi fog történni, mert az i-t már használjuk ciklusváltozónak is, valamint arra is, hogy a tömb elemét beállítsuk. Nem érdemes olyan kódot írni, ami valamilyen kiértékelődési sorrendet feltételez, ez pedig pont olyan.

```
#include <stdio.h>

int main(void)
{
    int tomb[5] = { 1, 2, 3, 4, 5 };
    for (int i = 0; i < 5; tomb[i] = i++) {
```

```
    printf("%d %d\n", i, tomb[i]);  
}  
  
return 0;  
}
```

v.

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

A kód nehezen olvashatósága mellett kétséges a kiértékelődési sorrend is. Azt mondjuk, hogy legyen $i=0$, valamint ha i kisebb mint n , és a d rátövetkezője megegyezik az s rátövetkezőjével, akkor növeljük meg az i -t egyel.

```
#include <stdio.h>  
  
int main(void)  
{  
    int a = 4, b = 4, n = 5, *d = &a, *s = &b;  
  
    for (int i = 0; i < n && (*d++ = *s++); ++i) {  
        printf("%d\n", i);  
    }  
  
    return 0;  
}
```

Ha ezt a csipetet beággyazzuk egy programba, és futtatjuk, akkor láthatjuk, hogy nem teljesen az történik, mint aimt szerettünk volna.

vi.

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

Itt az történik, hogy egy `printf` függvény argumentumában próbáljuk inkrementálni a változókat, melyre nem lehetünk abban biztosak, hogy milyen sorrendben fognak kiértékelődni. Megbizonyosodhatunk róla, hogy a program nem a kívánt eredményt fogja visszaadni, hogyha megpróbáljuk lefuttatni.

```
#include <stdio.h>  
  
int f(int a, int b)  
{  
    return a + b;  
}  
  
int main()  
{  
    int a = 4;  
  
    // 4 + 4 = 8
```

```
// 4 + 5 = 9 -> ehelyett 12-et kapunk  
// 6 + 6 = 10 -> ehelyett 10-et kapunk  
printf("%d %d \n", f(a, ++a), f(++a, a));  
return 0;  
}
```

vii.

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

Írassuk ki az f függvény a-ra való outputját, valamint magát az a-t is.

viii.

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

Itt is problémába ütközünk, hiszen az f függvény referencia szerint kapja meg az a változót, tehát másolás nélkül. Ebben az esetben az f függvény közvetlenül tudja módosítani az a változót, emiatt megint kétséges a kiértékelődési sorrend, valamit a kód is meglehetősen "rossz minőségű".

```
#include <stdio.h>  
  
int f(int *a)  
{  
    return *a * *a;  
}  
  
int main()  
{  
    int a = 2;  
    // kimenet: 4 2  
    printf("%d %d \n", f(&a), a);  
    return 0;  
}
```

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

A LaTeX-ről annyit kell tudni, hogy ez egy szövegformázó rendszer, amelyet matematikusok előszeretettel használnak dokumentumok és prezentációk formázására, de jelen van a nyomdákban is. Használata a tanulási fázist követően egyáltalán nem megerőltető, véleményem szerint kényelmesebb is mint egy Word dokumentumot szerkesztgetni, valamint jóval több testreszabhatóságot rejt, melyek pár leütésre vannak, és az összkép még így is sokkal szébb, mint bármelyik konkurenséé.

Az egyszerűség kedvéért ezen sorokat betesszük egy logikus.tex fájlba, majd lefordítjuk a forrást egy segédprogram használatával, hogy könnyebben olvasható legyen.

```
$ pandoc -t latex logikus.tex -o logikus.pdf
```

Az elkészült dokumentumot alább beágyazva megtalálhatja az olvasó is.

$$\begin{aligned} & (\forall x \exists y ((x < y) \wedge (y \text{ prím}))) \\ & (\forall x \exists y ((x < y) \wedge (y \text{ prím}) \wedge (SSy \text{ prím}))) \\ & (\exists y \forall x (x \text{ prím}) \supset (x < y)) \\ & (\exists y \forall x (y < x) \supset \neg(x \text{ prím})) \end{aligned}$$

3.3. ábra. LaTeX forrás lefordítva

Az Ar egy elsőrendű matematikai logikai nyelv. Az elsőrendű nyelveket azért éri meg tanulmányoznunk, mert kiválóan lehet vele a világunkat modellezni.

Ezek az elsőrendű logikai kifejezések lefordítva természetes szövegre:

- A prímszámok száma végtelen.
- Végtelen sok ikerprím van.
- A prímszámok száma véges.
- A prímszámok száma véges.

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

```
#include <cstdlib>      // C++  
  
int main(void)  
{  
    // Egesz  
    int a;  
  
    // Egeszre mutato mutato  
    int *b;  
  
    // Egesz referenciaja  
    int *c = &a;  
  
    // Egeszek tombje  
    int d[5];  
  
    // Egeszek tombjenek referenciaja  
    // int *e = malloc(5 * sizeof(int));  
    int (&e)[5] = d;  
  
    // Egeszre mutato mutatok tombje  
    int **f[5];  
  
    // Egeszre mutato mutatot visszaado fv.  
    int *g(void);  
  
    // Egeszre mutato mutatot visszaado fv.re mutato mutato  
    int *(* (*h)(void)) = h;  
  
    // Egeszet visszaado es ket egeszet kapo fvre mutato  
    // mutatot visszaado, egeszet kapo fv  
    int (* (*i)(int))(int, int);
```

```
// Fuggvenymutato egy egeszet visszaado es ket egeszet  
// kapo fvre mutato mutatot visszaado, egeszet kapo  
// fvre  
int (* (*j)(int))(int, int) = i;  
  
return 0;  
}
```

Mit vezetnek be a programba a következő nevek?

```
int a;
```

egész

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

egészre mutató mutató

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

egész referenciaja C++ban

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

egészek tömbje

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

egészek tömbjének referenciaja

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

egészek tömbjére mutató mutató

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

egészre mutató mutatót visszaadó függvény

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

egészre mutatóra mutatót visszaadó függvény

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

két egészet kapó, egy egészre mutatót visszaadó függvény

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

két egészet kapó, egy egészet visszaadó függvényre mutatót mutató

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix



Tutorált

Ebben a feladatban tutoráltam Őz Ágostont.

```
#include <stdio.h>
#include <stdlib.h> // malloc()

int main(void)
{
    const int db = 5;

    double **hm;
    printf("Mutato cime: %p\n", &hm);

    hm = (double **)malloc(db * sizeof(double));
    printf("Sorok tombjainak cime: %p\n", hm);

    for (int i = 0; i < db; i++)
        hm[i] = (double *)malloc(db * sizeof(double));
    printf("Elso sor cime: %p\n", hm[0]);

    for (int i = 0; i < db; i++) {
        printf("|");
        for (int j = 0; j < db; j++) {
            hm[i][j] = (i + 1) / (j + 1);
            printf("%.2f ", hm[i][j]);
        }
        printf("|\n");
    }
}
```

```
    return 0;  
}
```

Ebben a példában a C nyelv fő jellegzetességével foglalkozunk, ami nem az egyszerűsége, hanem a dinamikus memóriakezelés. Nézzünk meg egy olyan példát, ahol létrehozunk egy double ** háromszögmátrixot, ami lényegében nem más, mint egy két dimenziós tömb (C-ben a [] jelek használata a tömbök kezelésére csupán egy fordító adta kényelem, hogy ne kelljen minden mutató-, és cím aritmetikával foglalkozunk). A különbség itt az lesz, hogy nem mondjuk meg előre, hogy hány elemű tömbjeink lesznek, hanem azokat majd dinamikusan foglaljuk le.

A program úgy működik, hogy elsőnek megadjuk, hogy hányszor hányas tömböt szeretnénk létrehozni, és ezt eltároljuk a db változóba. Ezután lefoglaljuk magát a hm háromszögmátrixot, és a megértés érdekében kiíratjuk a memória címét is (példa lentebb). Ezt követően lefoglalunk db darab (esetünkben 5) tömböt (ezek lesznek a sorok), amelyek a következő lépésekben szintén tartalmazni fognak 5 db tömböt (ezek pedig az oszlopok).

Egy alsó háromszögmátrix egy olyan mátrix, melyben a főátló felett csupán nullák vannak. A programunk egy kvadratikus háromszögmátrixot készít el, olyan módon, hogy minden elemét úgy állítjuk be, hogy $(i+1)/(j+1)$, ahol az i a sor száma, a j pedig az oszlop száma. A +1 azért kell, hogy ne osszunk nullával véletlenül se.

```
$ gcc doubleharomszog.c && ./a.out  
Mutato cime: 0x7fffbc435de0  
Sorok tombjainak cime: 0x5567bca69670  
Elso sor cime: 0x5567bca696a0  
| 1.00  0.00  0.00  0.00  0.00 |  
| 2.00  1.00  0.00  0.00  0.00 |  
| 3.00  1.00  1.00  0.00  0.00 |  
| 4.00  2.00  1.00  1.00  0.00 |  
| 5.00  2.00  1.00  1.00  1.00 |
```

A feladat tanulsága, hogy a [] jelek valóban csak a fordító adta kényelmi funkció. C-ben azt a kifejezést, hogy `hm[1][1]` legalább háromféleképpen fejezhetjük ki, melyek a következők: `(*hm[1]+1)`, vagy `(*(*hm + 1)+1)`.

4.2. C EXOR titkosító

```
#define MAX_KEY_SIZE 100  
#define BUFFER_SIZE 256  
  
char key[MAX_KEY_SIZE];  
char buffer[BUFFER_SIZE];  
  
int key_index = 0, read_bytes = 0;  
int key_size = strlen(argv[1]);
```

```
strncpy(key, argv[1], MAX_KEY_SIZE);

while ((read_bytes = read(0, (void *)buffer, BUFFER_SIZE))) {
    for (int i = 0; i < read_bytes; i++) {
        buffer[i] ^= key[key_index];
        key_index = (key_index + 1) % key_size;
    }

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

Az XOR titkosító programunk meglehetősen egyszerűen működik. Az argumentumból kiolvassuk a kulcsot majd a while ciklusban olvassuk az érkező bementet az úgynevezett sztenderd inputon. Majd megnézzük a bemenetünk méretét, és ennek a teljes tartalmát "titkosítjuk", azaz elvégezzük rajta a bitenkénti XOR utasítást `buffer[i] ^=key[key_index];`). Ezután vesszük a buffert és kinyomjuk az úgynevezett sztenderd outputon. Ez azért jó, mert a felhasználó egyszerűen inspekálhatja, vagy fájlba írhatja a kimenetet, legalábbis UNIX-jellegű operációs rendszerek alatt.

Természetesen ugyanezzel a kóddal és a kulcs ismeretében, a generált titkosított állomány vissza is fejthető.

4.3. Java EXOR titkosító

**Tutorált**

Ebben a feladatban tutoráltam Füleky Ladislavot.

**Tutorált**

Ebben a feladatban tutoráltam Tóth Atillát.

```
import java.util.*;

class XorEncode {
    public static void main(String[] args) {
        String kulcs = "";

        if(args.length > 0) {
            kulcs = args[0];
        } else {
            System.out.println("Kulcs nélkül nem titkosítok!");
```

```
System.out.println("Használat: java XorEncode.java [kulcs]");
System.exit(-1);
}

Scanner sc = new Scanner(System.in);
String str = "";

while(sc.hasNext()) {
    str = sc.next();
    System.out.println(xor(kulcs, str));
}
}

public static String xor(String kulcs, String s) {
    StringBuilder sb = new StringBuilder();

    for(int i = 0; i < s.length(); i++) {
        sb.append((char)(s.charAt(i) ^ kulcs.charAt(i % kulcs.length())));
    }

    return sb.toString();
}
}
```

A Java egy objektum orientált nyelv, melyet főleg enterprise környezetekben használnak, mert jellegénél fogva jól modulálható, valamint nincs benne semmiféle memóriakezelés, sokan emiatt hívják viccesen memóriakezelés nélküli C-nek is, mely érhető, hiszen a két nyelv szintaxisa között valóban vannak hasonlóságok.

Ebben a programban is hasonlóan járunk el, mint a C-s változatban, az argumentumok közül kiolvassuk a kulcsot, majd a standard inputon megjelenő szöveget XOR-ozzuk ezzel a kulccsal, majd a kimeneten kiíratjuk a végeredményt.

4.4. C EXOR törő

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

Megoldás forrása: [gh/krook1024/textbook/master/files/caesar/xor_c.c](https://github.com/krook1024/textbook/master/files/caesar/xor_c.c)

Ez a program csak olyan állományok feltörésére alkalmas, amelyeknek a kulcsa csak számokból áll, és 8 karakter hosszú. Ezt természetesen a kódban lehet módosítani.

A programunk egy bruteforce algoritmust használ, hogy visszafejtse a titkosított szöveget. A bemenetet olvasva a program meghatározza a szavak hosszát, és amennyiben ez megfelel egy bizonyos értékhatárnak, megnézi, hogy előfordulnak-e benne a leggyakoribb magyar szavak (*a, meg, vagy, van, volt, már, stb...*), és ha sikerül találni ilyeneket, akkor kiírja a kimenetre.

Ebben a példában ismét az OpenMP-t használjuk párhuzamosításra, hogy minél gyorsabban végezzünk a töréssel.

4.5. Neurális OR, AND és EXOR kapu



Tutorált

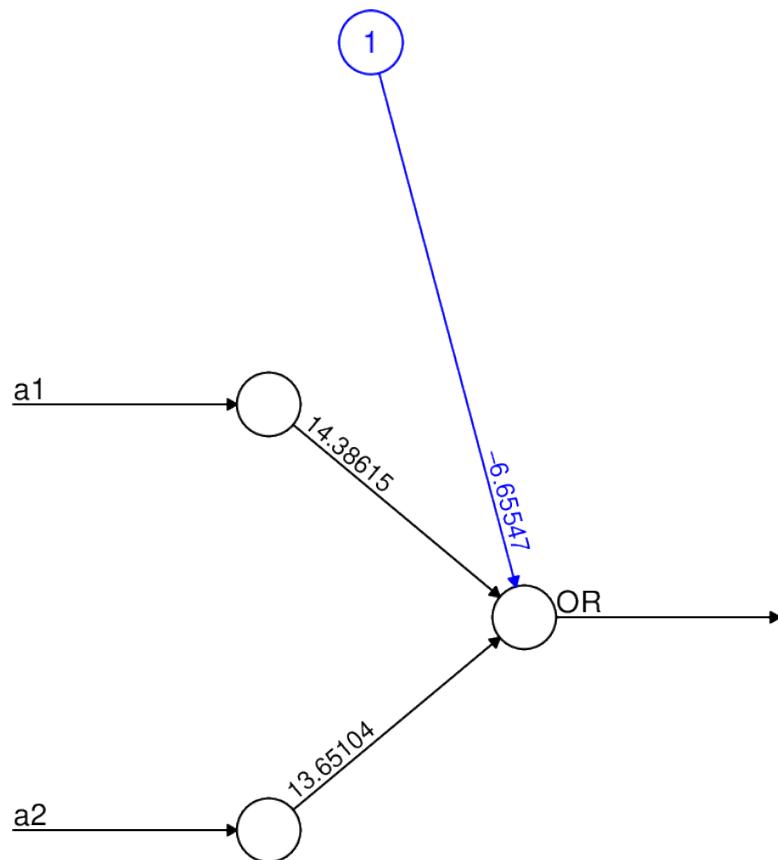
Ebben a feladatban tutorált Tóth Attila.

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

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

Ebben az R programban egy neurális hálót építünk fel. Egy neurális háló lényegében úgy működik, hogy megadjuk neki, hogy milyen bemenetre milyen kimenetet kell adnia, majd ő megprobálja ezt mesterségesen utánozni.

A programot futtatva a program kimenetét vizualizálva kapjuk, melyeket alább beágyozva láthat az olvasó is.



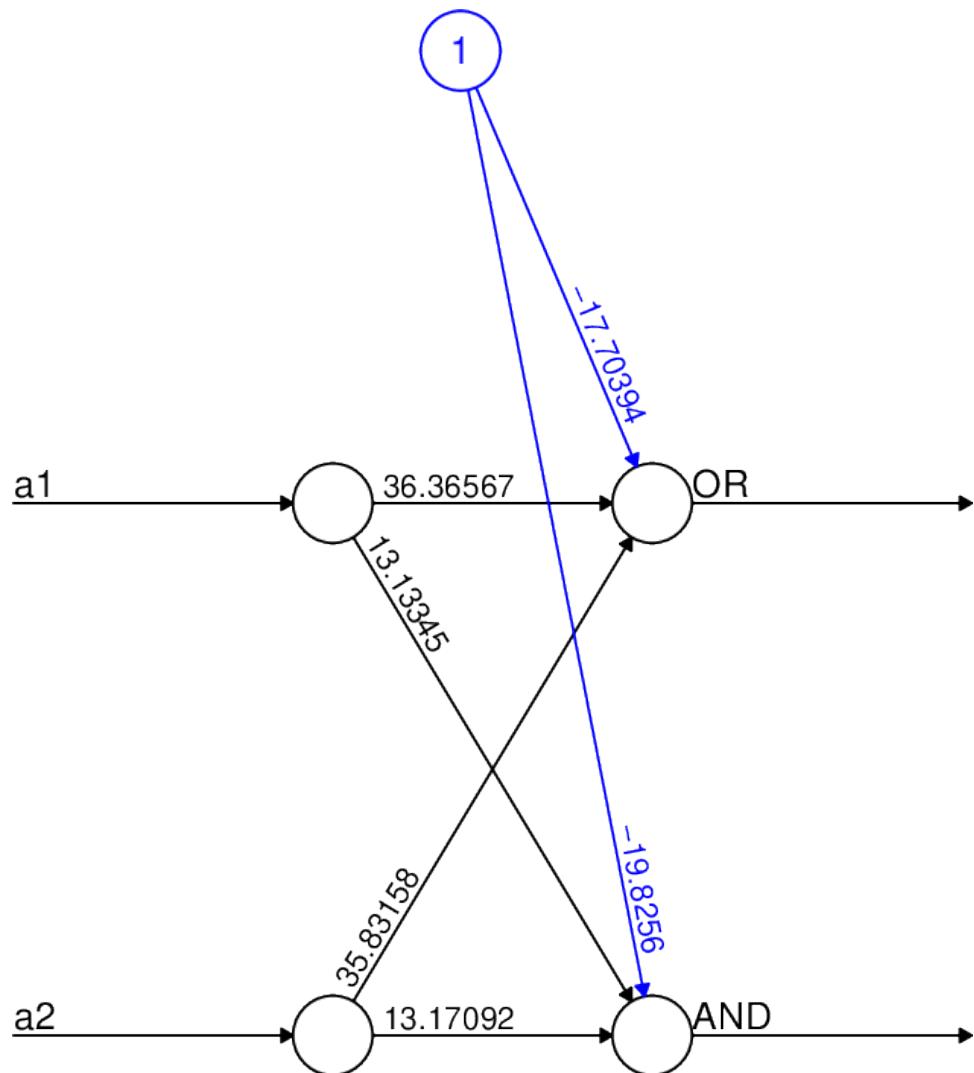
Error: 0.000001 Steps: 149

4.1. ábra. Neurális OR kapu

Erről az ábráról leolvasható, hogy 149 lépés alatt 0.000001 hibával sikerült megtanítani a neurális hálónkat az OR utasítás elvégzésére.

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

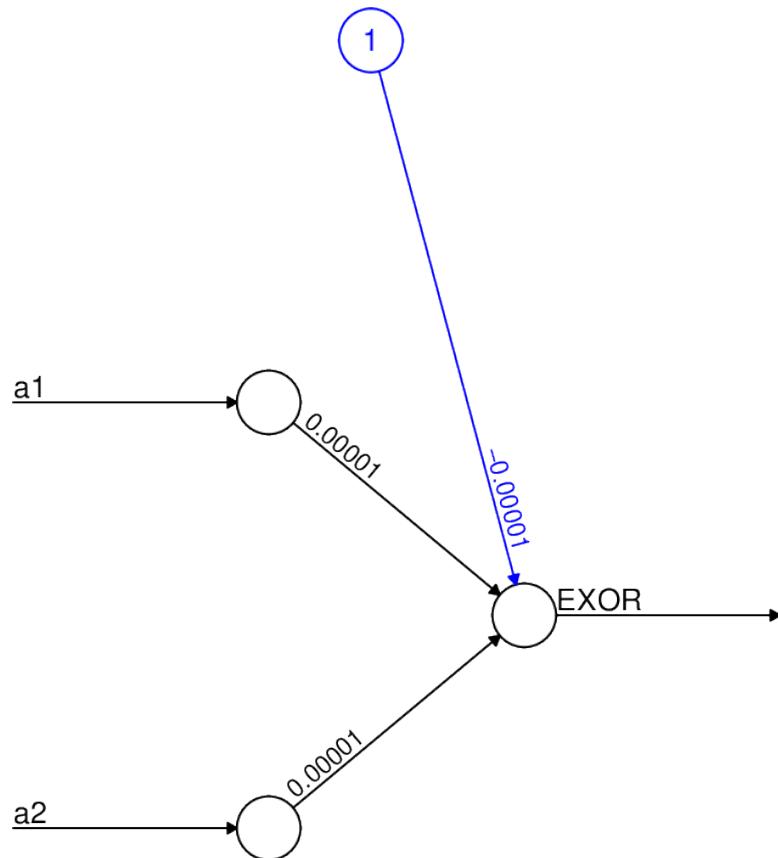
A hálót úgy építettük fel, hogy megadtuk, hogy milyen bemenetre milyen kimenetet kell adjon az OR (pl 0 OR 0 = 0; 1 OR 0 = 1; stb...), és mással nem kellett törődniünk, minden másról beleértve a teljes matematikai háttérét, az R és a neuralnet könyvtár végezte helyettünk.



Error: 0.000003 Steps: 359

4.2. ábra. Neurális OR, AND kapu

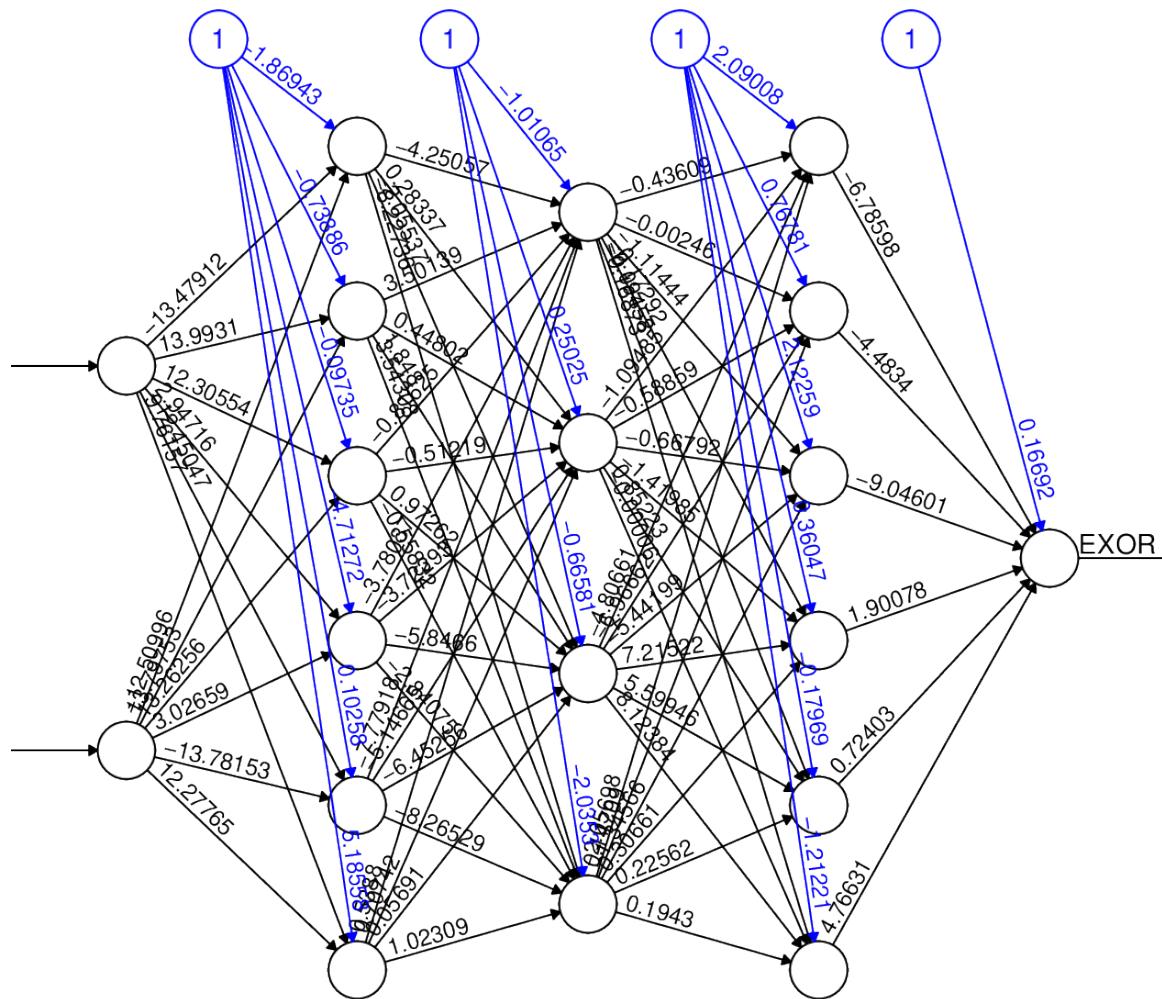
Ezen az ábrán viszont látjuk, hogy az OR és AND megtanulásához már 359 lépés kellett.



Error: 0.5 Steps: 94

4.3. ábra. Neurális EXOR kapu

Az EXOR megtanulásához 94 lépés volt elegendő, ám a hiba itt elég magas, 0.5. Emiatt a következő lépésekben új módszerekhez folymodunk, úgynevezett rejtett neuronokat iktatunk be.



Error: 0 Steps: 123

4.4. ábra. Neurális XOR kapu

Az új EXOR megtanulásához esetünkben 123 lépés volt szükséges, ám a hibát sikrult lecsökkentenünk 0-ra, olyan módon, hogy extra 14 rejtett neuront iktattunk be, 3 különböző szinten.

4.6. Hiba-visszaterjesztéses perceptron



Tutorált

Ebben a feladatban tutorált Duszka Ákos Attila.

Megoldás forrása: <https://github.com/krook1024/textbook/tree/master/files/caesar-perceptron>

```
#include "ml.hpp"

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

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

    int size = image.get_width() * image.get_height();

    Perceptron *p = new Perceptron(3, size, 256, 1);

    double *image_d = new double[size];

    for(int i = 0; i < image.get_width(); i++)
        for(int j = 0; j < image.get_height(); j++)
            image_d[i*image.get_width() + j] = image[i][j].red;

    double value = (*p)(image_d);

    std::cout << value << std::endl;

    delete p;
    delete[] image_d;

    return 0;
}
```

Tekintve hogy a program meglehetősen könnyen olvasható, eltekintenék a soronkénti magyarázástól, hiszen itt semmi mágikust nem történik.

Ebben a programban kipróbáljuk a Perceptron osztályt egy viszonylag egyszerű program keretében. Ez a program minden összetevőjét csinálja, hogy a bemenetén érkező képnek végigmegy az összes pixelén, és számolja a piros színkomponenst.

Az olvasó kényelmének érdekében ismét létrehoztunk egy makefilet, ami a tesztüket is lefuttatja, és ha minden jól megy, csak az eredményt adja vissza. Próbáljuk is ki ezzel fordítani a programunkat:

```
$ make
0.500964
```

Itt a kimenet közelítőleg 0.5, ez azt jelenti, hogy a piros színkomponens jelenléte 50% az egész képben. Ez ránézésre pontos, hiszen a képünk felét elfoglalja a fekete Mandelbrot-halmaz képe, a másik fele pedig a fehér keret, ami természetesen tartalmazza a piros színkomponenst. (A szóban forgó kép mgetalálható párt oldallal később a könyvben.)

5. fejezet

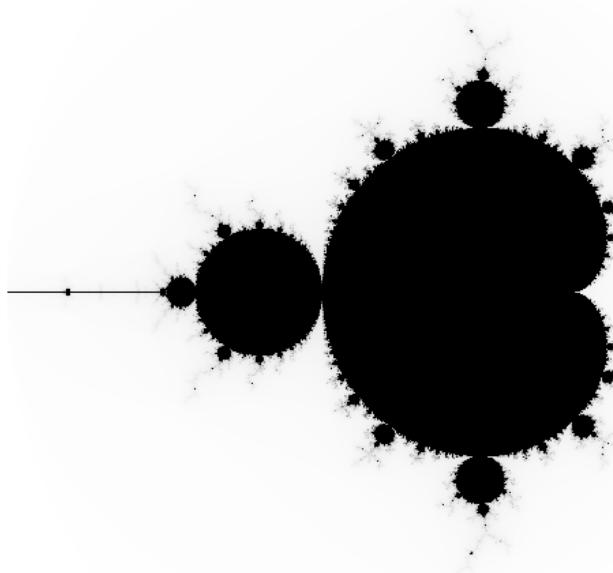
Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás forrása: [github/krook1024/textbook/mandelbrot](https://github.com/krook1024/textbook/mandelbrot)

Ebben a C++ programban a Mandelbrot halmazt ábrázoljuk egy képen. A Mandelbrot halmaz egy olyan komplex halmaz, amely illeszkedik a $f_c(z) = z^2 + c$ függvény képére, s nullától iterálva nem divergál, tehát a $f_c(0), f_c(f_c(0)), \dots$ abszolútértekben korlátos.

Amennyiben a forrásfájlt egyszerűen make-el lefordítjuk, úgy láthatjuk, hogy létrejön a kimenet.png, hiszen a Makefile-unkban úgy adtuk meg, hogy a forrást ne csak lefordítsa, hanem futtassa is a programunkat kimenet.png kimenettel. A kimenetet be is ágyazom, mert meglehetősen látványos.



5.1. ábra. A Mandelbrot halmaz a komplex síkon

A program alapját a *png++* C++ könyvtár adja, amely egy wrapper a C-s *libpng* könyvtárra.

Ebben a programban viszont nem használjuk a `std::complex` osztályt, amit a C++ biztosít, hanem egyszerűen magunknak adunk meg egy Komplex struktúrát, mely egyszerűen tartalmaz két double változót, ami megfeleltethető a komplex számok valós és imaginárius értékeinek. A további működési elv minden össze annyi, hogy megadunk egy halmazt, amelyre a megadott egyenletünk illeszkedik, majd végigmegyünk ezen a halmazon, és beszínezzük az erre alkalmas pixeleket.

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

Megoldás forrása: [github/krook1024/textbook/mandelb/mandelbrot_komplex/](https://github.com/krook1024/textbook/mandelb/mandelbrot_komplex/)

Ebben a programban, hasonlóan az előzőhez, a Mandelbrot halmazt ábrázoltatjuk. A különbség csupán annyi, hogy itt a beépített `std::complex` osztályt használjuk, ahelyett, hogy saját struktúrát írnának a komplex számok tárolására.

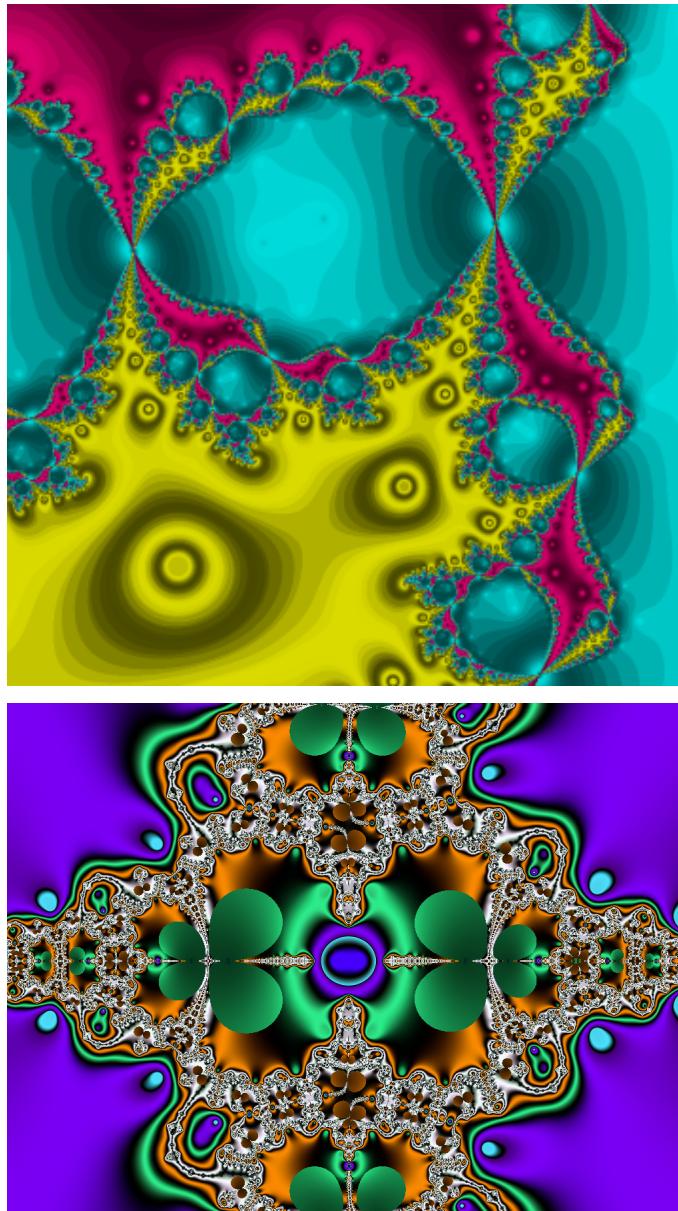
Ha elkészítünk egy egyszerű Makefilet a programhoz, úgy szintén egyszerű lesz azt futtatni. Nem meglepő, hogy ugyanazt a kimenetet adja, mint az előző programunk.

A `std::complex` osztály használata jár némi előnnyel, például lehetőséget ad kényelmesebb, elegánsabb változó definiálására (pl. `C ={MINX + j * dx, MAXY - i * dy};`, ahol az első rész a szám valós része, a másik pedig a szám imaginárius része).

5.3. Biomorfok

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

A biomorfok szorosan kapcsolódnak a Mandelbrot-halmazhoz. Hasonlóan, itt is a komplex számsíkon ábrázolható függvényeket nézünk meg, és ezek ábrázolásának C++ megvalósításait. Kedvcsinálóként előbb nézzünk meg egy pár biomorfot.

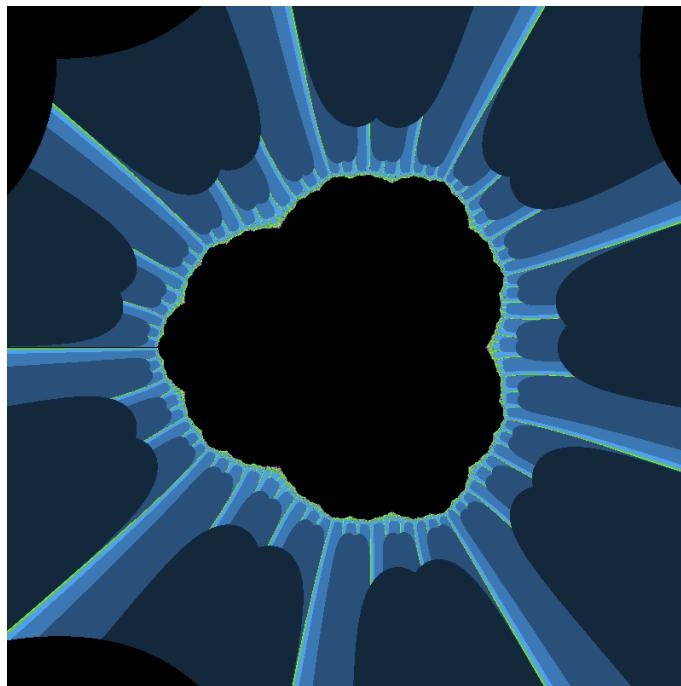


A biomorfok szoros kapcsolatban vannak a Mandelbrot-halmazzal, ám itt a $f_c(z) = z^2 + c$ képletet lecseréljük másra.

Az egyszerűség kedvéért itt is létrehoztunk egy Makefile-t, így a `make` parancs kiadásával lefuttatható a programunk, feltéve hogy telepítettük a `png++` könyvtárat és működik a `g++`-unk.

```
$ make
```

Hogyha kiadjuk a make parancsot, úgy létrejön egy bmorf.png fájlunk, amit beágynálva meg is tekinthet az olvasó.



5.2. ábra. A generált biomorf

Magán a formán kívül észrevehető további különbség az, hogy színes ábrát kaptunk a Mandelbrot-halmazos programunkkal ellentétben. A program működési elve az említett módon hasonló, csak más képlettel dolgozunk, valamint a színezést úgy valósítjuk meg, hogy az iteráció számát maradékosan osztjuk 255-el, így kapunk egy RGB színkódot, és ezt használjuk fel egy adott pixel színezésére.

5.4. A Mandelbrot halmaz CUDA megvalósítása



Tutorált

Ebben a feladatban tutoráltam Tóth Attilát.

Megoldás forrása: https://github.com/krook1024/textbook/blob/master/files/mandelb/cuda_mandel/cuda_mandel.cu

A CUDA az nVidia kártyák által használt API a kártyák vezérlésére nagyobb teljesítménnyel, mint például az OpenGL. A CUDA jelen van a legtöbb mai játékban, vagy

streamer, videóvágó szoftverben. A teljesítménynövekedést párhuzamos számítással érik el, több "magot" dolgoztatnak a videókártyán egyszerre. Ezen kívül használható másra is az interfész, nem csak grafikus dolgok programozására. Például széles körben elterjedt szokás CUDA kártyákkal "bányászni" különböző kriptovalutákat.

CUDA programok fejlesztéséhez hasznos olvasmány lehet a dokumentáció, amely a következő linken található meg; <https://docs.nvidia.com/cuda/index.html>.

Tekintsük meg a forráskódot egy pillanatra. A kód már ismerős lehet, hiszen ugyan azt az algoritmust taglaljuk az egész fejezetben. Megjelenik a Mandelbrot-halmaz számító while ciklusunk, a png++, és minden egyéb megszokott csipet amit ebben a fejezetben olvashattunk.

A fő különbséget az adja, hogy most a CUDA API-n keresztül kell dolgoznunk. Itt újdonságképpen jelennek meg a `_device_`, `_global_` és hasonló jellegű szövegek.

A dokumentációt olvasva hamar rájöhünk, hogy mit is jelentenek ezek. A `_device_` minősítővel rendelkeznek a magán a GPU-n futó függvények. A `_global_` minősítővel az úgynevezett "kernel" rendelkezik. Ezen belül igyekszünk megvalósítani a párhuzamosítást. Ezekre úgy gondolunk, mint például a `short int a;` kifejezés során, ahol a `short` a minősítő, az `int` a típus, az `a` pedig a változó neve. minden más pedig hoszt-oldalon fut, tehát a gazdagépen, a processzoron.

Ennyi előképzettséggel viszont már lássuk a kódot!

```
// Copyright (C) 2019
// Norbert Batfai, batfai.norbert@inf.unideb.hu
// Released under GNU GPLv3

#include <png++/image.hpp>
#include <png++/rgb_pixel.hpp>
#include <sys/times.h>
#include <iostream>

#define SIZE 600
#define ITERATION_LIMIT 32000

// Vegigzongorazza a CUDA a szelesség x magasság racsot:
_device_ int mandel(int k, int j)
{
    // most eppen a j. sor k. oszlopaban vagyunk

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

    float dx = (b - a) / width;
    float dy = (d - c) / height;
    float reC, imC, reZ, imZ, ujreZ, ujimZ;
    int iteration = 0;

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

```
reZ = 0.0;
imZ = 0.0;
iteration = 0;

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

    ++iteration;
}

return iteration;
}

__global__ void mandelkernel(int *buffer)
{

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

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

buffer[j + k * SIZE] = mandel(j, k);
}

void cudamandel(int buffer[SIZE][SIZE])
{
    int *deviceImageBuffer;
    cudaMalloc((void **)&deviceImageBuffer, SIZE * SIZE * sizeof(int));

    dim3 grid(SIZE / 10, SIZE / 10);
    dim3 tgrid(10, 10);
    mandelkernel <<< grid, tgrid >>> (deviceImageBuffer);

    cudaMemcpy(buffer, deviceImageBuffer, SIZE * SIZE * sizeof(int),
               cudaMemcpyDeviceToHost);
    cudaFree(deviceImageBuffer);
}

int main(int argc, char *argv[])
{
    // Merunk időt (PP 64)
    clock_t delta = clock();
```

```
// Merunk időt (PP 66)
struct tms tmsbuf1, tmsbuf2;
times(&tmsbuf1);

int buffer[SIZE][SIZE];

cudamandel(buffer);

png::image<png::rgb_pixel> image(SIZE, SIZE);

for (int j = 0; j < SIZE; ++j) {
    //sor = j;
    for (int k = 0; k < SIZE; ++k) {
        image.set_pixel(k, j,
            png::rgb_pixel(255 -
                (255 * buffer[j][k]) /
                ITERATION_LIMIT,
                255 -
                (255 * buffer[j][k]) /
                ITERATION_LIMIT,
                255 -
                (255 * buffer[j][k]) /
                ITERATION_LIMIT));
    }
}

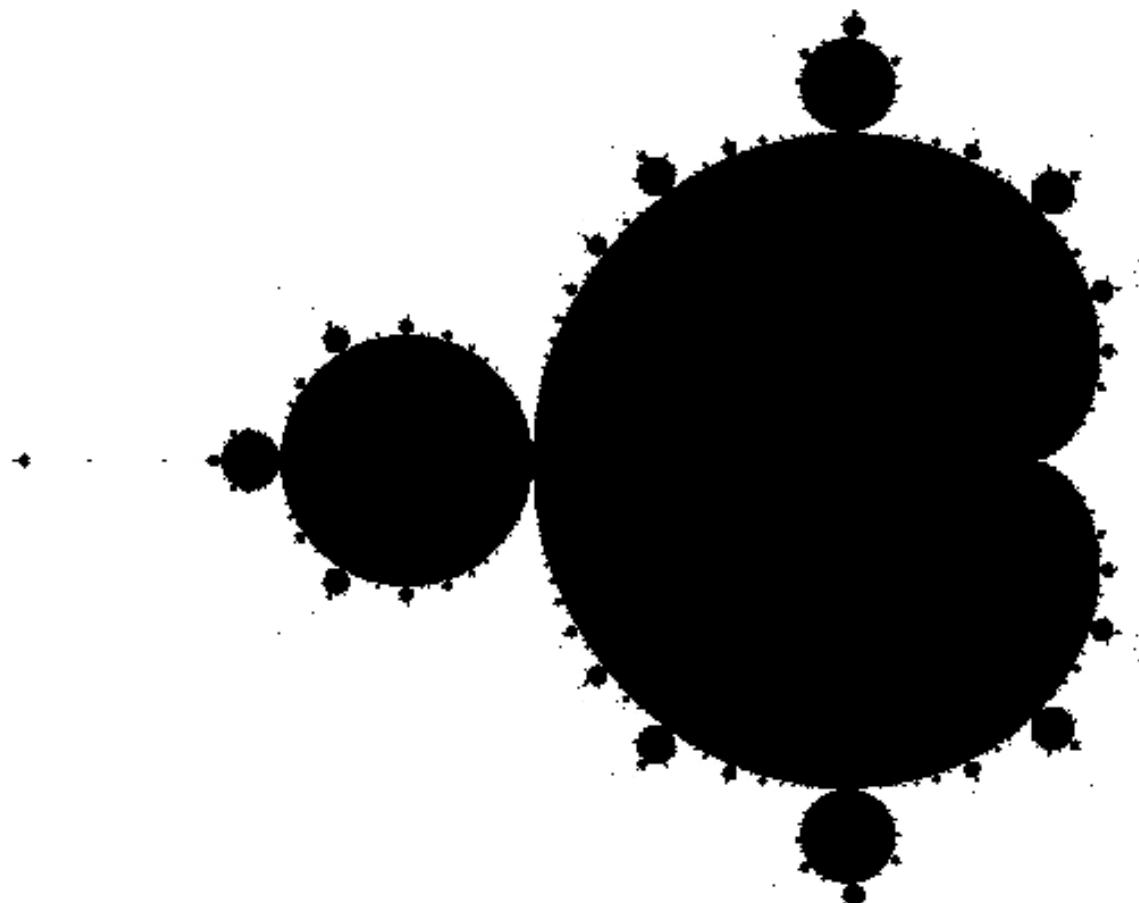
image.write("mandel.png");

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;
}
```

Segítséggéppen itt is mellékeltem egy `Makefile`-t, amivel könnyen és egyszerűen fordítható a program, feltéve, hogy a CUDA telepítére van a számítógépünkre, és van erre alkalmas kártyánk. Amennyiben ezek szükséges feltételek teljesülnek, de mégsem működne, próbáljuk meg a `Makefile`-ban korrigálni a fordító abszolút elérési útvonalát.

Hogyha fordítjuk és futtatjuk a programot, akkor láthatjuk hogy létrejön a `mandel.png` állományunk, amely szokás szerint be lesz ágyazva az olvasó számára.



5.3. ábra. Mandelbrot halmaz CUDA-val

```
$ make  
16  
0.159764  
$
```

A program kimenetéből látszik, hogy jóval gyorsabb mint a C++ és C-s verziók, ahol akár fél percet is várunk kellett egy-egy ábra elkészülésére, itt viszont a másodperc

törtrésze alatt elkészült a kép, mindez egy olyan számítógépen, amiben egy átlagos videókártya van és SSD nincs. Tehát a sebességnövekedés jelentős.

5.5. Mandelbrot nagyító és utazó C++ nyelven

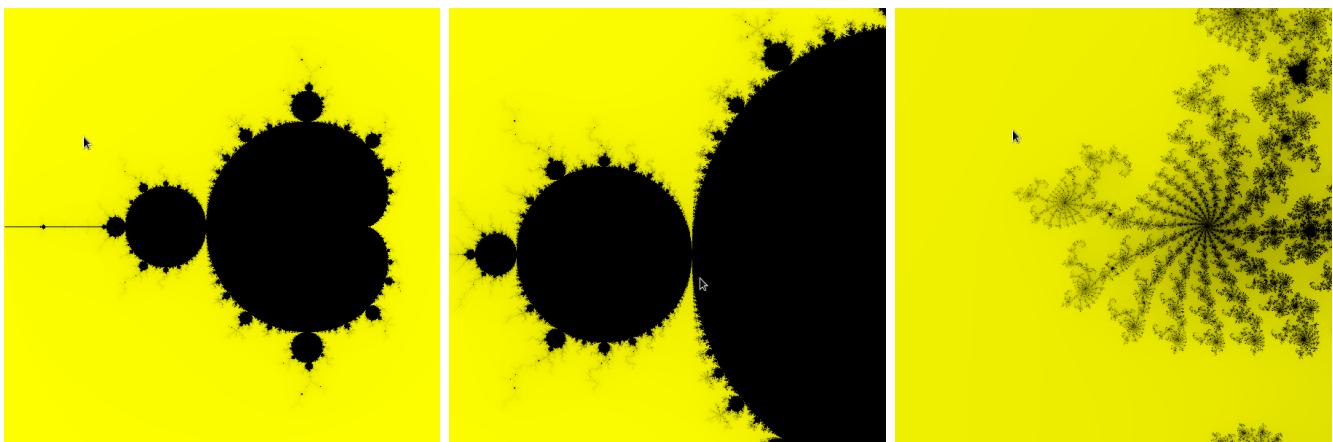
Építs GUI-t a Mandelbrot algoritmusra, lehessen egérrel nagyítani egy területet, illetve egy pontot egérrel kiválasztva vizualizálja onnan a komplex iteráció bejárta z_n komplex számokat

Megoldás forrása: https://github.com/krook1024/textbook/tree/master/files/mandel/mandel_nagyito

A feladat megvalósításához a Qt Creator nevű szoftvert fogjuk használni, mely egy több platformon átívelő keretrendszer elsősorban grafikus alkalmazások készítésére. A Qt nagyon népszerű választás a nagyobb projektek körében is, például a Qt az alapja a VLC-nek, és még sok más **szabad szoftvernek**.

Ebben a programban alapul vesszük a már meglévő Mandelbrot-halmaz számító C++ programunkat, és egy más kontextusba ültetjük be. Természetesen ez még nem lenne elég, hanem el kell készítenünk hozzá Qt Creatorban a felületet, valamint némi extra kódot. Ez a kód a repóban megtalálható.

A program használta során az egérrel jelölünk ki egy adott területet, melyet az újra renderel és betölt. A programról használat közben beillesztek néhány képernyőképet.



5.6. Mandelbrot nagyító és utazó Java nyelven

A feladat megoldásához az OpenJFX 11-et fogjuk használni. Az OpenJFX projekt weboldala megtalálható itt: <https://openjfx.io/>. Az OpenJFX, hasonlóan a Qt-hoz, egy keretrendszer grafikus felületek készítéséhez. Előnye, hogy kiváló útmutatást adnak ahhoz, hogy miképp és hogyan tudjuk elkezdeni a saját projektünket.

Például egy egyszerű program, ami nem csinál mást, csak kiírja az OpenJFX és a Java verzióját egy grafikus ablakban a következőképpen néz ki:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
public class HelloFX extends Application {
    @Override
    public void start(Stage stage) {
        String javaVersion = System.getProperty("java.version");
        String javafxVersion = System.getProperty("javafx.version");
        Label l = new Label("Hello, JavaFX " + javafxVersion + ", running ←
            on Java " + javaVersion + ".");
        Scene scene = new Scene(new StackPane(l), 640, 480);
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] args) {
        launch();
    }
}
```

Szerencsénkre C++ és Java között egyszerű a fordítás, néhány nyelvi sajátosságon, könyvtáron, valamint a mutatókon kívül semmit sem kell változtatnunk. Esetünkben a Mandelbrot-halmaz számító programunk C++ változatához képest minden össze egy struct-ot kell átírunk class-ra, valamint az utazást és a megjelenítést megvalósítani. Nézzük is meg a kódot!

```
/*
 * Mandelbrot osztály. A Mandelbrot halmaz számítására
 * valamint ábrázolására.
 * Készíte: Molnár Antal Albert.
 */

import javafx.scene.*;
import javafx.scene.paint.*;
import javafx.scene.canvas.*;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.application.Application;
import javafx.scene.input.MouseEvent;
import javafx.event.*;

public class Mandelb extends Application {
    private static final int N = 500;
    private static final int M = 500;

    private static double mousex, mousey;
```

```
@Override
public void start(Stage stage) {
    stage.setTitle("Hello, Mandelbrot!");

    Group root = new Group();

    Scene s = new Scene(root, N, M, Color.BLACK);

    final Canvas canvas = new Canvas(N, M);
    GraphicsContext gc = canvas.getGraphicsContext2D();

    root.getChildren().add(canvas);

    final double MAXX = 0.7;
    final double MINX = -2.0;
    final double MAXY = 1.35;
    final double MINY = -1.35;
    Mandelb m = new Mandelb();
    m.drawMandel(gc, m.calculateMandelbrot(MAXX, MINX, MAXY, MINY));

    root.setOnMouseMoved(new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
            mousex = event.getX();
            mousey = event.getY();
            //System.out.printf("coordinate X: %.2f, coordinate Y: %.2f\n", ←
                mousex, mousey);
        }
    });

    s.addEventFilter(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent <-
        >() {
        @Override
        public void handle(MouseEvent event) {
            // System.out.println("click! " + mousex + " " + mousey);
            double dx = (MAXX - MINX) / N;
            double dy = (MAXY - MINY) / M;

            double nMAXX, nMINX, nMAXY, nMINY;
            double range = 60;

            nMINX = MINX + (mousex*dx);
            nMAXX = MINX + (mousex*dx) + (range*dx);
            nMINY = MAXY - (mousey*dy) - (range*dy);
            nMAXY = MAXY - (mousey*dy);

            m.drawMandel(gc, m.calculateMandelbrot(nMAXX, nMINX, nMAXY, nMINY)) ←
                ;
        }
    });
}
```

```
        }

    });

    stage.setScene(s);
    stage.show();
}

public static void main(String[] args) {
    launch();

    Mandelb m = new Mandelb();
}

public static void drawMandel(GraphicsContext gc, int[][][] tomb) {
    for(int i = 0; i < tomb.length; i++) {
        for(int j = 0; j < tomb[i].length; j++) {
            gc.setFill(Color.rgb(tomb[i][j], tomb[i][j], tomb[i][j]));
            gc.fillRect(i, j, 1, 1);
        }
    }
}

public static int[][][] calculateMandelbrot(double MAXX, double MINX, ←
    double MAXY, double MINY) {
    int[][][] tomb = new int[N][M];
    int i, j, k;

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

    Komplex C = new Komplex();
    Komplex Z = new Komplex();
    Komplex Zuj = new Komplex();

    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;
            }
            tomb[i][j] = Zuj.re;
        }
    }
}
```

```

        Z.re = Zuj.re;
        Z.im = Zuj.im;
    }

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

return tomb;
}

public static class Komplex {
    public double re, im;
}
}

```

The screenshot shows a terminal window with the following content:

```

1 2 3
java_mandelb + ./run
Mandelb.java:32: error: incompatible types: possible lossy conversion from double to int
    final int MAXX = 0.7;
                           ^
Mandelb.java:33: error: incompatible types: possible lossy conversion from double to int
    final int MINX = -2.0;
                           ^
Mandelb.java:34: error: incompatible types: possible lossy conversion from double to int
    final int MAXY = 1.35;
                           ^
Mandelb.java:35: error: incompatible types: possible lossy conversion from double to int
    final int MINY = -1.35;
                           ^
4 errors
java_mandelb + ./run
java_mandelb + ./run
java_mandelb + ./run
java_mandelb + ./run

```

On the right side of the terminal, there is a small image of the Mandelbrot set, which is a fractal pattern of black and white pixels.

5.4. ábra. Mandelbrot nagyító és utazó Java nyelven

A végeredmény itt látható. A forrásfájlok pedig megtalálhatóak a repóban.

Megoldás forrása: https://github.com/krook1024/textbook/tree/master/files/mandelb/java_mandelb

6. fejezet

Helló, Welch!

6.1. Első osztályom

Megoldás forrása:

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

A módosított polártranszformációs generátor argumentum két véletlenszerű érték generálása alkalmazható. Próbáljuk is ki!

```
$ ./java/bin/java polargen.java
-0.7353431820414118
-0.33784190028284766
0.7750031835316805
0.5524713543467192
-0.5380423283211784
1.512849268596637
2.7148874695500966
-0.23688836801277952
-0.3238588036816322
-0.7963150809415576
$ ./java/bin/java polargen.java
-0.6566325405553158
0.40465899229436114
0.08634239512228409
-0.9470321445590416
0.1926238606249351
0.7705517022243931
0.9084531239664848
-1.4472688950554047
-1.6250659297425345
-0.7791586500972545
```

A program 10 darab véletlenszerűen generált normalizált számot köp ki, ahogyan azt várjuk is. Az OO előnye itt látszik meg, hiszen a matematikai háttér számunkra úgymond lényegtelen, és ezt az osztály el is rejtja a programozó szeme elől. A programunk mégis működik.

6.2. LZW

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

Megoldás forrása: [gh/krook1024/textbook/master/files/welch/z3a7.cpp](https://github.com/krook1024/textbook/master/files/welch/z3a7.cpp)

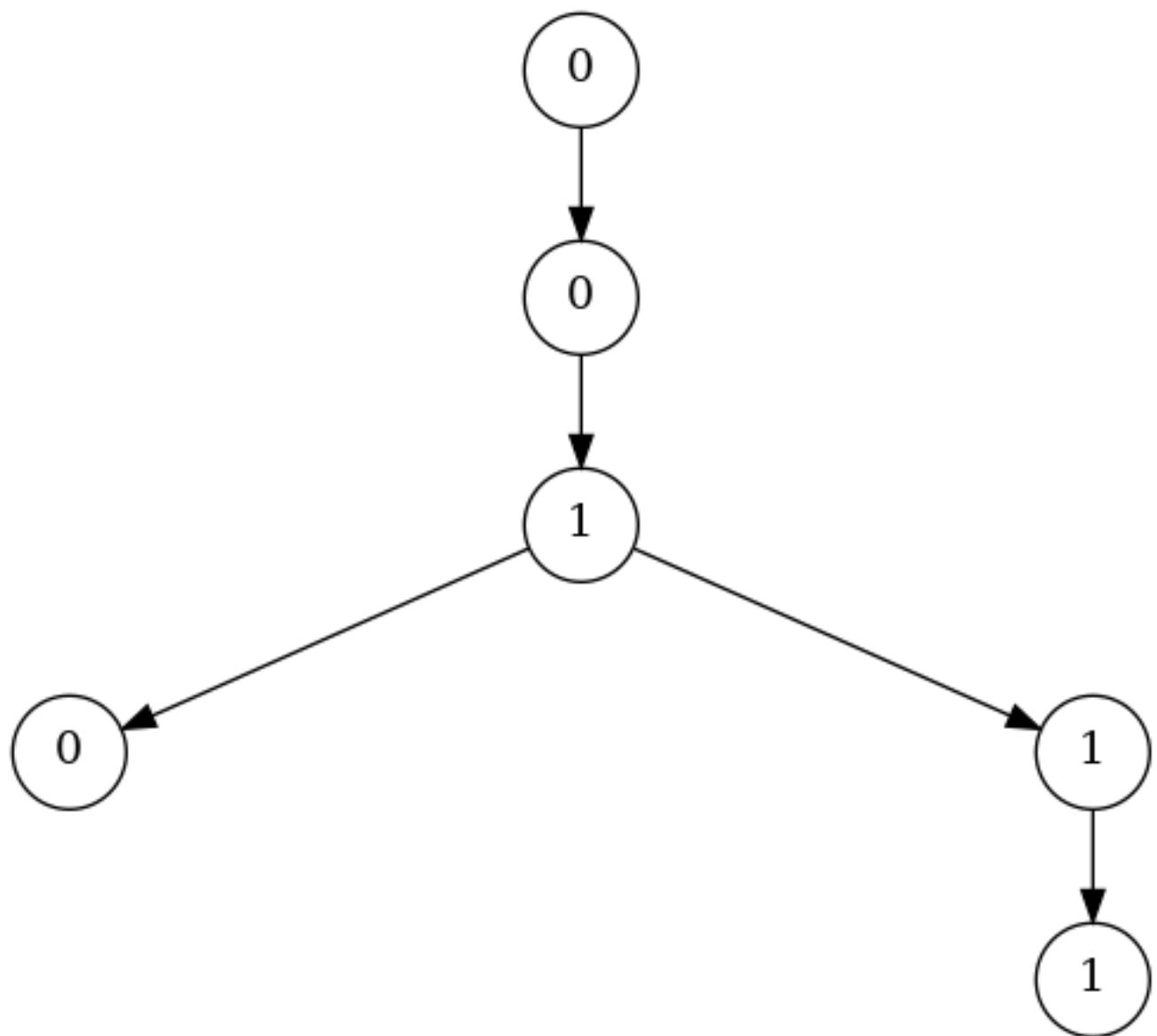
A LZWBInFa osztály felépíti a bemeneti fájl bináris fágát. Használata a következőképp néz ki: `./binfa [bemenő fájl] -o [kimenő fájl]`.

Egy bináris fát a következőképpen építhetünk fel:

- amennyiben 0-st kell betenni a fába, úgy megnézzük, hogy az aktuális csomópont tartalmaz-e 0-s elemet
 - amennyiben nem tartalmaz, úgy betesszük a nullás gyermekének a 0-t
 - amennyiben már tartalmaz, úgy készítünk egy új csomópontot és az ő gyermeként tesszük be a 0-st

A lépések ugyanezek 1-es betétele esetén is.

Tekintsünk meg egy példát is: rajzoljuk fel a 011010 bináris fágát!



6.1. ábra. A 011010 bináris fája

6.3. Fabejárás



Tutorált

Ebben a feladatban tutorált Duszka Ákos Attila.

Megoldás forrása: <https://github.com/krook1024/textbook/tree/master/files/welch-fabejaras>

Preorder bejárás esetén a gyökérrel kezdünk, majd bejárjuk a baloldali részfát, azt követően pedig a jobboldali részfát.

Postorder bejárás esetén pedig szintén a gyökérrel kezdünk, viszont a jobboldali részfával folytatjuk, és a baloldali részfával végzünk.

Inorder bejárás esetén a bal oldali részfát követően bejárjuk a gyökeret, majd a jobboldali részfát.

A program futtatása mostmár a következőképpen néz ki: `./binfa befile -o kifle [o/r]` ahol az o postorder, az r pedig preorder bejárást jelent. De mindez a Makefile tartalmazza, tehát az olvasó könnyen és egyszerűen kipróbálhatja és tanulmányozhatja a programot.

```
=====PREORDER BEJARAS=====
-----0(5)
-----0(4)
-----1(5)
-----0(3)
-----0(5)
-----1(4)
-----0(6)
-----1(5)
-----1(6)
-----0(2)
-----0(5)
-----0(4)
-----1(3)
-----0(6)
-----0(5)
-----1(4)
-----1(5)
---/(1)
-----0(3)
-----1(4)
-----1(2)
-----0(5)
-----0(4)
-----1(3)
```

```
-----1(4)
depth = 5
mean = 4.09091
var = 0.700649
=====POSTORDER BEJARAS=====
-----1(4)
-----1(3)
-----0(4)
-----0(5)
----1(2)
----1(4)
----0(3)
---/(1)
-----1(5)
-----1(4)
-----0(5)
-----0(6)
----1(3)
----0(4)
----0(5)
----0(2)
-----1(6)
-----1(5)
-----0(6)
-----1(4)
-----0(5)
----0(3)
----1(5)
----0(4)
----0(5)
depth = 5
mean = 4.09091
var = 0.700649
```

6.4. Tag a gyökér

Megoldás forrása: [gh/krook1024/textbook/master/files/welch/z3a7.cpp](https://github.com/krook1024/textbook/master/files/welch/z3a7.cpp)

Ebben az esetben felhasználjuk a már megírt LZWBinFa programot, mely eleve úgy lett megírva, hogy a gyökér kompozícióban van a fával.

6.5. Mutató a gyökér

Megoldás forrása: github.com/krook1024/Prog1vedes

Ebben a programban úgy írtuk át a már meglévő BinFa programunkat, hogy a bináris fánk gyökere mutató legyen.

A módosítások elvégzése egyszerű, mechanikus folyamat. A gyökér csomópont definíójánál foglalunk helyet a heapen a gyökérnek, majd az ebből kapott mutatót behelyettesítjük az olyan helykre, ahol egyébként a gyökér változóra hivatkoztunk. Ez a gyakorlatban annyit jelent, hogy eltávolítunk a kódból néhány címképző operátort (&).

6.6. Mozgató szemantika



Tutorált

Ebben a feladatban tutoráltam Rácz Andrást!

Megoldás forrása: <https://github.com/krook1024/textbook/blob/master/files/welch-mozgato/z3a8.cpp>

A mozgató szemantika alatt másoló-, és mozgató-konstruktorkat és a destruktort értjük. A bináris fák másolásához és mozgatásához rekurzív mozgató, és másoló függvényekre lesz szükségünk. Lássuk is ezeket.

```
Csomopont * masol ( Csomopont * elem, Csomopont * regifa ) {
    Csomopont * ujelem = NULL;
    if ( elem != NULL ) {
        switch (elem->getBetu()) {
            case '/':
                ujelem = new Csomopont ( '/' );
                break;
            case '0':
                ujelem = new Csomopont ( '1' );
                break;
            case '1':
                ujelem = new Csomopont ( '0' );
                break;
            default:
                std::cerr<<"HIBA!"<<std::endl;
                break;
        }
        ujelem->ujEgyesGyermekek(
            masol(elem->egyesGyermekek(), regifa)
        );
        ujelem->ujNullasGyermekek(
            masol(elem->>nullasGyermekek(), regifa)
        );
    }
}
```

```
    if ( regifa == elem )
        fa = ujelem;
    }
    return ujelem;
}
```

Az LZWBInFa osztályunk úgy épül fel, hogy az LZWBInFa osztályon belül megtalálhatóak beágyazott Csomopont osztályú objektumok, ezek alkotják a fát. Ebből következik, hogy a fát úgy tudjuk másolni, hogy ezeket a beágyazott csomópontokat másoljuk, rekurzívan.

A fenti kódcsipet a rekurzív másoló függvényünk, amit a másoló konstruktorból hívunk meg, olyan módon, hogy átadjuk neki a gyökeret és magát a fát.

```
LZWBInFa ( const LZWBInFa & regi ) {
    std::cout << "LZWBInFa copy ctor" << std::endl;
    gyoker = masol ( regi.gyoker, regi.fa ) ;
}
```

A másolókonstruktort úgy teszteljük, hogy nem referencia szerint adjuk át egy egyszerű függvénynek a fánkat, hanem érték szerint, ilyenkor a fa másolódik. Ha kipróbáljuk a programot akkor láthatjuk, hogy a másolókonstruktornak működik.

```
$ make
LZWBInFa copy ctor
-----0(3)
-----0(2)
-----0(1)
---/(0)
-----0(2)
-----1(1)
depth = 3
mean = 2.5
var = 0.707107
-----1(3)
-----1(2)
-----1(1)
---/(0)
-----1(2)
-----0(1)
depth = 3
mean = 2.5
var = 0.707107
```

Ennek alapjára most készítsük el a mozgatókonstruktort is. Ez alatt azt értjük, hogy felül fogjuk terhelni a '=' operátort (operator overloading).

```
LZWBinFa & operator=(LZWBinFa & regi) {
    std::cout << "mozgato ctor" << std::endl;
    this->gyoker = regi.gyoker;
    this->fa = regi.fa;
    return *this;
}
```

```
LZWBinFa copy ctor
-----0(3)
-----0(2)
-----1(3)
-----1(4)
-----0(1)
-----0(3)
-----1(2)
---/(0)
-----0(4)
-----0(3)
-----1(4)
-----1(5)
-----0(2)
-----1(3)
-----1(4)
-----1(1)
-----0(5)
-----0(4)
-----1(5)
-----0(3)
-----1(4)
-----1(2)
-----0(4)
-----1(3)
-----1(4)
depth = 5
mean = 4.09091
var = 0.700649
mozgato ctor
-----1(3)
-----1(2)
-----0(3)
-----0(4)
-----1(1)
-----1(3)
-----0(2)
---/(0)
-----1(4)
-----1(3)
-----0(4)
```

```
-->-->-->-->-->0(5)
-->-->-->-->1(2)
-->-->-->-->0(3)
-->-->-->-->0(4)
-->-->-->0(1)
-->-->-->1(5)
-->-->-->1(4)
-->-->-->0(5)
-->-->-->1(3)
-->-->-->0(4)
-->-->-->0(2)
-->-->-->1(4)
-->-->-->0(3)
-->-->-->0(4)
depth = 5
mean = 4.09091
var = 0.700649
-->-->-->1(3)
-->-->-->1(2)
-->-->-->0(3)
-->-->-->0(4)
-->-->1(1)
-->-->-->1(3)
-->-->-->0(2)
-->-->/(<0>)
-->-->-->1(4)
-->-->-->1(3)
-->-->-->0(4)
-->-->-->0(5)
-->-->1(2)
-->-->0(3)
-->-->-->0(4)
-->-->0(1)
-->-->-->1(5)
-->-->-->1(4)
-->-->-->0(5)
-->-->1(3)
-->-->-->0(4)
-->-->0(2)
-->-->-->1(4)
-->-->-->0(3)
-->-->-->0(4)
depth = 5
mean = 4.09091
var = 0.700649
```

A kimenetünkön megjelennek mind a "mozgato ctor" és "copy ctor" üzenetek, melyeket indikátorokként ültettünk el a forráskódban, hogy láthatunk, működnek-e.

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk



Tutorált

Ebben a feladatban tutorált Rácz András.

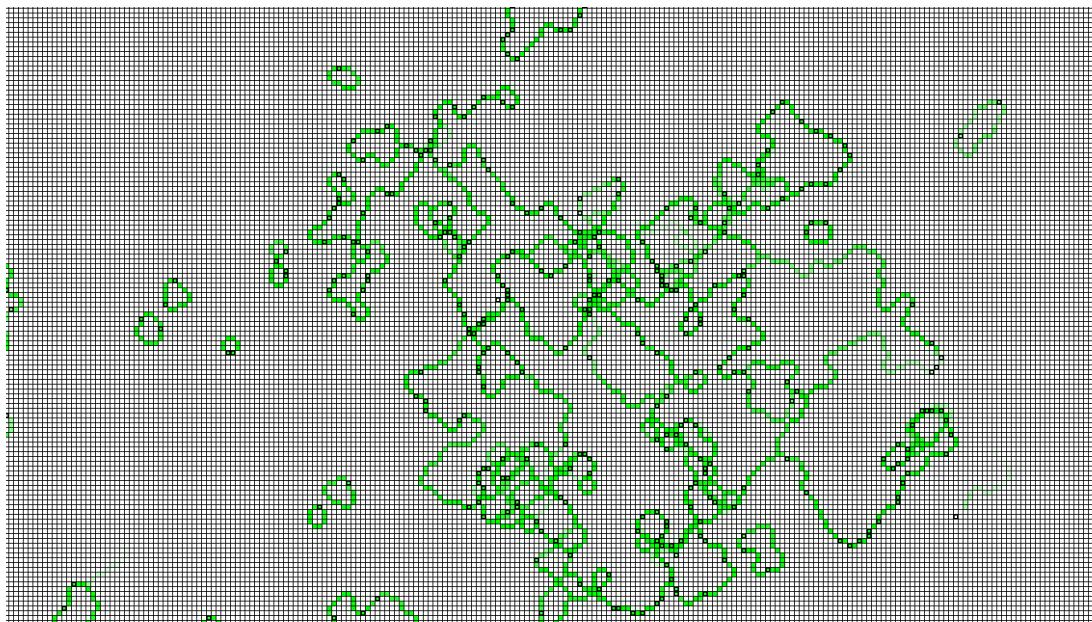
Írj Qt C++-ban egy hangyaszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Megoldás forrása: <https://github.com/krook1024/textbook/tree/master/files/conway/hangya>

Tudjuk, hogy ahogy a hangyák közlekednek, úgy feromonnyomokat hagynak maguk után, és így útvonalakat hoznak létre. Ezekben a nyomvonalakon később ők és társaik közlekednek, így találnak haza, vagy éppen élelemlelőhelyekre.

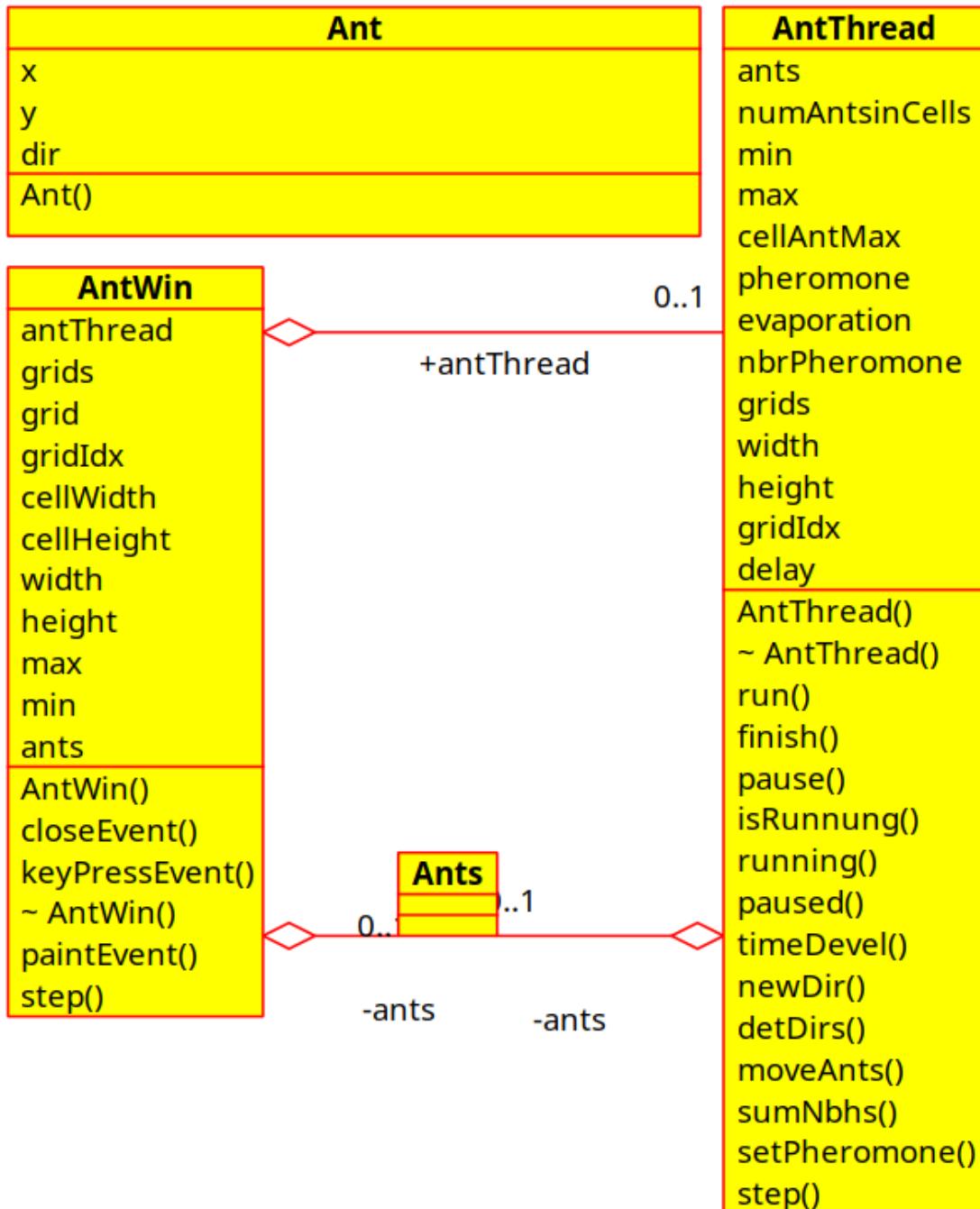
Ezeket próbáljuk meg szimulálni ebben a programban. Az olvasó kényelméért egy run szkriptet létrehoztam, így a program egyszerűen, egy parancs kiadásával futtatható (Qt telepítés szükséges).

```
$ ./run
```



7.1. ábra. Hangyaszimulációk futás közben

Az UML osztálydiagram általában reverse engineeringnél jön számításba, hiszen az eddig megtudott információkat jól lehet ábrázolni rajta. Viszont ennél a tulajdonságánál fogva könnyítheti a megértést (pláne ha a felesleges, figyelemterelő adatokat elrejtjük, ilyenkor úgy sem kell), ha egy olyan programhoz generálunk osztálydiagrammot, aminek a forráskódját ismerjük. Ezt tesszük mi most.



7.2. ábra. Hangyaszimulációk UML osztálydiagram

Az osztálydiagrammot legenerálhatjuk például az Umbrello nevű programmal, ami része a KDE családnak. A projekt weboldala elérhető itt: <https://umbrello.kde.org/installation.php>. Erről az ábráról leolvashatóak a legfontosabb adatok; osztályok, és azokba beágyazott függvények, konstruktorok, destrukturátorok, adattagok, stb.

7.2. Java életjáték

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

Megoldás forrása:

```
/*
 * Sejtautomata.java
 *
 * DIGIT 2005, Javat tanítok
 * Batfai Norbert, nbatfai@inf.unideb.hu
 *
 */
/*
 * Sejtautomata osztály.
 *
 * @author Batfai Norbert, nbatfai@inf.unideb.hu
 * @version 0.0.1
 */
public class Sejtautomata extends java.awt.Frame implements Runnable {
    /* Egy sejt lehet elő */
    public static final boolean EL0 = true;
    /* vagy halott */
    public static final boolean HALOTT = false;
    /* Két racsot használunk majd, az egyik a sejtter állapotat
     * a t_n, a másik a t_n+1 időpillanatban jellemzi. */
    protected boolean[][][] racsok = new boolean [2][][];
    /* Valamelyik racsra mutat, technikai jellegű, hogy ne kelljen a
     * [2][][]-bol az első dimenziót használni, mert vagy az egyikre
     * állítjuk, vagy a másikra. */
    protected boolean [][] racs;
    /* Megmutatja melyik racs az aktuális: [racsIndex][][] */
    protected int racsIndex = 0;
    /* Pixelben egy cella adatai. */
    protected int cellaSzelesseg = 20;
    protected int cellaMagassag = 20;
    /* A sejtter nagysága, azaz hanyszor hany cella van? */
    protected int szelesseg = 20;
    protected int magassag = 10;
    /* A sejtter két egymást követő t_n és t_n+1 diszkret időpillanata
     * közötti valós idő. */
    protected int varakozas = 1000;
    // Pillanatfelvétel készítésehez
    private java.awt.Robot robot;
    /* Készítünk pillanatfelvételt? */
    private boolean pillanatfelvétel = false;
    /* A pillanatfelvetelek számozásához. */
    private static int pillanatfelvételSzamlalo = 0;
    /*
```

```
* Letrehoz egy <code>Sejtautomata</code> objektumot.
*
* @param      szelesseg    a sejtter szelessege.
* @param      magassag     a sejtter szelessege.
*/
public Sejtautomata(int szelesseg, int magassag) {
    this.szelesseg = szelesseg;
    this.magassag = magassag;
    // A ket racs elkeszitese
    racsok[0] = new boolean[magassag][szelesseg];
    racsok[1] = new boolean[magassag][szelesseg];
    racsIndex = 0;
    racs = racsok[racsIndex];
    // A kiindulo racs minden cellaja HALOTT
    for(int i=0; i<racs.length; ++i)
        for(int j=0; j<racs[0].length; ++j)
            racs[i][j] = HALOTT;
    // A kiindulo racsra "elolenyeket" helyezunk
    //siklo(racs, 2, 2);
    sikloKilovo(racs, 5, 60);
    // Az ablak bezarasakor kilepünk a programból.
    addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosing(java.awt.event.WindowEvent e) {
            setVisible(false);
            System.exit(0);
        }
    });
    // A billentyűzetrol erkező események feldolgozása
    addKeyListener(new java.awt.event.KeyAdapter() {
        // Az 'k', 'n', 'l', 'g' és 's' gombok lenyomását figyeljük
        public void keyPressed(java.awt.event.KeyEvent e) {
            if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K) {
                // Felezük a cella méreteit:
                cellaSzelesseg /= 2;
                cellaMagassag /= 2;
                setSize(Sejtautomata.this.szelesseg*cellaSzelesseg,
                        Sejtautomata.this.magassag*cellaMagassag);
                validate();
            } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {
                // Duplazzuk a cella méreteit:
                cellaSzelesseg *= 2;
                cellaMagassag *= 2;
                setSize(Sejtautomata.this.szelesseg*cellaSzelesseg,
                        Sejtautomata.this.magassag*cellaMagassag);
                validate();
            } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
                pillanatfelvetel = !pillanatfelvetel;
            else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
                varakozas /= 2;
            else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L)
```

```
        varakozas *= 2;
        repaint();
    }
});

// Eger kattinto esemenyek feldolgozasa:
addMouseListener(new java.awt.event.MouseAdapter() {
    // Eger kattintassal jeloljuk ki a nagyitando teruletet
    // bal felso sarkat vagy ugyancsak eger kattintassal
    // vizsgaljuk egy adott pont iteracioit:
    public void mousePressed(java.awt.event.MouseEvent m) {
        // Az egermutato pozicioja
        int x = m.getX()/cellaSzelesseg;
        int y = m.getY()/cellaMagassag;
        racsok[racsIndex][y][x] = !rcsok[racsIndex][y][x];
        repaint();
    }
});
// Eger mozgas esemenyek feldolgozasa:
addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
    // Vonszolossal jeloljuk ki a negyzetet:
    public void mouseDragged(java.awt.event.MouseEvent m) {
        int x = m.getX()/cellaSzelesseg;
        int y = m.getY()/cellaMagassag;
        racsok[racsIndex][y][x] = EL0;
        repaint();
    }
});
// Cellameretek kezdetben
cellaSzelesseg = 10;
cellaMagassag = 10;
// Pillanatfelvetel keszitesehez:
try {
    robot = new java.awt.Robot(
        java.awt.GraphicsEnvironment.
        getLocalGraphicsEnvironment().
        getDefaultScreenDevice());
} catch(java.awt.AWTException e) {
    e.printStackTrace();
}
// A program ablakanak adatai:
setTitle("Sejtautomata");
setResizable(false);
setSize(szelesseg*cellaSzelesseg,
         magassag*cellaMagassag);
setVisible(true);
// A sejtter eletrekeltese:
new Thread(this).start();
}
/* A sejtter kirajzolasa. */
public void paint(java.awt.Graphics g) {
```

```
// Az aktualis
boolean [][] racs = racsok[racsIndex];
// racsot rajzoljuk ki:
for(int i=0; i<racs.length; ++i) { // vegig leped a sorokon
    for(int j=0; j<racs[0].length; ++j) { // s az oszlopok
        // Sejt cella kirajzolasa
        if(racs[i][j] == EL0)
            g.setColor(java.awt.Color.BLACK);
        else
            g.setColor(java.awt.Color.WHITE);
        g.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                   cellaSzelesseg, cellaMagassag);
        // Racs kirajzolasa
        g.setColor(java.awt.Color.LIGHT_GRAY);
        g.drawRect(j*cellaSzelesseg, i*cellaMagassag,
                   cellaSzelesseg, cellaMagassag);
    }
}
// Készítünk pillanatfelvetelt?
if(pillanatfelvetel) {
    // a biztonság kedveirt egy kép készítése után
    // kikapcsoljuk a pillanatfelvetelt, hogy a
    // programmal ismerkedő olvasó ne irja tele a
    // fájlrendszeret a pillanatfelvetelekkel
    pillanatfelvetel = false;
    pillanatfelvetel(robot.createScreenCapture
                      (new java.awt.Rectangle
                        (getLocation().x, getLocation().y,
                         szelesseg*cellaSzelesseg,
                         magassag*cellaMagassag)));
}
/*
 * Az kérdezett állapotban lévő nyolcszomszedök száma.
 *
 * @param racs a sejtter racs
 * @param sor a racs vizsgált sora
 * @param oszlop a racs vizsgált oszlopa
 * @param állapot a nyolcszomszedök vizsgált állapota
 * @return int a kérdezett állapotbeli nyolcszomszedök száma.
 */
public int szomszedokSzama(boolean [][] racs,
                            int sor, int oszlop, boolean állapot) {
    int állapotuSzomszed = 0;
    // A nyolcszomszedök vegizongorázása:
    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)
            // A vizsgált sejtet magát kihagyva:
            if(((i==0) && (j==0))) {
                // A sejtterbol szelenek szomszedai
```

```
// a szembe oldalakon ("periodikus hatarfelteset")
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;
}
/*
 * A sejtter idobeli fejlodese a John H. Conway fele
 * eletjatek sejtautomata szabalyai alapjan tortenik.
 * A szabalyok reszletes ismerteteset lasd peldaul a
 * [MATEK JATEK] hivatkozasban (Csakany Bela: Diszkret
 * matematikai jatekok. Polygon, Szeged 1998. 171. oldal.)
 */
public void idoFejlodes() {

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

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

            int elok = szomszedokSzama(racsElotte, i, j, EL0);

            if(racsElotte[i][j] == EL0) {
                /* Elo elo marad, ha ketto vagy harom elo
                 * szomszedja van, kulonben halott lesz. */
                if(elok==2 || elok==3)
                    racsUtana[i][j] = EL0;
                else
                    racsUtana[i][j] = HALOTT;
            } else {
                /* Halott halott marad, ha harom elo
                 * szomszedja van, kulonben elo lesz. */
                if(elok==3)
                    racsUtana[i][j] = EL0;
                else
                    racsUtana[i][j] = HALOTT;
            }
        }
    }
}
```

```
        }
    }
    racsIndex = (racsIndex+1)%2;
}
/* A sejtter idobel fejlodese.*/
public void run() {

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

        idoFejlodes();
        repaint();
    }
}
/*
 * A sejtterbe "elolenyeket" helyezunk, ez a "siklo".
 * Adott irányban halad, masolja magat a sejtterben.
 * Az eloleny ismerteteset lasd peldaül a
 * [MATEK JATEK] hivatkozasban (Csakany Bela: Diszkret
 * matematikai jatekok. Polygon, Szeged 1998. 172. oldal.)
 *
 * @param racs    a sejtter ahova ezt az allatkat helyezzuk
 * @param x       a befoglalo tegla bal felső sarkanak oszlopa
 * @param y       a befoglalo tegla bal felső sarkanak sora
 */
public void siklo(boolean [][][] racs, int x, int y) {

    racs[y+ 0][x+ 2] = EL0;
    racs[y+ 1][x+ 1] = EL0;
    racs[y+ 2][x+ 1] = EL0;
    racs[y+ 2][x+ 2] = EL0;
    racs[y+ 2][x+ 3] = EL0;

}
/*
 * A sejtterbe "elolenyeket" helyezunk, ez a "siklo agyu".
 * Adott irányban siklokat lo ki.
 * Az eloleny ismerteteset lasd peldaül a
 * [MATEK JATEK] hivatkozasban /Csakany Bela: Diszkret
 * matematikai jatekok. Polygon, Szeged 1998. 173. oldal./,
 * de itt az abra hibas, egy oszloppal told meg balra a
 * bal oldali 4 sejtes negyzetet. A helyes agyu rajzat
 * lasd pl. az [ELET CIKK] hivatkozasban /Robert T.
 * Wainwright: Life is Universal./ (Megemlíthetjük, hogy
 * mindenketto tartalmaz ket felesleges sejtet is.)
 *
 * @param racs    a sejtter ahova ezt az allatkat helyezzuk
```

```
* @param x      a befoglalo tegla bal felső sarkanak oszlopa
* @param y      a befoglalo tegla bal felső sarkanak sora
*/
public void sikloKilovo(boolean[][][] racs, int x, int y) {
    racs[y+ 6][x+ 0] = ELO;
    racs[y+ 6][x+ 1] = ELO;
    racs[y+ 7][x+ 0] = ELO;
    racs[y+ 7][x+ 1] = ELO;

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

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

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

    racs[y+ 6][x+ 11] = ELO;
    racs[y+ 6][x+ 15] = ELO;
    racs[y+ 6][x+ 16] = ELO;
    racs[y+ 6][x+ 22] = ELO;
    racs[y+ 6][x+ 23] = ELO;
    racs[y+ 6][x+ 24] = ELO;
    racs[y+ 6][x+ 25] = ELO;

    racs[y+ 7][x+ 11] = ELO;
    racs[y+ 7][x+ 15] = ELO;
    racs[y+ 7][x+ 16] = ELO;
    racs[y+ 7][x+ 21] = ELO;
    racs[y+ 7][x+ 22] = ELO;
    racs[y+ 7][x+ 23] = ELO;
    racs[y+ 7][x+ 24] = ELO;

    racs[y+ 8][x+ 12] = ELO;
    racs[y+ 8][x+ 14] = ELO;
    racs[y+ 8][x+ 21] = ELO;
    racs[y+ 8][x+ 24] = ELO;
    racs[y+ 8][x+ 34] = ELO;
    racs[y+ 8][x+ 35] = ELO;

    racs[y+ 9][x+ 13] = ELO;
    racs[y+ 9][x+ 21] = ELO;
    racs[y+ 9][x+ 22] = ELO;
    racs[y+ 9][x+ 23] = ELO;
    racs[y+ 9][x+ 24] = ELO;
    racs[y+ 9][x+ 34] = ELO;
    racs[y+ 9][x+ 35] = ELO;
```

```
racs[y+ 10][x+ 22] = EL0;
racs[y+ 10][x+ 23] = EL0;
racs[y+ 10][x+ 24] = EL0;
racs[y+ 10][x+ 25] = EL0;

racs[y+ 11][x+ 25] = EL0;

}

/* Pillanatfelvetelek készítése. */
public void pillanatfelvetel(java.awt.image.BufferedImage felvetel) {
    // A pillanatfelvetel kép fajlneve
    StringBuffer sb = new StringBuffer();
    sb = sb.delete(0, sb.length());
    sb.append("sejtautomata");
    sb.append(++pillanatfelvetelSzamlalo);
    sb.append(".png");
    // png formátumú képet mentünk
    try {
        javax.imageio.ImageIO.write(felvetel, "png",
            new java.io.File(sb.toString()));
    } catch(java.io.IOException e) {
        e.printStackTrace();
    }
}
// Ne villogjon a felület (mert a "gyári" update()
// lemeszelne a vaszon felületet).
public void update(java.awt.Graphics g) {
    paint(g);
}
/*
 * Peldanyosít egy Conway-féle életjatek szabalyos
 * sejtter objektumot.
 */
public static void main(String[] args) {
    // 100 oszlop, 75 sor merettel:
    new Sejtautomata(100, 75);
}
```



Figyelem! Deprecated szoftver!

A program a már deprecated AWT grafikus függvénykönyvtárat használja, melyet már a mai napon nem használnánk. Helyette próbálunk meg valami korszerűbbet, például OpenJFX-et használni!

A program futtatására az olvasó kényelmét szolgálálandó létrehoztam egy run szkriptet, ezzel egy parancs kiadásával futtatható a program.

A programot futtatva láthatjuk a sikló-kilövőt.

7.3. Qt C++ életjáték

Megoldás forrása: <https://github.com/krook1024/textbook/tree/master/files/conway-Sejtauto>

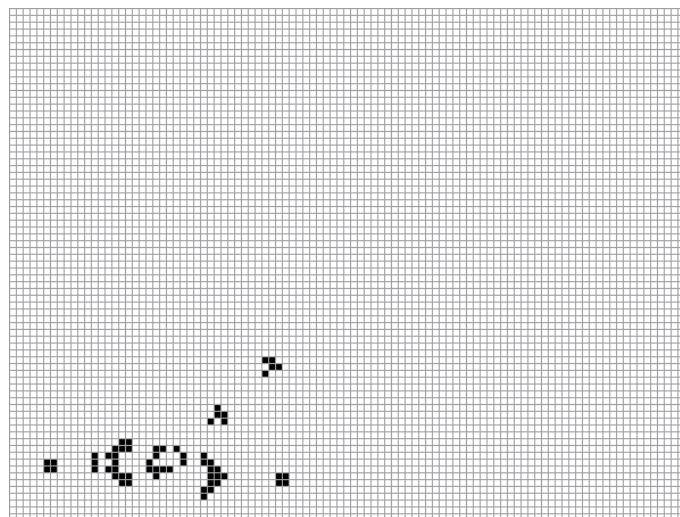
Az életjáték bármelyik vírusnál nagyobb bénulást okozott az informatikában az 1970-es években. mindenki ezzel akart játszani, nézni a monitorokon a program kimenetét, ezzel már-már bénulást okozva az akkori számítógépes redszereknek, melyet időosztással lehetett igénybe venni, elsősorban kutatási célokra.

A John Conway-féle életjátékban nagyon egyszerű szabályok uralkodnak:

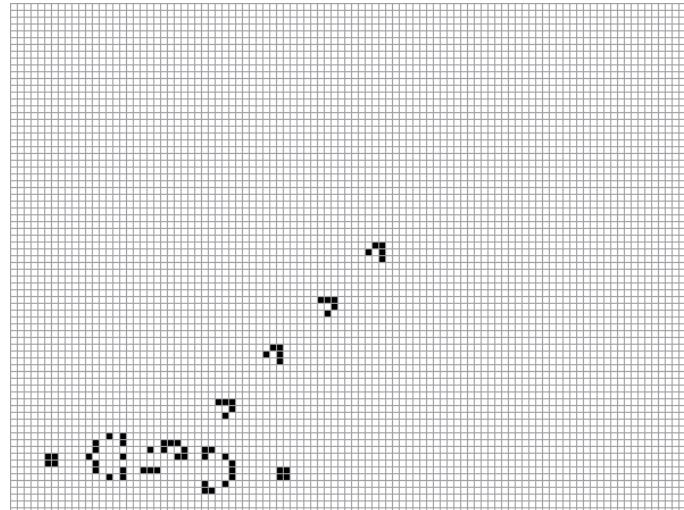
- minden sejt amelynek két vagy három szomszédja van, életben marad
- minden sejt amelynek négy vagy több szomszédja van, meghal
- minden sejt amelynek kevesebb mint kettő szomszédja van, meghal
- ha egy üres cellának pontosan három élő sejt vesz körül, ott új sejt szülteik

A Qt már ismerős lehet, hiszen már a Mandelbrotos fejezetünkben előkerült. Most ismét felélesztjük, és a John H. Conway féle életjátékot fogjuk kipróbálni.

Szokás szerint egyszerűen a mappába lépve a `./run` parancsot kiadva le fog fordulni és futtatva lesz a program. Futás közbeni képeket alább találhat meg az olvasó.

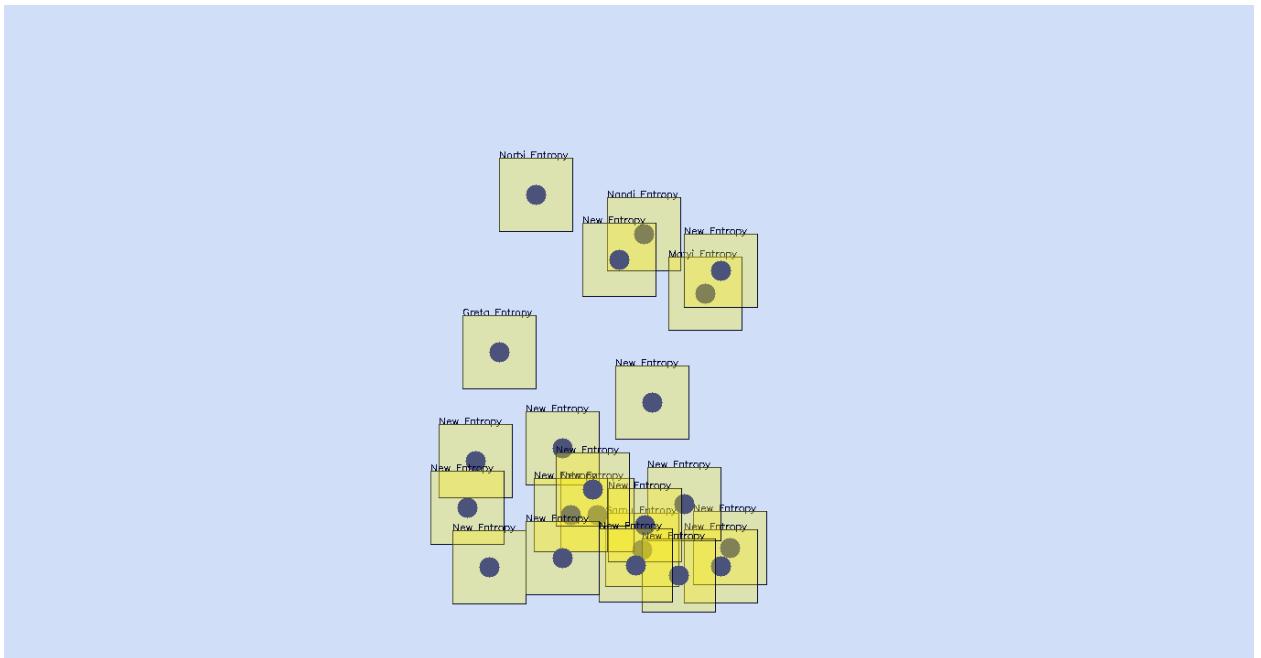


7.3. ábra. Qt C++ életjáték



7.4. ábra. Qt C++ életjáték

7.4. BrainB Benchmark



7.5. ábra. BrainB Benchmark

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

A játék célja hogy pár perc játékidő után az egy E-Sport "profilt" állítson elő a játékosról, próbálja megjósolni, hogy az adott játékosból mennyire lehet jó E-sportoló.

A játék során a lényeg hogy a Samu Entropy entrópia közepén megjelenő fekete pöttyön tartsuk az egeret, ameddig csak tudjuk. Ez az entrópia a játék során folyamatosan mozog, zizeg, alábújik más entrópiáknak, ezzel nehezítve a dolgunkat.

A programot kipróbálhatja bárki a következő parancsok kiadásával: (OpenCV-re szükségünk lesz)

```
$ git clone https://github.com/nbatfai/esport-talent-search
$ cd esport-talent-search
$ mkdir build && cd build
$ qmake ..
$ make
$ ./BrainB
```

Az én eredményem 3 perc játék után a következő lett:

```
NEMESPOR BrainB Test 6.0.3
time      : 1809
bps       : 21720
noc       : 22
nop       : 0
lost      :
45230 44090 15600 12730 82840 49690 63660 86510 63010 43770 32800 18760 ←
 19120 10990 14510 66850 48580 56320 35800 54540 21200 56470 35880 36880 ←
 29050
mean      : 41795
var       : 21398.8
found     :
20170 19560 27830 49580 30150 50580 66080 22670 35600 5450 9730 ←
 15130 43400 33610 51300 56580 47370 48940 62560 57660 58400 45580 6920 ←
 15380 15440 13460 8590 25120 24650 29940 44950 50710 51920 48310 30030 ←
 48740 29900 37420 31020 35740 37290 46400
mean      : 35472
var       : 16558.8
lost2found: 22670 5450 47370 57660 6920 51920 30030 29900 31020
mean      : 31437
var       : 18435.2
found2lost: 44090 82840 63660 86510 66850 48580 56320 54540 56470
mean      : 62206
var       : 14492.7
mean(lost2found) < mean(found2lost)
time      : 3:0
U R about 5.71552 Kilobytes
```

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Megoldás forrása: https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol

A TensorFlow egy nyílt forráskódú szabad szoftver, amit a Google is használ, illetve fejleszt. A TensorFlow-val bárki egyszerűen végezhet mesterséges intelligenciához kapcsolódó kísérleteket, kutatásokat.

Most mi is pár ilyet nézünk meg. A fejezet olvasása előtt nézzük meg és familiarizáljuk magunkat a Bazel Build (<https://bazel.build/>) rendszerrel, mely kelleni fog nekünk a C++-os TensorFlow projektek fordításához.

Az első program amit megtekintünk Pythonban nem csinál más, csak összeszoroz két számot neurális hálók segítségével. Ezt úgy képzeljük el, mint C-ben a hello worldöt, tehát a legegyszerűbb programot, amit el tudunk készíteni. Lássuk is a programot!

```
#!/usr/bin/env python2
# TensorFlow Hello World 1!
# twicetwo.py
#
import tensorflow as tf

node1 = tf.constant(2)
node2 = tf.constant(2)

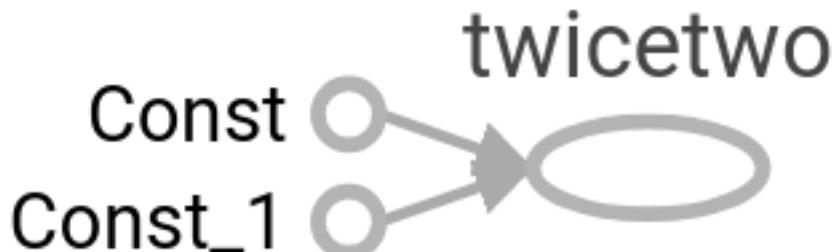
node_twicetwo = tf.math.multiply(node1, node2, name="twicetwo")

sess = tf.Session()
print sess.run(node_twicetwo)

writer = tf.summary.FileWriter("/tmp/twicetwo", sess.graph)
# nbafai@robopsy:~/Robopsychology/repos/tf/tf/tensorboard$ python ←
    tensorboard.py --logdir=/tmp/twicetwo
```

```
tf.train.write_graph(sess.graph_def, "models/", "twicetwo.pb", as_text= ←  
False)
```

A számítási ábra ehhez a következőképp néz ki:



8.1. ábra. 2*2 számítási ábra

Az **MNIST** egy kézírásfelismerő adatbázis, mely számok kézzel írt változatát képes felismerni.



8.2. ábra. Példák az MNIST adatbázisból

A kódot ehhez átvesszük a TensorFlow weboldaláról, íme: <https://www.tensorflow.org/tutorials>.

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

A programot futtatva a következőket látjuk:

```
$ ./mnist.py
Epoch 1/5
60000/60000 [=====] - 13s 224us/sample - loss: ←
  0.2188 - acc: 0.9359
Epoch 2/5
60000/60000 [=====] - 13s 224us/sample - loss: ←
  0.0956 - acc: 0.9702
Epoch 3/5
60000/60000 [=====] - 14s 226us/sample - loss: ←
  0.0678 - acc: 0.9783
Epoch 4/5
60000/60000 [=====] - 13s 224us/sample - loss: ←
  0.0536 - acc: 0.9829
Epoch 5/5
60000/60000 [=====] - 13s 224us/sample - loss: ←
  0.0457 - acc: 0.9853
10000/10000 [=====] - 1s 74us/sample - loss: ←
  0.0685 - acc: 0.9804
```

Ezzel, mint láthatjuk, egy körülbelül 98 százalékos pontossággal rendelkező hálózatot sikerült megtanítanunk.

8.2. Mély MNIST

Passzolt feladat



Ezt a feladatot a SMNIST-es kutatáshoz való hozzájárulásomra hivatkozva passzoltam.

Komment linkje: https://www.facebook.com/groups/udprog/permalink/1075942815926940/?comment_id=1080205895500632

8.3. Deep dream

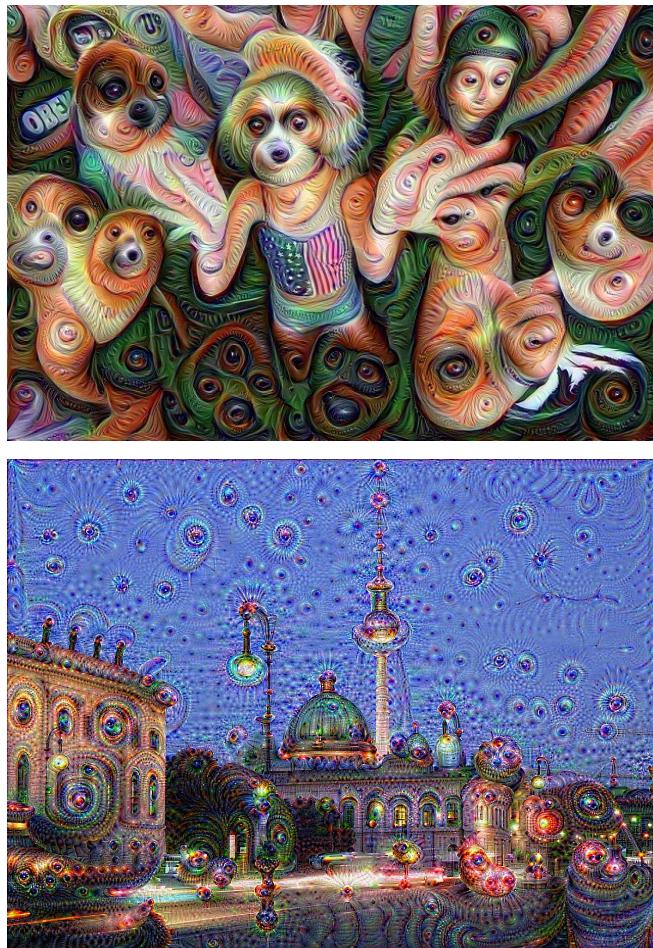
Megoldás forrása: https://github.com/keras-team/keras/blob/master/examples/deep_dream.ipynb

A Keras deep dreamben mindenféle kábítószer használata nélkül tekintehetünk meg olyan példákat, mint egy LSD trippen. A látvány egyszerre hátborzongató és különleges.

Ebben a fejezetben kitérnek a Python fejlesztés egyik kulcselemére, a virtuális környezeketre (`virtualenv`, `venv`). A virtuális környezetek előnye, hogy egy-egy program dependenciáit (könyvtárak, stb) nem kell globális, a root felhasználóval telepítenünk, elég azt egy virtuális környezetben telepíteni, a projekt mappáján belül. A legjobb az, hogy ezeket a letöltött dependenciákat nem kell a verziókezelőnkkel kezeltetni, elég ha berakjuk őket célszerűen egy `requirements.txt` fájlba, és bárki aki fejleszteni akarja a projektünket, vagy dolgozni rajta, telepítheti, anélkül hogy rendszergazdai jogosultságai lennének az adott gépen. Egy-egy könyvtár verziószámát is megadhatjuk, akár a minimálist, akár a maximálist abból.

Itt látható néhány deep dreammel generált kép. Forrás: <https://www.google.hu/search?q=deepdream>.



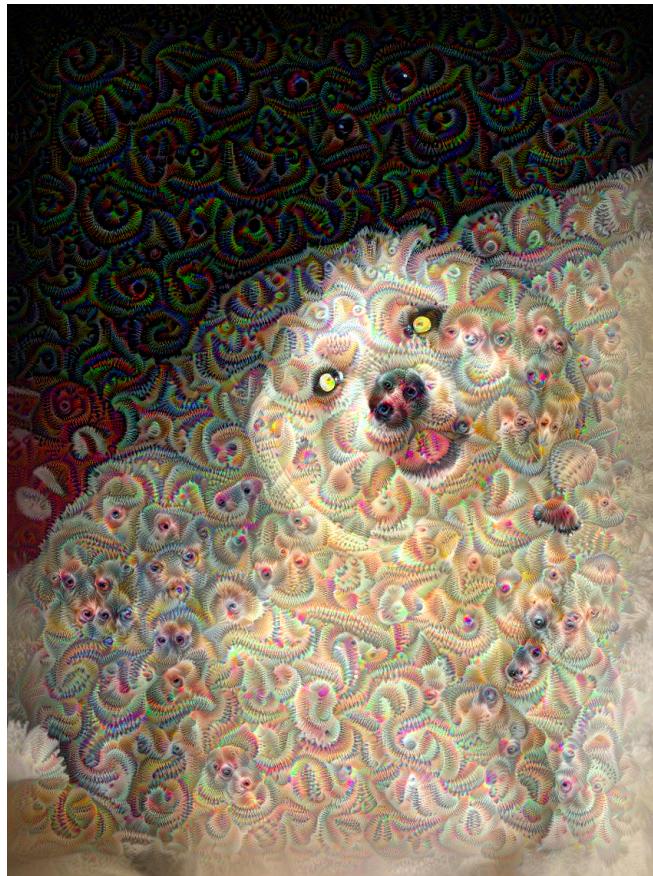


Ha ezek a képek elnyerték a tetszésünket, próbáljuk ki a deep dreamet mi magunk is, hiszen csinálhatunk ilyet a kedvenc képeinkből is.

```
$ virtualenv -p /usr/bin/python3 venv # Készítsük el a virtuális környezetet
Running virtualenv with interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /home/b1/Repos/textbook/files/svajci/deepd/venv/bin/python3
Also creating executable in /home/b1/Repos/textbook/files/svajci/deepd/venv/bin/python
Installing setuptools, pip, wheel...
done
$ source venv/bin/activate # Aktiváljuk a virtuális környezetet
(venv) $ pip3 install tensorflow keras pillow # Telepítsük a Keras-t
[...]
(venv) $ pip3 freeze > requirements.txt # Mentsük el a dependenciákat
[...]
```

Immáron használhatjuk is a deepdreamet, feltéve, hogy minden úgy ment, ahogy

mennie kell. Próbáljuk ki saját képekkel! Alább látható néhány az én próbálkozásaimból. A programkódot olvasva láthatjuk, hogy a beállítások tweakelhetők.





8.4. Minecraft-MALMÖ

A Minecraft-MALMÖ egy programozható ágens kiegészítő a Minecraft játékhoz. A projekt GitHub oldala a következő címen érhető el: <https://github.com/Microsoft/malmo>. Ezen a címen találunk példaprogramokat is, valamint remek dokumentációt.

A Malmö telepítéséhez végezzük el a repóban javasoltakat, mely némi időt igényelhet, hiszen a dependenciák közé tartozik a Boost C++ könyvtár, és ennek Python interfésze. Ha megadatik az az áldás, hogy Ubuntu 16.04 vagy 18.04-et futtatunk, úgy használjuk a repón belül a `scripts/build.sh` szkriptet, mely elvégzi helyettünk a piszkos munkát.

A Malmö előnye, hogy a Minecraftot nem kell megvegyük, hiszen tartalmaz egy bináris változatot, amit bárki használhat, akinek feltett szándéka, hogy ágenst programozzon Malmöben.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

A LISP a második legrégebbi magas szintű programozási nyelv a Fortran után. A használatához nem kell mást tennünk, mint telepíteni egy Lisp futtató programot, esetünkben, GNU/Linux alatt a `clisp` megteszi. A programokat majd `clisp` fájl név.`.lisp` módon kell futtatnunk.

A LISP érdekessége, hogy a kifejezéseket nem a megszokott, C-s módon, hanem prefix alakban kéri. Ez azt takarja, hogy például $(3 + 2)$ helyett azt írnánk, hogy $(+ 3 2)$.

Egy egyszerű LISP program a Hello, world! kiíratására így néz ki:

```
(write-line "Hello World")
```

Az iteratív faktoriális számoltatást a következőképp oldjuk meg:

```
(defun factorial (N)
  (let ((R 1))
    (do ((i 1 (+ i 1)) ((> i N) R)
         (setf r (* r i)))
        )
  )
  )

(print (factorial 5))
```

A rekurzív példa ettől egyszerűbb:

```
(defun factorial (N)
  (if (= N 1)
```

```
    1
    (* N (factorial (- N 1)))
  )
)

(print (factorial 5))
```

Természetesen mindenki program a következő eredményt produkálja:

```
$ clisp iter.lisp
120
$ clisp reku.lisp
120
```

Tehát a helyes választ, azaz $5! = 120$ -at kapjuk.

9.2. Weizenbaum Eliza programja

A programot Joseph Weizenbaum tervezte az MIT-n, 1966-ban, és a célja vele az volt, hogy a mai chatbotokhoz (pl. Cleverbot) hasonlót alkossan. A program úgy működik, hogy tartalmaz sok-sok szót, kifejezést, és ezekre mintaválaszokat, melybe behellyettesíti a user-től kapott inputot, ennek hatására úgy látszik, valódi emberrel beszélkedünk.

A programot úgy futtathatjuk, ha lehúzzuk a következő weboldalról: <http://norvig.com/paip/README.html>, majd a következő módon (abból a mappából, ahova kicsomagoltuk) elindítjuk:

```
$ clisp
> (load "auxfns.lisp")
> (load "eliza.lisp")
> (eliza)
ELIZA>
```

9.3. Gimp Scheme Script-fu: króm effekt

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

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

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_L1_Chrome

Tanulságok, tapasztalatok, magyarázat...

9.4. Gimp Scheme Script-fu: név mandala

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

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

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_L1_Mandala

Tanulságok, tapasztalatok, magyarázat...

10. fejezet

Helló, Guttenberg!

10.1. Juhász István - Magas szintű programozási nyelvek

Programozási nyelvek között megkülönböztetünk imperatív és deklaratív programozási nyelveket. Az imperatív nyelvekkel ez a könyv nem foglalkozik. A deklaratív nyelvek pedig a hagyományos nyelvek, amiket ismerünk, s látunk ebben a könyvben is.

Emelett különbséget teszünk alacsony-, és magasszintű programozási nyelvek között is. Alacsony szintű nyelv például az Assembly, mely ugyan ember számára olvasható, de csak nehezen értelmezhető. Vegyük egy Assembly helloworld példát.

```
org 100h
mov dx, msg
mov ah, 9
int 21h
mov ah, 4Ch
int 21h
msg db 'Hello, World!', 0Dh, 0Ah, '$'
```

Látható, hogy még C-ben néhány angol szó ismeretével el tudunk igazodni egy programon, Assemblyben rögtön nehezebb dolgunk lesz.

Egy magasszintű programozási nyelv attól magasszíntű, hogy sok úgynevezett absztraktiós rétegen megy át, míg gépi kód lesz belőle, amit a processzor értelmezni tud. Hagyományosan, mint például a C-ben is, amit mi, emberek írunk, azt úgy hívjuk, hogy forráskód, amiből a fordító csinál majd a számítógép számára értelmezhető kódot. Ahhoz, hogy a fordító ezt el tudja végezni, nekünk pontos nyelvtani szabályokat kell követnünk a programunk írása során.

A fordítóprogramok olyan programok, amelyek a magas szintű programozási nyelven megírt szöveget lefordítják gépi nyelvre. Eszközei a következők:

- lexikális elemzés,
- szintaktikai elemzés,
- szemantikai elemzés,
- kódgenerálás.

Minden program alapelemei a karakterek, ezekből állnak össze a bonyolultabb nyelvi elemek, mint a

- lexikális egységek,
- szintaktikai egységek,
- utasítások,
- programegységek,
- fordítási egységek,
- program.

A legtöbb programozási nyelvben megtalálható az *egész* típus. Néhány nyelv ismeri ezeknek az előjelezett, és előjel nélküli változatát is (C-ben `unsigned` minősítővel adható meg). Alapvetőek még a valós típusok, melyek belső ábrázolása lebegőpontos. Ezen kívül gyakran megtalálhatók a *karakter* és *karakterlánc* típusok is.

A nevesített konstans olyan eszköz, mely három komponensből áll, ezek: név, típus, érték. A lényege, hogy néha beszélő neveket adunk nekik, ezáltal könnyebben olvashatóvá téve a kódunkat. C-ben így néz ki egy nevesített konstans: `#define név érték`.

Alapelemek programozási nyelvekben

A C nyelv vonatkozásában a következőket lehet elmondani.

- aritmetikai típusok
 - egész (int)
 - karakter (char)
 - valós (float, double)
- származtatott típusok
 - tömb
 - függvény
 - mutató
 - struktúra
 - union

- `void` típus

Az eljárásorientált nyelvekben a programot felbonthatjuk úgynevezett programalegységekre. Ezeket bizonyos programozási nyelvekben együtt kell fordítani, míg másokban, köztük C-ben is, külön-külön fordíthatóak, és később összekapcsolhatóak egy úgynevezett *linker* segítségével.

Változók hatásköréről is beszélnünk kell. A hatáskör egy változóra vonatkozik, és azt mondja meg, hogy a változó értéke honnan érhető el. A hatáskör minden befelé terjed, kifelé soha. Vegyük például az alábbi C programot:

```
#include <stdio.h>
int a = 1;
int main(void) {
    int b = 3;
    {
        int c = 4;
        printf("%d\n", c);
    }
    printf("%d\n", b);
    for(int i=0; i < 1; i++)
    {
        printf("%d\n", i);
    }
    printf("%d\n", i);
    return 1;
}
```

Mit gondolunk, mi lesz a program kimenete? Lefordul egyáltalán? Próbáljuk elemzni a változók hatáskörét, hogy melyik változó milyen scopeból látszik.

10.2. Kernighan & Ritchie - A C programozási nyelv

Alapismeretek

A könyv a Hello, World! C változatával indít, ami a könyvben a következőképp néz ki;

```
#include <stdio.h>

main() {
    printf("Hello, World!\n");
}
```

```
$ cc hw.c
hw.c:3:1: warning: return type defaults to 'int' [-Wimplicit-int]
 main() {
 ^
$
```

Aki nem először lát C kódot annak feltűnhet, hogy milyen önző programozó írhatta ezt a kódot, aki nem hajlandó a rendszernek visszaadni egy 0-t a kód futtatásáért cserébe, de talán lépjünk is ezen tovább.

Az első fejezetben a könyv a legalapvetőbb C utasításokon megy végig. Ismerteti a C-ben használatos típusokat, változókezelést, ciklusokat (`for`, `while`, `do-while`), I/O utasításokat, tömböket, függvényeket, függvények használatát, argumentumkezelést, karaktertömböket (sztringeket), és a változók érvényességét, avagy milyen scope-ban milyen változó látszik. Egyszóval tényleg az abszolút minimumot.

Típusok, operátorok, kifejezések

C-ben az alapvető típusok a `char`, `int`, `float`, `double`. Ezekhez hozzájöhetsz úgynevezett minősítők, mint a `short`, `long`. C-ben lehetőségünk van arra, hogy saját típusokat definiálunk, ezt megtehetjük a `typedef` utasítással. Egész állandókat megadhatunk 1234 formában, ez minden külön jelzés nélkül `int` típusú. Néhány különleges karaktert úgynevezett *escape stringekkel* adhatunk meg, mint például az újsor karakter (`\n`), ezek minden backslashtel kezdődnek. C-ben változók deklarálása a következőképp néz ki (példa): `int a, b =4, c =b;char c, sor[1000];`. Tehát a változó típusával kezdünk, majd megadjuk a nevét, és opcionálisan kezdőértéket adunk neki. Amennyiben szögletes zárójelek között (`[]`) megadunk egy számot, úgy tömböt deklarálunk, és a szám a tömb mérete lesz. C-ben a következő aritmetikai operátorok működnek: `% maradékos osztás, (+, -, *, /)` magyarázat nélkül. Emellett vannak logikai és relációs operátorok, melyek a következők: `>, >=, <, <=, ==, !=`. A típusok közötti átváltás nehézkes C-ben, nincsen egyszerű és elegáns mód arra, hogy például egy `int` típusú változát `char` tömbe változtassunk. Inkrementáló és dekrementáló operátorok azok az egyoperandusú operátorok, melyek egy változó értékét növelik, vagy csökkentik. Például: `++i`. Vannak még bitenkénti logikai operátorok, ezek úgy működnek, hogy logikai műveleteket (pl. konjunkció, diszjunkció) végeznek el a változó, vagy konstans bináris alakjával és egy másik változóval, vagy konstanssal. Például a következő kifejezés: `int b =3 & 4;`. Az olvasó feladata eldönteni, hogy a `b` változónak mennyi lesz az értéke.

Vezérlési szerkezetek

A 3.3-as fejezetben tárgyalva.

Függvények és a program szerkezete

Mutatók és tömbök

Struktúrák

Adatbevitel és adatkivitel

Kapcsolódás a UNIX op. rendszerhez

10.3. BME könyv

A könyvet még nem sikerült beszerezniem ezért az olvasónapló megírását halasztani vagyok kénytelen.

[OK] 1-16 [OK] 17-59 [OK] 93-96 [] 73-90 [] 187-197

A könyv a C++ és C összehasonlításával indít, de még nem tér rá a C++ OOP tulajdonságaira, csak az alapokat fekteti le, hogy miben lett más/jobb a C++ mint a C. A "veszélyes" dolgokat próbálták meg lecserélni jobbakra, biztonságosabbakra.

A könyv kitér arra, hogy mi változott a függvénydefiníciókban, a main függvény definícióira, típusokra, tehát az alapokra.

C++-ban lehetőségünk van függvénynevek túlterhelésére, nézzünk is erre egy példát a könyvből, mert ez egy nagyon érdekes dolog, ami a C++ előnye a C-vel szemben.

```
#include <time.h>
struct Time {
    int Hour;
    int Minute;
}
void PrintTime(struct Time time) { // 1-es változat
    printf("Hour: %d, Minute: %d", time.Hour, time.Minute);
}
void PrintTime(int hour, int minute) { // 2-es változat
    printf("Hour: %d, Minute: %d", time.Hour, time.Minute);
}
```

Itt azt láthatjuk, hogy egy függvénynévnek két definíciót is megadtunk, csak más paraméterlistával. Ezzel azt érjük el, hogy a PrintTime függvényünk hívható printTime(time) és printTime(9, 32) módokon egyaránt. Próbáljuk is ki!

```
int main() {
    Time time = { 3, 4 };
    printTime(time); // 1-es változatot hívja
    printTime(9, 32); // 2-es változatot hívja
    return 0;
}
```

Egy másik hasznos előnye a C++-nak a referenciaiák bevezetése. Ez lehetővé teszi a referencia szerinti paraméterátadást függvényeknek, ami azt jelenti, hogy van egy új, effektív módunk paraméterekek átadására. Amennyiben mutató szerint adjuk át a paramétereket, úgy a programunk nehezen olvashatóvá válik, amennyiben pedig érték szerint, úgy másolódnak a változók, és nem lesz effektív a programunk.

A következő fejezetben a könyv az objektumokról ír, osztályokról, láthatósági szabályokról, konstruktorkról, destruktorkról, valamint a friend osztályokról. Emellett szó esik beágyazott osztályokról, beágyazott függvényekről, melyek olyan osztályok és függvények, amelyek egy osztályon belül lettek definiálva.

III. rész

Második felvonás

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

Az előző feladat kódját fejleszd tovább: vizsgáld, hogy vannak-e Java osztályok a Pi hexadecimális kifejtésében!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

- [FOO] Eric Steven Raymond, *The Unix Koans of Master Foo*
[MARX] Marx György, *Gyorsuló idő*, Typotex , 2005.

11.4. C

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

11.5. C++

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

11.6. Lisp

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

Köszönet illeti az UDPORG tanulószoba, <https://www.facebook.com/groups/udprog> tagjaikat inspiráló érdeklődésükért és hasznos észrevételeikért.

COPYRIGHT 2019 Dr. BÁTFAI NORBERT

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>