

Univerzális programozás

Így neveld a programozód!

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, Bátfai, Margaréta, Ács Boda, Zsolt	2020. április 3.	

REVISION HISTORY

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

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket, sorozatokat nézzek meg?	2
II. Tematikus feladatok	4
2. Helló, Turing!	6
2.1. Végtelen ciklus	6
2.2. Lefagyott, nem fagyott, akkor most mi van?	9
2.3. Változók értékének felcserélése	12
2.4. Labdapattogás	14
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	17
2.6. Helló, Google!	20
2.7. A Monty Hall probléma	24
2.8. 100 éves a Brun téTEL	27
2.9. Vörös Pipacs Pokol/csiga (folytonos mozgási parancsokkal)	30
3. Helló, Chomsky!	32
3.1. Decimálisból unárisba átváltó Turing gép	32
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	34
3.3. Hivatalos nyelv	35
3.4. Saját lexikális elemző	37
3.5. Leetspeak	38

3.6. A források olvasása	41
3.7. Logikus	43
3.8. Deklaráció	43
3.9. Vörös Pipacs Pokol/csiga (diszkrét mozgási parancsokkal)	44
4. Helló, Caesar!	46
4.1. double ** háromszögmátrix	46
4.2. C EXOR titkosító	50
4.3. Java EXOR titkosító	51
4.4. C EXOR törő	53
4.5. Neurális OR, AND és EXOR kapu	56
4.6. Hiba-visszaterjesztéses perceptron	64
4.7. Vörös Pipacs Pokol/írd ki, mit lát Steve	66
5. Helló, Mandelbrot!	69
5.1. A Mandelbrot halmaz	69
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	73
5.3. Biomorfok	77
5.4. A Mandelbrot halmaz CUDA megvalósítása	81
5.5. Mandelbrot nagyító és utazó C++ nyelven	85
5.6. Mandelbrot nagyító és utazó Java nyelven	87
5.7. Vörös Pipacs Pokol/fel a láváig és vissza	92
6. Helló, Welch!	94
6.1. Első osztályom	94
6.2. LZW	94
6.3. Fabejárás	94
6.4. Tag a gyökér	94
6.5. Mutató a gyökér	95
6.6. Mozgató szemantika	95
7. Helló, Conway!	96
7.1. Hangyaszimulációk	96
7.2. Java életjáték	96
7.3. Qt C++ életjáték	96
7.4. BrainB Benchmark	97

8. Helló, Schwarzenegger!	98
8.1. Szoftmax Py MNIST	98
8.2. Mély MNIST	98
8.3. Minecraft-MALMÖ	98
9. Helló, Chaitin!	99
9.1. Iteratív és rekurzív faktoriális Lisp-ben	99
9.2. Gimp Scheme Script-fu: króm effekt	99
9.3. Gimp Scheme Script-fu: név mandala	99
10. Helló, Gutenberg!	100
10.1. Magas szintű programozási nyelvek 1	100
10.2. KERNIGHANRITCHIE	109
10.3. Programozás	113
10.4. Python bevezetés	115
III. Második felvonás	118
11. Helló, Arroway!	120
11.1. A BPP algoritmus Java megvalósítása	120
11.2. Java osztályok a Pi-ben	120
IV. Irodalomjegyzék	121
11.3. Általános	122
11.4. C	122
11.5. C++	122
11.6. Lisp	122

Ábrák jegyzéke

2.1. 0%-os magmozgatás	7
2.2. 100%-os magmozgatás egy mag esetén	8
2.3. 100%-os magmozgatás az összes mag esetén	9
2.4. T100	10
2.5. T1000	11
2.6. BogoMIPS eredménye nálam	20
2.7. PageRank képlet	21
2.8. Pange-Rank eredmény	24
2.9. Monty Hall eredmény	26
2.10. A B_2 konstans közelítése	30
3.1. Turing gép	32
3.2. Lexikális elemző	38
3.3. Leetspeak	41
4.1. A double ** háromszögmátrix a memóriában	49
4.2. A program által kiírt eredmény	50
4.3. A program működése	53
4.4. A neurális háló illusztrálása	56
4.5. Neurális OR kép	58
4.6. Neurális OR eredmény	59
4.7. Neurális AND kép	60
4.8. Neurális AND eredmény	61
4.9. Neurális EXOR kép	62
4.10. Neurális EXOR eredmény	63
4.11. Neurális EXOR kép	64
4.12. Mandelbrot kép	65

5.1. A kiadott kép	73
5.2. A program áltak megcsinált kép	77
5.3. Biomorf kép	81
5.4. C++ mandelbrot nagyító és utazó	87
5.5. Java mandelbrot nagyító és utazó	91
5.6. Többszöri nagyítás után	92

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

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

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket, sorozatokat nézzek meg?

- [21 - Las Vegas ostroma](#), benne a **Monty Hall probléma** bemutatása.
- [Kódjátszma](#), benne a **kódtörő feladat** élménye.

- [Egy csodálatos elme](#), benne a zsenialitás mellett megjelenő küzdelmes lét.
- [Mr. Robot](#) S02E11, benne a dekódolás folyamatának bemutatása. Ezen felül az egész sorozatot érdemes végignézni.
- [Westworld](#), benne az emberi elme és a mesterséges intelligencia közti különbségek és hasonlóságok bemutatása.
- [Mátrix](#), benne szintén a mesterséges intelligencia és az ember kapcsolata, a technológia fellázadása az őt megteremtő ember ellen.
- [Blöff / A Ravasz, az Agy és két füstölgő puskacső](#), ezeknek egyáltalán nincs közük a tematikához, de tökéletes, ha szeretnénk egy kis szünetet tartani a nagy programozás tanulásban. Illetve mindenkinél illik legalább egy idézetet fejből tudni belőlük.

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Szerintem a kód és amit csinál nagyon egyértelmű volt és reeszletesen is dokumentáltam a könyvben, ezért egyéb alámondást nem igényel a videó.

Megoldás videó: <https://youtu.be/GoR2cJ9vzZc>

C végtelen ciklus, mely 0 százalékban dolgoztat egy magot.

A forráskód megtalálható a következő linken is: [./Forraskodok/Turing/2.1/vegtelen1.c](#)

```
#include <unistd.h>

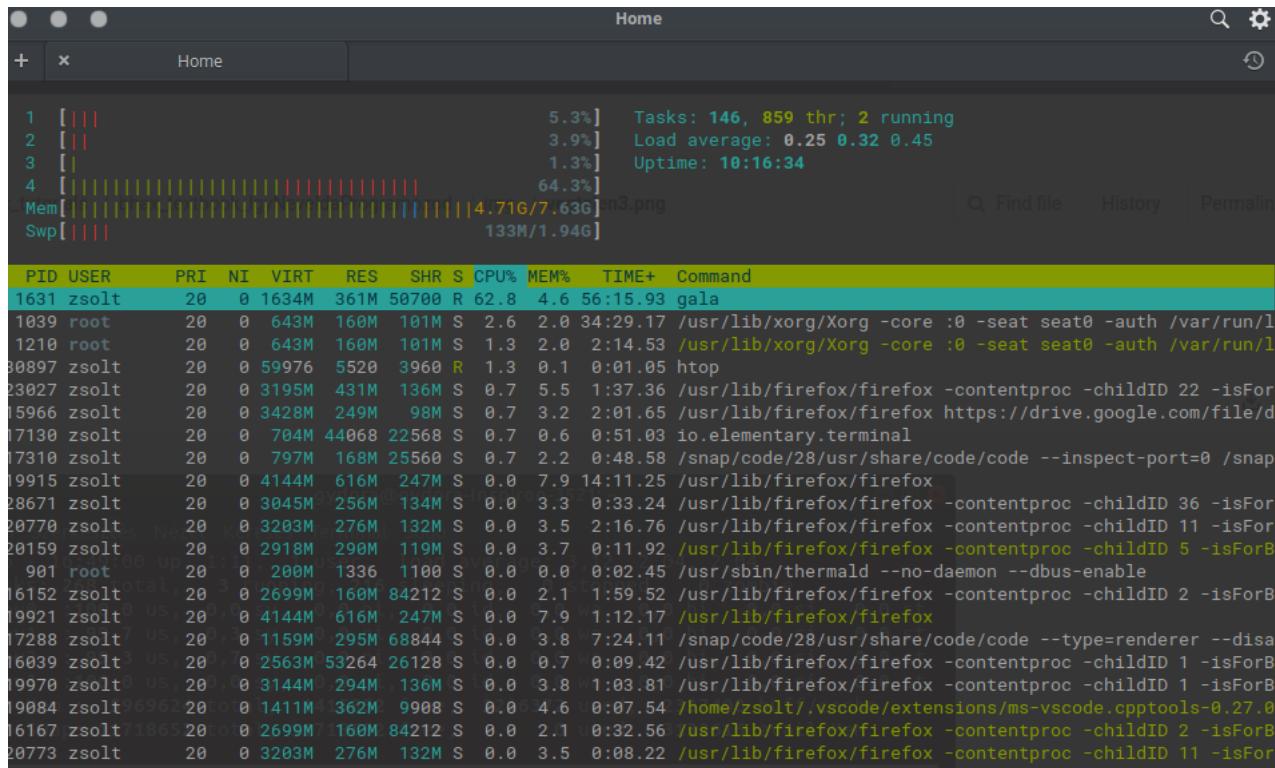
int main ()
{
    for (;;)
        sleep (1);
    return 0;
}
```

Fordítás: **gcc vegtelen1.c -o vegtelen1**

Futtatás: **./vegtelen1**

Tudjuk, hogy a for ciklusban ;-vel válsztjuk el a felsorolást. Mint latjuk a programrészletben ezek hiányoznak, ezért alakul ki a végtelen ciklus, egy ciklus mely sosem áll le, amit mi kell leállitsuk, megöljünk (kill). A for ciklusunk egyetlen egy parancsot tartalmaz, ez a **sleep** parancs ami miatt a program 0 százalékban mozgatja a magot, a paracsrról a **man 3 sleep** kézikönyvből többet is megtudhatunk.

Minden esetben, a program futtatása után, lekérünk egy **top** parancsot, hogy megnézzük a tényleges használatot (a top-on belül az 1-es gomb lenyomásával láthatóvá válik az összes mag).



2.1. ábra. 0%-os magmozgatás

Felül a CPU-knál látjuk, hogy mindegyik nagyon közel van a 0-hoz. A listázott részben pedig azért nem jelenik meg a programunk, mert a magmozgatás szempontjából csökkenő sorrendbe van rendezve a lista (de a lista végére görgetve láthtuk azt is).

C végtelen ciklus, mely 100 százalékban dolgoztat egy magot.

A forráskód megtalálható a következő linken is: [./Forraskodok/Turing/2.1/vegtelen2.c](#)

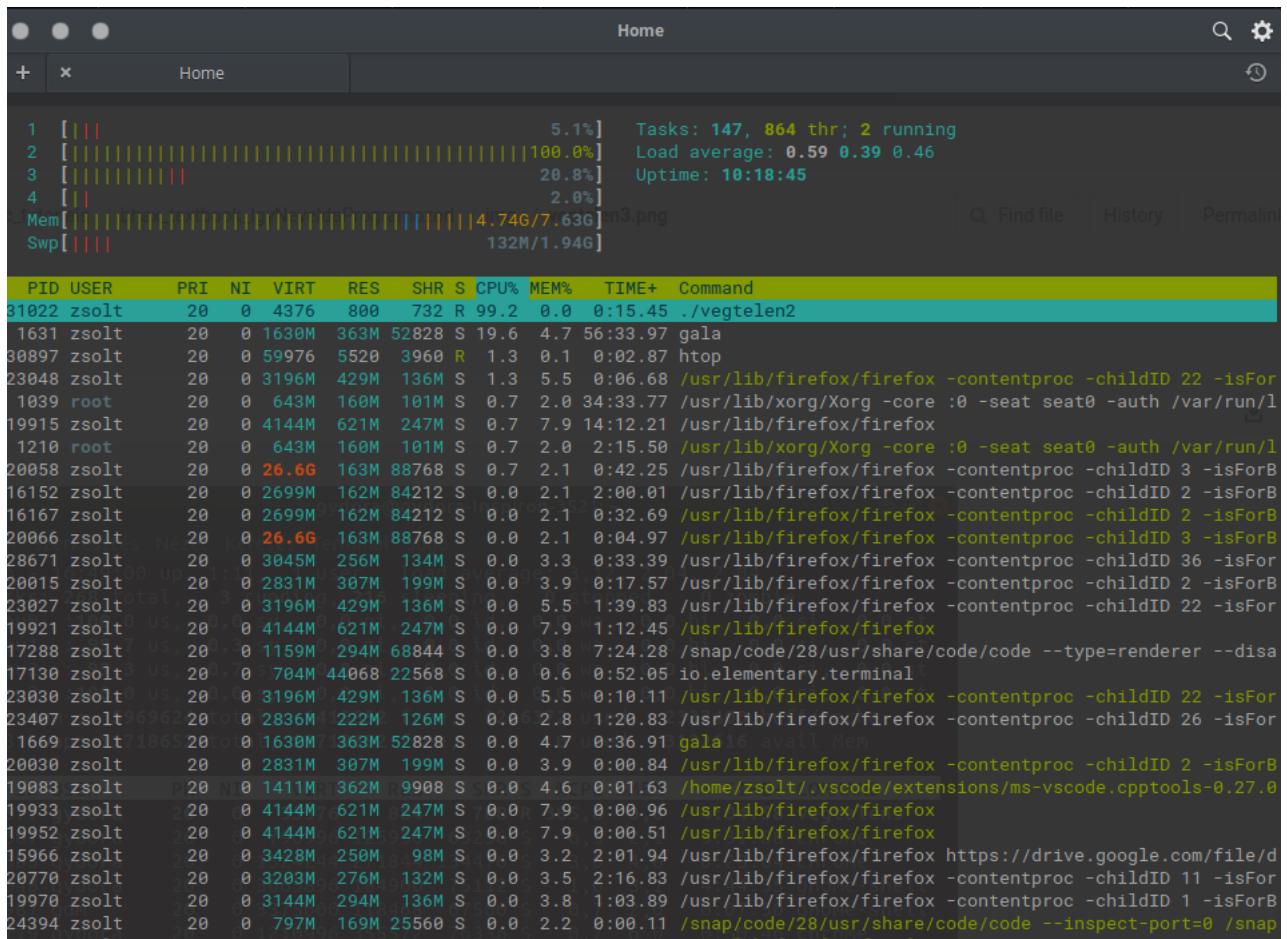
```
#include <stdio.h>

int main() {
    for (;;)
    {
        ;
    }
}
```

Fordítás: **gcc vegtelen2.c -o vegtelen2**

Futtatás: **./vegtelen2**

A programban mint látjuk a for ciklusban nincs sem feltétel, sem elvégezendő utasítás (;), ezért alakul ki a végtelen ciklus. A **top** parancs futtatása után látjuk hogy egy magot mozgat csak, de azt 100%-on. A lista részben viszont az előző példával szemben, most a lista élén jelenik meg a programunk, jelezve a 100%-os magmozgatást.



2.2. ábra. 100%-os magmózgatás egy mag esetén

C végtelen ciklus, mely 100 százalékban dolgoztat minden magot.

A forráskód megtalálható a következő linken: [./Forraskodok/Turing/2.1/vegtelen3.c](#)

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

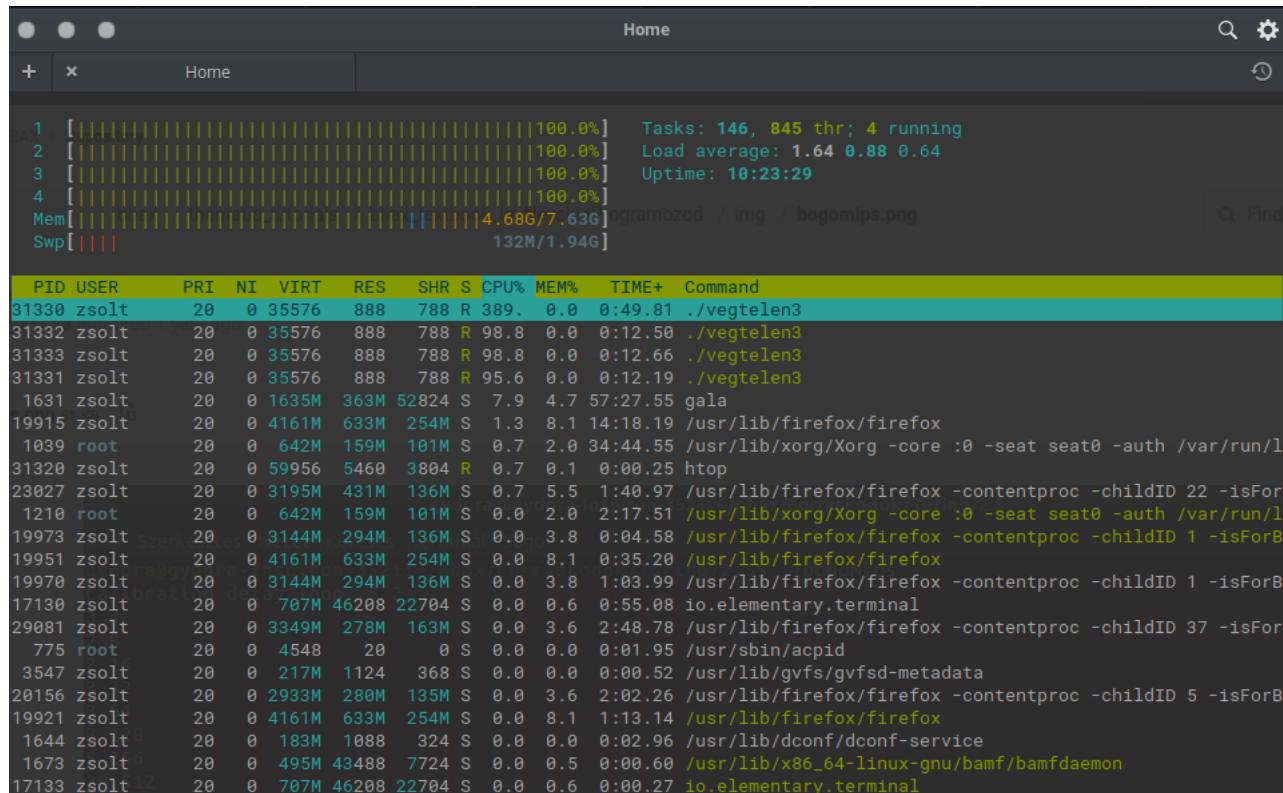
int main() {
    #pragma omp parallel
    {
        for (;;) ;
    }
    return 0;
}
```

Fordítás: **gcc vegtelen3.c -o vegtelen3 -fopenmp**

Futtatás: **./vegtelen3**

A for ciklus ugyanúgy működik mint az előző példákban. A **#pragma omp parallel** parancs egy szálcsoportot hoz létre és amikor fut a végtelen ciklus változatja ezeket a szálakat. Majd újabb **top**-al megnézzük.

Mint látjuk, minden mag sorában megjelenik a 100% érték, mely azt jelenti, hogy sikerült elérnünk azt hogy a program minden magot egyszerre mozgasson. A listás résznél az élen újra a végtelen program van, a közel 400-as értékkel sugallva, hogy mind a 4 mag mozog.



2.3. ábra. 100%-os magmozgatás az összes mag esetén

Itt már látszik hogy minden mag sorában megjelenik a 100% érték, mely azt jelenti, hogy sikerült elérnünk azt hogy a program minden magot egyszerre mozgasson. A listás résznél az élen újra a végtelen program van, a közel 400-as értékkel sugallva, hogy mind a 4 mag mozog.

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 döntenı, hogy van-e benne végtelen ciklus:

```
Program T100
```

```
{
```

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

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

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

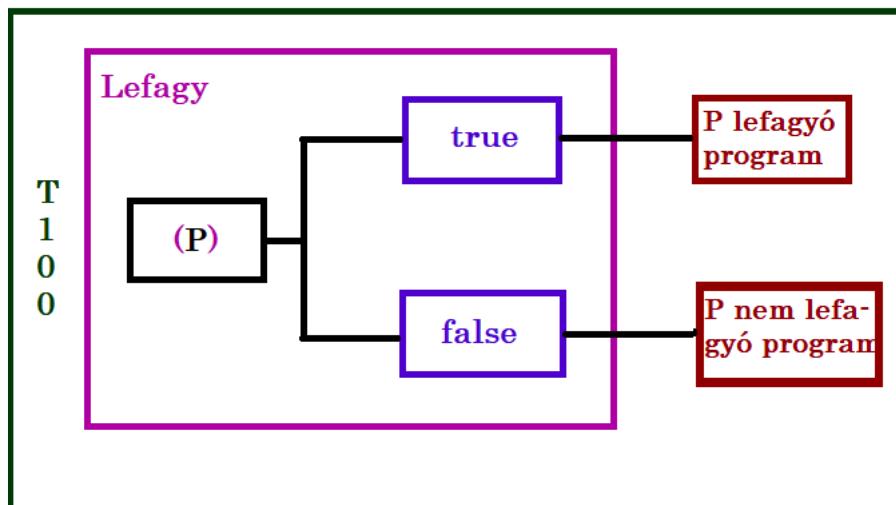
```
T100(t.c.pseudo)
true
```

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

Egy kép, mely illusztrálja a feltételezett program működését:



2.4. ábra. T100

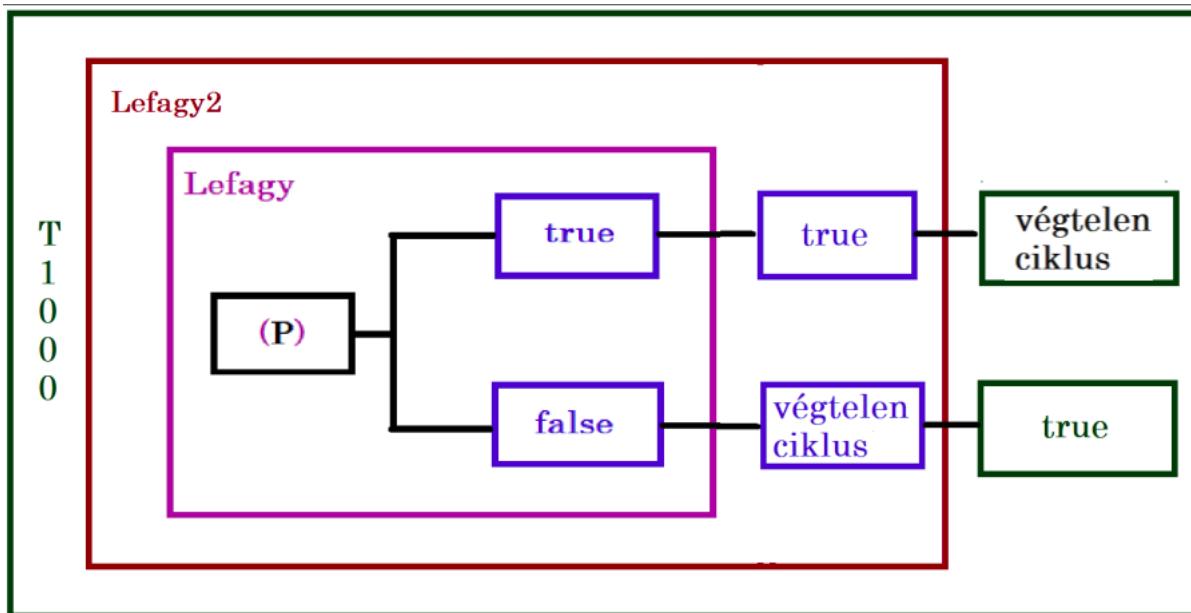
A T100-as programot felhasználva készítsük most el az alábbi T1000-set, amelyben a Lefagy-ra épőlő Lefagy2 már nem tartalmaz feltételezett, csak csak konkrét kódot:

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

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

    main(Input Q)
    {
        Lefagy2(Q)
    }
}
```

Egy újabb ilusztráció a program működéséről:



2.5. ábra. T1000

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

Tehát, feltételezések alapján bebizonyítottuk, hogy akár mekkora hackerek legyünk, nem lehet ilyen programot írni. A T100-as program lefut és el is dönti hogy a betáplált programról, hogy lefagyó program e, helyesen adja vissza a true és a false értékeket. Viszont a T1000-es esetén már ellentmondásba ütközünk, mivel amikor saját magát kapja paraméterül akkor elég más eredményeket ad mint amire várunk. Amikor a T1000-es egy lefagyó program lesz, akkor a vegén nem fog lefagyni, hanem a true értéket fogja visszaadni. Ellenkező esetben pedig, amikor a T1000 nem egy lefagyó program, akkor le fog fagyni, vagyis a false értéket kapjuk. Ez az ellentmondás arra ad bizonyítékot, hogy egy ilyen program még ha meg is lehetne írni, akkor se működne.

2.3. Változók értékének felcserélése

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

Szerintem a kód és amit csinál nagyon egyértelmű volt és reeszletesen is dokumentáltam a könyvben, ezért egyéb alámondást nem igényel a videó.

Megoldás videó: <https://youtu.be/f23nJjxGl7k>

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

Ezt a feladatot háromféle példával fogom illusztrálni (egykben sem használva segédváltozót). A prgram mindegyik esetben a 3-as és a 8-as értéket fogja felcserélni (de természetesen más számokkal is működik), az elején kiiratja az eredeti, a végén pedig a megváltoztatott értékeket. A számolás menete minden esetben követhető lesz a programrészletben.

Az első program a legegyszerűbb műveletekkel, összeadással és kivonással végzi el a változócsereit.

A forráskód megtalálható a következő linken is: [./Forraskodok/Turing/2.1/valtcserkul.c](#)

```
#include <stdio.h>

int main()
{
    int a=3;
    int b=8;
    printf("Eredeti ertekek:");
    printf("\na=%d", a);
    printf("\nb=%d", b);

    a = a+b;      //a=3+8=11
    b = a-b;      //b=11-8=3
```

```
a = a-b;      //a=11-3=8

printf("\nMegvaltoztatott ertekek:");
printf("\na=%d", a);
printf("\nb=%d", b);
printf("\n");
return 0;
}
```

Fordítás: **gcc valtcserkul.c -o valtcserkul**

Futtatás: **./valtcserkul**

A program elején deklarálunk két változót, ezeket ki is íratjuk a standard kimenetre, hogy a felhasználó tudja ellenőrizni a program működését. Majd elvégezzük a változócserét. A három program leírása során csak a változók cseréjének a módszere fog különbözni, a többi úgyanaz. A műveletek elvégeztével kiíratjuk a felhasználó számára a felcserélt értékeket is, bizonyítva a program helyességét.

A második program a szorzás és osztás segítségével cseréli meg a változók értékeit.

A forráskód megtalálható a következő linken is: [..//Forraskodok/Turing/2.3/valtcserszor.c](#)

```
#include <stdio.h>

int main()
{
    int a=3;
    int b=8;
    printf("Eredeti ertekek:");
    printf("\na=%d", a);
    printf("\nb=%d", b);

    a=a*b;    //a=3*8=24
    b=a/b;    //b=24/8=3
    a=a/b;    //a=24/3=8

    printf("\nMegvaltoztatott ertekek:");
    printf("\na=%d", a);
    printf("\nb=%d", b);
    printf("\n");
    return 0;
}
```

Fordítás: **gcc valtcserszor.c -o valtcserszor**

Futtatás: **./valtcserszor**

Végül pedig a harmadik programban, amelyik az exor(xor) művelettel hajtja végre a változócserét.

A forráskód megtalálható a következő linken is: [..//Forraskodok/Turing/2.3/valtcserexor.c](#)

```
#include <stdio.h>

int main()
```

```
{  
    int a=3;      //a=0011  
    int b=8;      //b=1000  
    printf("Eredeti értékek:");  
    printf("\na=%d", a);  
    printf("\nb=%d", b);  
  
    a=a^b;      //a=0011^1000=1011  
    b=a^b;      //b=1011^1000=0011  
    a=a^b;      //a=1011^0011=1000  
  
    printf("\nMegvaltoztatott értékek:");  
    printf("\na=%d", a);  
    printf("\nb=%d", b);  
    printf("\n");  
    return 0;  
}
```

Fordítás: **gcc valtcserek.c -o valtcserek**

Futtatás: **./valtcserek**

A xor művelet, másnépp kizáró vagy, egy bitművelet. Két egyforma hosszúságú bitsorozaton végzi el a kizáró vagy műveletet, ami annyiból áll, hogy ahol megegyezik a bitek értéke ott 0-át ad eredményül, ahol pedig eltér ott 1-et.

2.4. Labdapattogás

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

Megoldás videó: <https://youtu.be/qHB1u2OdSxY>

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

Labdapattogtatás if-el

A forráskód megtalálható a következő linken is: <..//Forraskodok/Turing/2.4/labda1.c>

```
#include <stdio.h>  
#include <curses.h>  
#include <unistd.h>  
  
int  
main ( void )  
{  
    WINDOW *ablak;  
    ablak = initscr ();  
  
    int x = 0;
```

```
int y = 0;  
  
int xnov = 1;  
int ynov = 1;  
  
int mx;  
int my;
```

A program elején egy curses nevű header fájt deklarálunk, ami lehetővé teszi az olyan függvények egyszerű meghívását mint például a **WINDOW**, az **initscr** stb. Ezek a függvények mind az ablak (melyben majd a labda pattogni fog) beállításokról, műveletekről felelnek (méret, az ablak inicializálása, output kirajolása az ablakba, stb).

A header fájlok felsorolása után jöhet a program többi része. A fenti programrészletben a **WINDOW** parancccsal létrehozzuk azt az ablakot amelyikben a labdát fogjuk pattogtatni, majd inicializáljuk azt (**initscr**). Az x, y tengely mentén rajzolódik majd ki a kis labda, ehhez változókat deklarálunk és kezdeti értékeket adunk meg a labda kezdeti pozíciójának.

```
for ( ; ; ) {  
  
    getmaxyx ( ablak, my , mx );  
  
    mvprintw ( y, x, "o" );  
  
    refresh ();  
    usleep ( 100000 );
```

Mint látjuk, a nemrég megírt programok egyikét felhasználva, egy végtelen ciklust hozunk létre, ez fogja tartalmazni a labda pattogtatásával kapcsolatos információkat, mozgatási feltételeit. A **getmaxyx** függvényel meghatározzuk az ablak méreteit, hogy hol és mennyi távolságot pattogjon a labda. A **mvprintw** függvény fogja kirajzolni a mi labdánkat a képernyőre, minden rajzolás után frissítve állapotát (**refresh**). Az **usleep** parancccsal pedig a labda pattogási, mozgási sebességét tudjuk megadni.

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

```
    return 0;  
}
```

A program utolsó felében azokat az eseteket tárgyaljuk amikor a labda eléri az ablak valamelyik szélét (a commnetekkel jelölve hogy éppen melyik szélre ural), amikor a labda eléri a keretet. A négy darab if a négy szél tárgyalása.

Fordítás: **gcc labda1.c -o labda1 -lncurses**

Futtatás: **./labda1**

Labdapattogtatás is nékül

A forráskód megtalálható a következő linken is: [..//Forraskodok/Turing/2.4/labda2.c](#)

```
#include <stdio.h>  
#include <stdlib.h>  
#include <curses.h>  
#include <unistd.h>  
  
int  
main (void)  
{  
    int xj = 0, xk = 0, yj = 0, yk = 0;  
    int mx = 80 * 2, my = 24 * 2;  
  
    WINDOW *ablak;  
    ablak = initscr ();  
  
    for (;;) {  
        xj = (xj - 1) % mx;  
        xk = (xk + 1) % mx;  
  
        yj = (yj - 1) % my;  
        yk = (yk + 1) % my;  
  
        clear ();  
  
        mvprintw (0, 0,  
                  " ←  
                  -----  
                  ");  
        mvprintw (24, 0,  
                  " ←  
                  -----  
                  ");  
        mvprintw (abs ((yj + (my - yk)) / 2),  
                  abs ((xj + (mx - xk)) / 2), "○");  
  
        refresh ();  
        usleep (150000);
```

```
    }
    return 0;
}
```

Fordítás: **gcc labda2.c -o labda2 -lncurses**

Futtatás: **./labda2**

A labdapattogtatás if nélküli megoldása már nem érvényes minden nagyságú ablakra, a szaggatott vonalak segítségével fogjuk behatárolni majd azt. A program felépítése hasonló az előzőjéhez, a header fájlok után a főprogram, abban változó deklarálások (melyek az ablak kinézetének és a labda helyzetének a meghatározásához kellenek), az ablak létrehozása amelyikben majd a labda mozog. A for ciklus segítségével újra készítünk egy végtelen ciklust, hogy a labdapattogtatás újra addig tartson, amíg mi le nem állítjuk. Mint ahogy a program neve is mutatja, if nélküli labdapattogtatás, nincs benne if, tehát a pattogtatás bármiféle logikai utasítás vagy kifejezés használata nélkül megy végbe. A labda adott (jelenlegi) helyzetét elosztjuk az ablak méretével (szélességet szélességgel, magasságot magassággal, vagyis az x-eseket az mx-el, az y-osokat az my-al) és hogyha ennek az osztásnak a maradéka 0, akkor a labda elérte az adott szélt és itt az ideje hogy visszaforduljon. A továbbiakban ūhasznált három függvény az if-es labdapattogtatásban már be vannak mutatva.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó:

Megoldás forrása:

Elsősorban vizsgálunk meg egy olyan programot, amelyik megnézi, hogy hány bites a szó a gépünkön, azaz mekkora az int mérete, a **sizeof** segítségével.

A forráskód megtalálható a következő linken is: [./Forraskodok/Turing/2.5/szohossz.c](#)

```
#include <stdio.h>

int main()
{
    printf("%ld\n", sizeof(int)*8);
    return 0;
}
```

Fordítás: **gcc szohossz.c -o szohossz**

Futtatás: **./szohossz**

Végeredményül a 32-t kapjuk. Az eredményt a **sizeof()** segítségével kapjuk meg, melyben megnézzük hogy mennyi az int hossza, a kapott értéket megszorozzuk 8-al mivel így az eredményt bitben kapjuk meg, nem pedig bájtból.

A második program ugyancsak a szóhosszt nézi meg, annyi különbséggel, hogy ez bitműveletes módszerrel.

A forráskód megtalálható a következő linken is: [./Forraskodok/Turing/2.5/szohossz2.c](#)

```
#include <stdio.h>

int main(void)
{
    int h = 0;
    int n = 0x01;
    do
        ++h;
    while (n<<=1);
    printf("A szohossz ezen a gepen: %d bites\n", h);
    return 0;
}
```

Fordítás: **gcc szohossz2.c -o szohossz2**

Futtatás: **./szohossz2**

Az n változót egy hexadecimális számmal deklaráljuk. Majd egy hátultesztelős do-while ciklussal növeljük h értékét minden lépésnél, a while-ban egy helyiérték eltolás van balra, ez a feltétel akkor vális hamissá, amikor a h eléri a 32 bitet. A végeredmény a következő lesz: A szohossz ezen a gepen: 32 bites.

Végül pedig nézzük hogy mi is az a BogoMIPS.

A BogoMIPS a "Bogus MIPS"-ből származik (bogus=hamis). A MIPS rövidítésnek két megfelelője is elterjedt a köztudtaban. Az első a Meaningless Indication of Processor Speed (a processzor sebességének értelmetlen jellemzése), ami annyit tesz, hogy a processzor a másodperc egyik milliomodnyi részében sem csinál semmit. Az második értelmezése a Million Instructions Per Second (millió utasítás/művelet másodpercenként), mely alatt a számítógép számítási sebességét értjük. A BogoMIPS Linus Torvalds találmánya. A rendszerindítás (bootolás) során a kerner megméri, hogy egy bizonyos ciklusnak mennyi a futási ideje. Tehát mint a nevének leírásában is, a BogoMIPS eredményből következtetni tudunk a processzor sebességére, de ez annyira ál, már-már megalapozatlan, tudománytalan, hogy csak a BogoMIPS kifejezés illet rá. A BogoMIPS eredmények többnyire 386 és 586 között vannak, ha az eredmény ez az intervallum alatt van (tehát kisebb mint 386), akkor a CPU sebesség vagy a Turbó gomb nincs megfelelően, optimusan beállítva.

A forráskód megtalálható a következő linken is: [..Forraskodok/Turing/2.5/bogomips.c](#)

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

void
delay (unsigned long long int loops)
{
    unsigned long long int i;
    for (i = 0; i < loops; i++)
}

int
main (void)
{
    unsigned long long int loops_per_sec = 1;
    unsigned long long int ticks;
```

```
printf ("Calibrating delay loop..");
fflush (stdout);

while ((loops_per_sec <= 1))
{
    ticks = clock ();
    delay (loops_per_sec);
    ticks = clock () - ticks;

    printf ("%llu %llu\n", ticks, loops_per_sec);

    if (ticks >= CLOCKS_PER_SEC)
    {
        loops_per_sec = (loops_per_sec / ticks) * CLOCKS_PER_SEC;

        printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec / 500000,
               (loops_per_sec / 5000) % 100);

        return 0;
    }
}

printf ("failed\n");
return -1;
}
```

Fordítás: **gcc bogomips.c -o bogomips**

Futtatás: **./bogomips**

A `delay` segít lemérni az elvégzéshez szükséges időt. A `while` feltételében `loop_per_sec` értéke minden ciklus kezdeténél egyel balra lesz tolva, tehát a ciklus csak túlcordultság vagy az `if` igazsága miatt fog leállni. A ciklusban a `ticks` először az aktuális, majd az eltelt időt fogja tárolni. Majd kiiratódik a `ticks` és a `loop_per_sec` aktuális értéke. Ha bekövetkezik az, hogy az `if` feltétele igaz lesz, tehát ha a `ticks` nagyobb vagy egyenlővé válik mint a `CLOCKS_PER_SEC` akkor a program elér a végéhez és kiírja a BogoMIPS eredményt a képernyőre. Az utolsó két sor csak akkor lesz végrehajtva, ha az `if` feltétele sosem lesz igaz.

```
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Turing/2
$S gcc bogomips.c -o bogomips
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Turing/2
$ ./bogomips
Calibrating delay loop..1 2
0 4
1 8
1 16
1 32
1 64
1 128
1 256
2 512
4 1024
7 2048
14 4096
25 8192
55 16384
100 32768
200 65536
400 131072
807 262144
1600 524288
3226 1048576
5088 2097152
8270 4194304
16810 8388608
32709 16777216
65034 33554432
126077 67108864
248034 134217728
494909 268435456
992918 536870912
1976347 1073741824
ok - 1086.00 BogoMIPS
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Turing/2
$S
```

2.6. ábra. BogoMIPS eredménye nálam

Ahogy a csatolt képen is látszik végeredményül 1086-at kaptam, ami annyit jelent, hogy az én gépem 1 másodperc alatt 1086 millió utasítást tud végrehajtani.

2.6. Helló, Google!

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

Megoldás videó:

Megoldás forrása:

A PageRank egy nagyon elterjedt algoritmus az informitikában. Az algoritmust Larry Page és Sergey Brin, a Google alapítói írták 1998-ban a Stanford Egyetemen. Az algoritmus és ezáltal a Google internetes kereső is arra a feltevésre épül, hogy a weboldalak készítői azokra a weblapokra hivatkoznak a saját oldalukról, amit jónak tartanak. Ez hasonlít egy szavazáshoz: a hivatkozások az adott oldalra (tehát minden hiperlink) egyfajta szavazatnak számítanak. A hivatkozások mellett fontos szerepet játszik még az, hogy annak az oldalnak ami hivatkozott, ami leadta a szavazatát, annak hány szavazata van, tehát milyen fontos. Itt kialakul egyfajta rekúrzsí folyamat: az az oldal számít fontonak, amelyikre olyan oldalak mutatnak amik fontosak.

Kezdetben minden oldalnak van valamennyi szavazata, illetve azok a szavazatok amiket más oldalaktól kapnak. Ezeket egyenlően szétozt azok az oldalak között amikre majd ő hivatkozik. Az oldal PageRank

értéke megegyezik a kapott szavazatok számával. Hogy minden jól menjen be kell vezetni egy csillapító erőt, tényezőt, amelyik arról felel, hogy a saját szavazatuk (1-d) része maradjon meg az oldalnak, d részét pedig továbboszthat. Ezért a következő a képletet lehet felírni:

$$\text{PageRank}(i) = (1 - d) + d \sum_{j \in M(i)} \frac{\text{PageRank}(j)}{L(j)}$$

2.7. ábra. PageRank képlet

ahol az $M(i)$ az i. oldalra hivatkozó oldalaknak a száma, az $L(j)$ pedig a j. oldalról kimenő hivatkozásoknak a száma.

Mindezen ismertetők után lássuk a programot, amelyik kiszámolja ezt az értéket.

A forráskód megtalálható a következő linken is: [./Forraskodok/Turing/2.6/pagerank.c](#)

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

void
kiir (double tomb[], int db) {

    int i;

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

A program elején deklarálunk néhány olyan függvényt. Ilyen a `kiir` függvény is, melynek külön való megírása egyszerűsíti a dolgunkat és sokkal átláthatóbbá teszi a programunkat. A `kiir` függvénnnyel egy egydimenziós tömböt, vagyis egy vektort tudunk majd kiiratni, minden elemét új sorba íratva.

```
double
tavolsag (double PR[], double PRv[], int n) {

    int i;
    double osszeg=0;

    for (i = 0; i < n; ++i)
        osszeg += (PRv[i] - PR[i]) * (PRv[i] - PR[i]);

    return sqrt(osszeg);
}
```

A `tavolsag` nevű függény három paramétert kap, két egydimenziós tömböt és egy `n`-t, ami a tömbök méretét jelöli. A távolság számítása a következőképpen történik: minden tömbből kivonjuk egymásból az egyforma indexű elemeket, ezeket a négyzetre emeljük, majd a kapott értéket hozzáadjuk egy `osszeg` nevű változóhoz (az `osszeg`-ben `n` darab érték összege lesz). A függvény visszatérítési értéke pedig ennek az összegnek a gyöke lesz.

```

void
pagerank(double T[4][4]) {
    double PR[4] = { 0.0, 0.0, 0.0, 0.0 };      //ebbe megy az eredmény
    double PRv[4] = { 1.0/4.0, 1.0/4.0, 1.0/4.0, 1.0/4.0};    //ezzel szorzok

    int i, j;

    for(;;) {
        // ide jön a mátrix művelet
        for (i=0; i<4; i++) {
            PR[i]=0.0;
            for (j=0; j<4; j++) {
                PR[i] += T[i][j] * PRv[j];
            }
        }

        if (tavolsag(PR,PRv,4) < 0.0000000001)
            break;

        // ide meg az átpakolás PR-ből PRv-be
        for (i=0;i<4; i++) {
            PRv[i]=PR[i];
        }
    }

    kiir (PR, 4);
}

```

A pagerank függvény az ami elvégzi nekünk a mátrixszorzást és visszatéríti a szorzás eredményét. A paraméterül kapott T kétdimenziós tömb az amit mi a main-ben megadunk, a függvény ezt szorozza össze a PRv tömbbel, mely az oldalak első iterációbeli értékeit tartalmazza, az eredmény pedig a PR-be fog kerülni. Matematikai szemszögből a helyes megfogalmazás az lenne hogy a PRv-t szorozzuk a T-vel mivel a mátrixszorzás szabalyit figyelembe véve csak így végezhető el a mátrixszorzás, vagyis egy 1x4-es mátrix szorzása egy 4x4-es mátrixal egy 1x4-es mátrixot ad eredményül (másképp mondva egy egydimenziós tömböt). Kezdünk egy végtelen ciklust, jön a mátrixszorzás majd megvizsgáljuk az PR (eredmény mátrix) és a PRv közötti távolságot, ha ez az érték kisebb mint 0.0000000001, akkor a break fügvénnyel kiléünk a végtelen ciklusból, ha pedig nem akkor a PRv-ben lévő értékeket lecseréljük az eredmátrixban lévő értékekkel és a következő körben a mi általunk megadott mátrixot ezzel fogjuk szorozni.

```

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 L1[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}
    };
}

```

```
{1.0, 1.0/2.0, 1.0/3.0, 0.0},  
{0.0, 1.0/2.0, 0.0, 0.0},  
{0.0, 0.0, 1.0/3.0, 0.0}  
};  
  
double L2[4][4] = {  
    {0.0, 0.0, 1.0/3.0, 0.0},  
    {1.0, 1.0/2.0, 1.0/3.0, 0.0},  
    {0.0, 1.0/2.0, 0.0, 0.0},  
    {0.0, 0.0, 1.0/3.0, 1.0}  
};  
  
printf("\nAz eredeti mátrix értékeivel történő futás:\n");  
pagerank(L);  
  
printf("\nAmikor az egyik oldal semmiré sem mutat:\n");  
pagerank(L1);  
  
printf("\nAmikor az egyik oldal csak magára mutat:\n");  
pagerank(L2);  
  
printf("\n");  
  
return 0;  
}
```

A főprogramban már csak annyi dolgunk van, hogy megadjuk a kezdeti mátrixokat az L-t, L1-et és az L2-t. Az L kétdimenziós mátrix tartalmazza a linkelési adatokat, tehét hogy melyik oldal melyik oldalra hivatkozik. Az L1 esetén azt az esetet tudjuk megtekinteni amikor az egyik oldal semmiré sem mutat, az L2 esetén pedig azt amikor az egyik oldal csak magára mutat. Ezeket a mátrixot beküldjük a pagerank függvénybe, ott minden művelet elvégződik, az eredmény pedig kiiratódik a standard outputra.

Fordítás: **gcc pagerank.c -lm**

Futtatás: **./a.out**

Eredményül a következőt kapjuk:

```
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Turing/2
.6$ ./pagerank

Az eredeti mátrix értékeivel történő futás:
0.000909
0.545455
0.272727
0.090909

Amikor az egyik oldal semmirre sem mutat:
0.000000
0.000000
0.000000
0.000000

Amikor az egyik oldal csak magára mutat:
0.000000
0.000000
0.000000
1.000000

(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Turing/2
.6$ █
```

2.8. ábra. Pange-Rank eredmény

2.7. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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

A Monty Hall probléma alapja három ajtó, mely mindegyike rejt valamit: két ajtó mögött egy-egy roncs autó van, egy ajtó mögött pedig egy vadiúj autó. Ez annyit jelent, hogy két ajtó értéktelen és egy ajtó pedig értékes ajándékot rejti. Azért, hogy könnyebben tudunk beszélni az egészről, mondjuk azt, hogy két ajtó mögött nincs és egy ajtó alatt pedig van ajándék. A folyamat különböző állomásait a program ismertetése közben prezentálom.

A forráskód megtalálható a következő linken is: [./Forraskodok/Turing/2.7/montyhall.r](#)

```
# An illustration written in R for the Monty Hall Problem
# Copyright (C) 2019 Dr. Norbert Bátfa, 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.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
```

```
# along with this program. If not, see <http://www.gnu.org/licenses/>
#
# https://bhaxor.blog.hu/2019/01/03/ ←
# erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcsan
#
kiserletek_szama=10000000
kiserlet = sample(1:3, kiserletek_szama, replace=T)
jatekos = sample(1:3, kiserletek_szama, replace=T)
```

Az első dolgunk hogy megadjuk a kísérletek számát, tehát azt, hogy hanyszor végezzük el a kísérletet, hogy hanyszor játszik el a játékot (ez a mi estetünkben, a mi programunkban 10000000 kört jelent). minden játékban van harom ajtónk: 1-es, 2-es és 3-as ajtó. A következő sorban a kiserlet vektorba berakjuk, hogy az egyes játékokba, melyik ajtó mögé lett betéve az ajándék, tehát hogy melyik ajtó a nyerő. (a **replace** megengedi, hogy a játékok során egy ajtó mögé többször is kerülhessen ajándék). Következik a játékos választásai. A jatekos vektor feltöltése, ugyanúgy működik mint a kiserlet esetében, itt a játékos tippjét rögzítjük, hogy szerinte melyik ajtó rejti az ajándékot.

```
musorvezeto=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama) {

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

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

  }else{

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

  }

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

}
```

A műsorvezetőnek egy kicsit komplexebb dolga van. Neki most ki kell nyitnia a játékosnak egy ajándékot nem takaró ajtót, de figyelembe kell vennie azt, hogy hova lett elhelyezve az ajándék és hogy mit választott a játékos. A for cíussal végigmegyünk az összes játékon. Az if segítségével döntjük el, hogy a műsorvezető melyik ajtót nyissa majd ki. Az if első ágán azt nézzük meg, hogy a játékos ugyanazt az ajtót választotta-e mint amelyikbe az ajándékot elrejtették. Ha igen, akkor a mibol a másik két ajtósáma egyike lesz (az 1, 2 és 3-as ajtók közül kivesszük azt amelyikben az ajándék van). Ha viszont a játékos nem azt az ajtót választja amelyik mögött a nyeremény szerepel, akkor jön az else ág, és azt mondjuk, hogy a mibol legyen az az ajtó száma, amelyik mögött nem az ajándék van és amelyiket nem mondta a játékos (tehát az 1, 2 és 3-as ajtók közül kivesszük azt amelyik a nyereményt rejti és azt amelyiket a jáékos választotta). Miután sikerült eldönteneni hogy mi legyen a mibol értéke, átadjuk azt a műsorvezetőnek, aki kinyitja azt az ajtót. Igy az ajtó amit mjd kinyit egy üres ajtó lesz.

```
nemvaltoztatesnyer= which(kiserlet==jatekos)
```

```
valtoztat=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama) {

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

valtoztatesnyer = which(kiserlet==valtoztat)
```

A következőkben a játékosnak el kell döntenie, hogy kiáll az eredeti választása mellett vagy változtat és másik ajtót választ. A nemvaltoztatesnyer akkor kap értéket ha a játékos azt az ajtót választotta amelyik az ajándékot rejti, és így nyer majd. Ha viszont nem azt választotta, akkor változtathat a döntésén és ehhez szükségünk lesz egy for ciklusra, amelyik újra végig megy majd az összes játékon. A holvart annak az ajtónak a számát jegyzi meg, amelyik nem is a műsorvezető által kinyitott és nem is a játékos által választott ajtó volt, a valtoztat pedig átveszi tőle ezt az értéket. Aztan megnézi hogy ha az ajándékot rejtő ajtó megegyezik a változtatott értékkel akkor a valtoztatesnyer lesz kiiratva nem pedig a nemvaltoztatesnyer.

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

A program végén már csak egy összegző kiírás van, melyben kiiratjuk, hogy mennyi a játékok száma (hány kört játszottunk), hányszor volt az, hogy a játékos nem változtatott és nyert illetve hogy változtatással nyert, ezek hányadosa illetve összege.

A program futtatása R-ben történik, ehhez szükséges telepíteni azt. A telepítés után belépünk R-be, először a **kiserletek_szama=10000000** sort adjuk be, majd pedig a program többi részét. Az eredményre egy keveset várni kell hiszen ahogy a program elején latjuk a kísérletek száma elég nagy (10000000). Ha ezt az értéket lentebb visszük, akkor a végeredményt is kevesebb idő alatt kapjuk majd meg. Többnyire minden futtatás után más és más eredményt kapunk. Az én esetben a következő eredmény jön ki:

```
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/
.7$ Rscript montyhall.r
^T[1] "Kiserletek szama: 10000000"
[1] 3336466
[1] 6663534
[1] 0.5007052
[1] 10000000
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/
.7$ █
```

2.9. ábra. Monty Hall eredmény

2.8. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

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

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

Bepillantás a Brun téTELre: A téTEL alapja az ikerprímek, tehát azok a prímszámok, melyeknek a különbségük 2. Mivel végtelen sok termézesztes, egész... számunk van, ezért úgyanúgy ahogy az ōk darabszámét, úgy az ikerprímekét se tudjuk. De van amit tudunk, mégpedig azt, hogy ezek reciprokainak az összege egy számhoz konvergál, ami nemmás mint a Brun-konstans. Ennek értéke se egy pontos szám amit tudunk, de az érték megközelítéséhez a következõ R programot használjuk, kifejtése lennebb található.

A forráskód megtalálható a következõ linken is: [./Forraskodok/Turing/2.8/brun.r](#)

A termézesztes számok építőelemei a prímszámok, abban az értelemben, hogy minden termézesztes szám előállítható prímszámok szorzataként. Például $12=2*2*3$, vagy például $33=3*11$.

Prímszám az a termézesztes szám, amely csak önmagával és eggyel osztható. Eukleidész görög matematikus már Krisztus előtt tudta, hogy végtelen sok prímszám van, de ma sem tudja senki, hogy végtelen sok ikerprím van-e. Két prím ikerprím, ha különbségük 2.

Két egymást követõ páratlan prím között a legkisebb távolság a 2, a legnagyobb távolság viszont bármilyen nagy lehet! Ez utóbbit könnyű bebizonyítani. Legyen n egy tetszőlegesen nagy szám. Akkor szorozzuk össze $n+1$ -ig a számokat, azaz számoljuk ki az $1*2*3*\dots*(n-1)*n*(n+1)$ szorzatot, aminek a neve $(n+1)$ faktoriális, jele $(n+1)!$.

Majd vizsgáljuk meg az a sorozatot:

$(n+1)!+2, (n+1)!+3, \dots, (n+1)!+n, (n+1)!+(n+1)$ ez n db egymást követõ szám, ezekre (a jól ismert bizonyítás szerint) rendre igaz, hogy

- $(n+1)!+2=1*2*3*\dots*(n-1)*n*(n+1)+2$, azaz $2*$ valamennyi $+2$, 2 többszöröse, így ami osztható kettővel
- $(n+1)!+3=1*2*3*\dots*(n-1)*n*(n+1)+3$, azaz $3*$ valamennyi $+3$, ami osztható hárommal
- ...
- $(n+1)!+(n-1)=1*2*3*\dots*(n-1)*n*(n+1)+(n-1)$, azaz $(n-1)*$ valamennyi $+(n-1)$, ami osztható $(n-1)$ -el
- $(n+1)!+n=1*2*3*\dots*(n-1)*n*(n+1)+n$, azaz $n*$ valamennyi $+n$, ami osztható n-el
- $(n+1)!+(n+1)=1*2*3*\dots*(n-1)*n*(n+1)+(n-1)$, azaz $(n+1)*$ valamennyi $+(n+1)$, ami osztható $(n+1)$ -el

tehát ebben a sorozatban egy prim nincs, akkor a $(n+1)!+2$ -nél kisebb elsõ prim és a $(n+1)!+(n+1)$ -nél nagyobb elsõ prim között a távolság legalább n!

Az ikerprímszám sejtés azzal foglalkozik, amikor a prímek közötti távolság 2. Azt mondja, hogy az egymástól 2 távolságra lévõ prímek végtelen sokan vannak.

A Brun téTEL azt mondja, hogy az ikerprímszámok reciprokaiból képzett sor összege, azaz a $(1/3+1/5)+(1/5+1/7)+(1/11+1/13)+\dots$ véges vagy végtelen sor konvergens, ami azt jelenti, hogy ezek a törtek összeadva egy határt adnak ki pontosan vagy azt át nem lépve növekednek, ami határ számot B_2 Brun konstansnak

neveznek. Tehát ez nem dönti el a több ezer éve nyitott kérdést, hogy az ikerprímszámok halmaza végtelen-e? Hiszen ha véges sok van és ezek reciprokait összeadjuk, akkor ugyanúgy nem lépjük át a B_2 Brun konstans értékét, mintha végtelen sok lenne, de ezek már csak olyan csökkenő mértékben járulnának hozzá a végtelen sor összegéhez, hogy így sem lépnék át a Brun konstans értékét.

Ebben a példában egy olyan programot készítettünk, amely közelíteni próbálja a Brun konstans értékét. A repó [bhax/attention_raising/Primek_R/stp.r](#) mevű állománya kiszámolja az ikerprímeket, összegzi a reciprokaikat és vizualizálja a kapott részeredményt.

```
# 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.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>
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")
```

Soronként értelemezük ezt a programot:

```
primes = primes(13)
```

Kiszámolja a megadott számig a prímeket.

```
> primes=primes(13)
> primes
[1] 2 3 5 7 11 13
```

```
diff = primes[2:length(primes)]-primes[1:length(primes)-1]

> diff = primes[2:length(primes)]-primes[1:length(primes)-1]
> diff
[1] 1 2 2 4 2
```

Az egymást követő prímek különbségét képzi, tehát 3-2, 5-3, 7-5, 11-7, 13-11.

```
idx = which(diff==2)

> idx = which(diff==2)
> idx
[1] 2 3 5
```

Megnézi a `diff`-ben, hogy melyiknél lett kettő az eredmény, mert azok az ikerprím párok, ahol ez igaz. Ez a `diff`-ben lévő 3-2, 5-3, 7-5, 11-7, 13-11 különbségek közül ez a 2., 3. és 5. indexűre teljesül.

```
t1primes = primes[idx]
```

Kivette a `primes`-ból a párok első tagját.

```
t2primes = primes[idx]+2
```

A párok második tagját az első tagok kettő hozzáadásával képezzük.

```
rt1plust2 = 1/t1primes+1/t2primes
```

Az $1/t1primes$ a $t1primes$ 3,5,11 értékéből az alábbi reciprokokat képzi:

```
> 1/t1primes
[1] 0.33333333 0.20000000 0.09090909
```

Az $1/t2primes$ a $t2primes$ 5,7,13 értékéből az alábbi reciprokokat képzi:

```
> 1/t2primes
[1] 0.20000000 0.14285714 0.07692308
```

Az $1/t1primes + 1/t2primes$ pedig ezeket a törteket rendre összeadja.

```
> 1/t1primes+1/t2primes
[1] 0.5333333 0.3428571 0.1678322
```

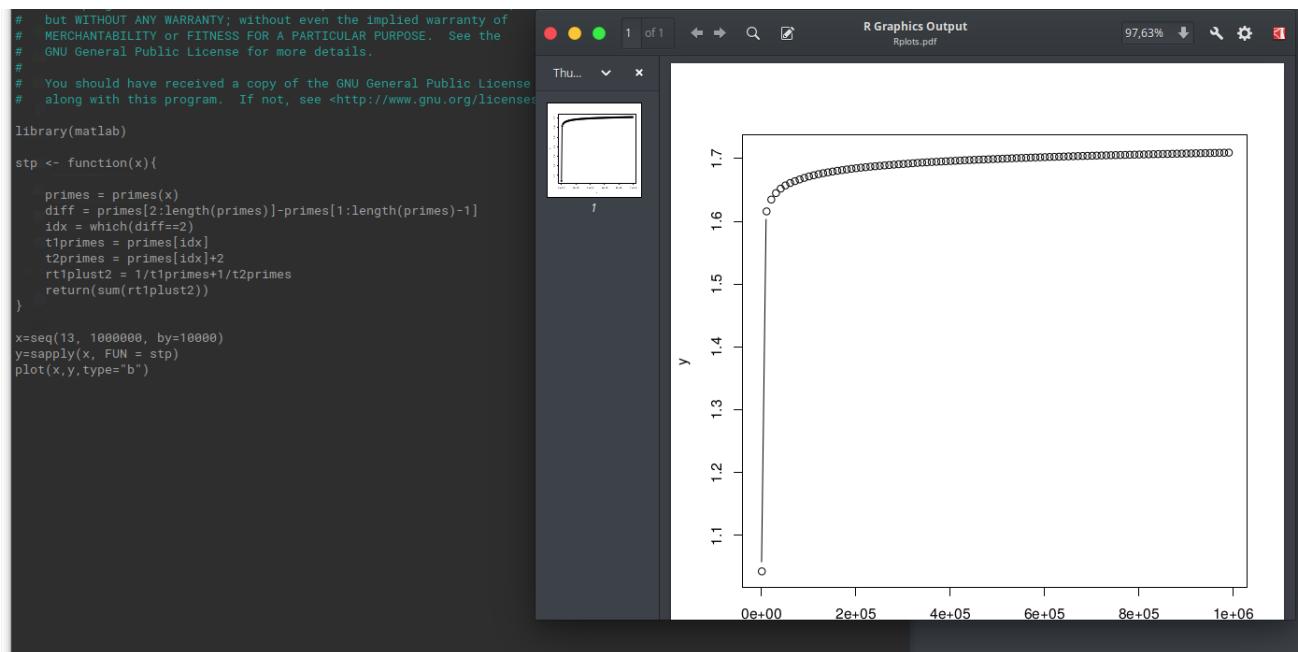
Nincs más dolgunk, mint ezeket a törteket összeadni a `sum` függvényel.

```
sum(rt1plust2)
```

```
> sum(rt1plust2)
[1] 1.044023
```

A következő ábra azt mutatja, hogy a szumma értéke, hogyan nő, egy határértékhez tart, a B_2 Brun konstanshoz. Ezt ezzel a csipettel rajzoltuk ki, ahol először a fenti számítást 13-ig végezzük, majd 10013, majd 20013-ig, egészen 990013-ig, azaz közel 1 millióig. Vegyük észre, hogy az ábra első köre, a 13 értékhez tartozó 1.044023.

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



2.10. ábra. A B_2 konstans közelítése

Werkfilm

- <https://youtu.be/VkMFrgBhN1g>
- <https://youtu.be/aF4YK6mBwf4>

2.9. Vörös Pipacs Pokol/csiga (folytonos mozgási parancsokkal)

Írj olyan Minecraft MALMÖ python programot, amivel Steve csigavonalban járja be a pályát a folytonos mozgási parancsok felhasználásával.

Megoldás videó: Saját megoldás

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Turing/csiga_folytonos.py, <http://hackers.inf.unideb.hu:443/RedFlowerHell/>

Steve a négyzetgörbű pálya minden oldalát bejárja "uthossz" időegységgel, illetve fordul 90 fokot, ha elérte a az oldal szélét. minden negyedik ilyen bejárás után ugrik egyet a következő szintre, és növeljük az "uthossz" változó értékét, hiszen felfelé haladva nő az út hossza. Ezt a folyamatot Steve a lávában való elégésig ismétli.

```
class Steve:
    def __init__(self, agent_host):
        self.agent_host = agent_host

        self.nof_red_flower = 0

    def run(self):
        world_state = self.agent_host.getWorldState()
        uthossz = 2
        # Loop until mission ends:
        while world_state.is_mission_running:
            print("--- nb4tf4i arena -----\\n" -->
)
            self.agent_host.sendCommand( "move 1" )
            time.sleep(uthossz)
            self.agent_host.sendCommand( "turn 1" )
            time.sleep(.5)
            self.agent_host.sendCommand( "turn 0" )
            self.agent_host.sendCommand( "move 1" )
            time.sleep(uthossz)
            self.agent_host.sendCommand( "turn 1" )
            time.sleep(.5)
            self.agent_host.sendCommand( "turn 0" )
            self.agent_host.sendCommand( "move 1" )
            time.sleep(uthossz)
            self.agent_host.sendCommand( "turn 1" )
            time.sleep(.5)
            self.agent_host.sendCommand( "turn 0" )
            self.agent_host.sendCommand( "move 1" )
            time.sleep(uthossz)
            self.agent_host.sendCommand( "jump 1" )
            time.sleep(.5)
            self.agent_host.sendCommand( "jump 0" )
            uthossz = uthossz + 2
            world_state = self.agent_host.getWorldState()
```

3. fejezet

Helló, Chomsky!

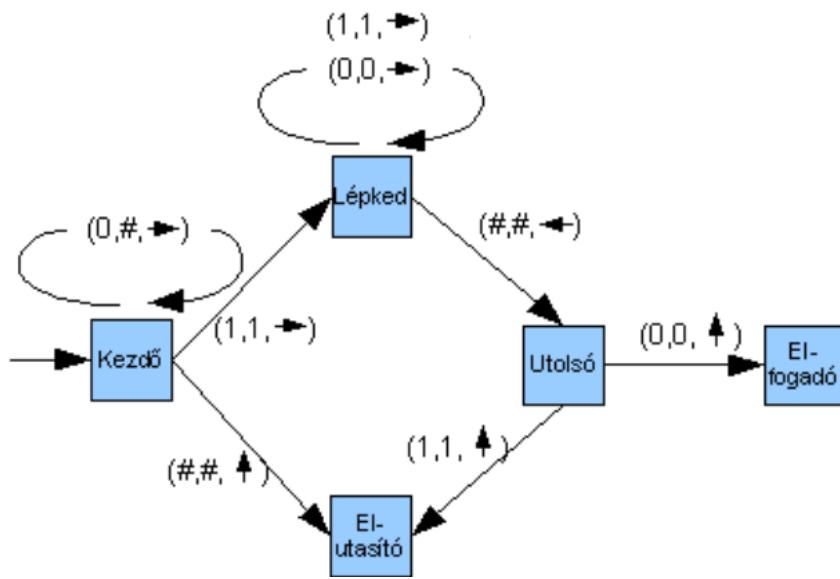
3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html#id511324>

A Turing gép működését Alan Turing angol matematikus dolgozta ki 1936-ban az egyik cikkében. A gép matematikai számítások, algoritmusok és eljárások sorozatából áll. Ez volt a legjobb és legpróbálkozásban leegyszerűsített leírása az akkoriban még nem létező számítógépeknek. Ez a gép modell nagy és fontos támaszt volt az első számítógép, a Colossus megépítésekor, amiben Alan Turing is részt vett.



3.1. ábra. Turing gép

A kezdetbeli, kiinduló ábrán csak a Kezdő, Lépked és Vég állapotok voltak. Az állapotátmenet gráf olvasása: megnézzük, hogy milyen állapotba vagyunk, majd hogy mit olvasunk és hogy mit fogunk írni, aztán

továbbléünk úgy ahogy az él vezet. Hárrom fő jelünk van: 1, 0 és # (üres cella). Ha jól megfigyejük az ábrát akkor rájövünk, hogy a gép sosem fog leállni, tehát ez az egész egy végtelen ciklust fog eredményezni. Végiglépked az input betűin, majd az üres cellákon is, és azt ír vissza amit olvasott. Nézzük meg a Lépked részt, abból másis ki fog derülni. Mint látjuk, ha a gép 0-t olvas, akkor 0-t is fog visszaírni, továbblépni pedig újra csak a Lépked-be fog, tehát visszamegy oda ahonnan jött. Úgyanez a helyzet az 1-es és a #-el is. A Kezdő állapot esetén ha 0-t olvas akkor egy #-et ír és újra a Kezdőbe megy, továbblépni akkor fog ha 1-et kap (1-et fog írni és továbbmegy a Lépkedbe). A végtelen ciklus kikerülése érdekében jött létre a fenti ábra, ahol megjelennek az Elfogadó és Elutasító állapotok is. Ez a gép feladat már komplexebb, el tudja dönteni, hogy az input milyen tulajdonságokkal rendelkezik. Például döntse el, hogy az input szám (melynek a gép bináris kódját vizsgálja) osztható-e 2-vel. Ha igen (tehát ha a bináris kód 0-ra végződik), akkor a gép az Elfogadó állapotba fog megállni, ha pedig nem osztható (a bináris kód 1-re végződik vagy input hiány esetén), akkor az Elutasító státuszban.

Mivel a feladat címében mégicska az szerepel, hogy decimálisból unárisba való átváltás, ezért nézzünk meg egy programot ami ezt meg is valósítja nekünk.

A forráskód megtalálható a következő linken is: [./Forraskodok/Chomsky/3.1/decim.c](#)

```
#include <stdio.h>

int
main()
{
    int a, db=0;
    printf("Adj meg egy decimalis szamot!\n");
    scanf("%d", &a);
    printf("A megadott szam unarisba atváltva:\n");
    for (int i = 0; i < a; i++)
    {
        printf(" | ");
        db++;
        if (db % 5 == 0)
        {
            printf("   ");
        }
    }
    printf("\n");
    return 0;
}
```

Fordítás: **gcc decim.c -o decim**

Futtatás: **./decim**

A decimális, másképp tízes számrendszerbeli számok a legelterjedtebbek, hiszen a úgy a hétköznapi életben, mint a matematikai tanulmányaink során, ezeket használjuk. Az unáris vagy egyes számrendszerbeli számok pedig a legegyszerűbbek, ezeket ábrázolásra használjuk. Egy decimális szám, legyen N, unárisan ábrázolva annyit tesz, hogy egy általunk megválasztott karaktert (a mi programunk esetén: "l") N-szer írunk le valamilyen közönkként törölve a könnyebb olvasás érdekéért (nálunk ez a törlés ötössével történik). Nézzük egy példát: a 8-as decimális szám unárisan a következőképpen néz ki: ||||| |||.

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

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

Megoldás videó:

Megoldás forrása:

A generatív grammatika Noam Chomsky nevéhez kötődik. Úgy vizsgálja a nyelvtant mint az ismeret alapját, hiszen ha nem lenne nyelvtanunk, akkor a tudást nem tudnánk se megörökíteni, se továbbadni. Nézete azt a vállotta, hogy a tudás és az ismeret többnyire öröklött (generációról generációra terjed), vagyis univerzális (gondolván a gyerekekre, akik könnyedén elsajátítják anyanyelvüket). A generatív grammatikának négy fő része van: nemterminális jelek, terminális jelek, helyettesítési/képzési szabályok és mondat/kezdő szimbólumok, illetve három nyelvtan fajtája: környezetfüggő, környezetfüggetlen és reguláris. Nézzünk meg két példát környezetfüggő leírásra (a nyilak jelölik majd a képzési szabályokat)

S, X, Y: „változók” (a nemterminálisok)

a, b, c: „konstansok” (a terminálisok)

$S \rightarrow abc$, $S \rightarrow aXbc$, $Xb \rightarrow bX$, $Xc \rightarrow Ybcc$, $bY \rightarrow Yb$, $aY \rightarrow aaX$, $aY \rightarrow aa$ ($a \leftrightarrow$ helyettesítési szabályok)

S (a mondat szimbólum)

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

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

A, B, C: „változók” (a nemterminálisok)

a, b, c: „konstansok” (a terminálisok)

$A \rightarrow aAB$, $A \rightarrow aC$, $CB \rightarrow bCc$, $cB \rightarrow Bc$, $C \rightarrow bc$ (a képzési szabályok)

S (A kezdőszimbólum)

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

```
A (A → aAB)
aAB ( A → aAB)
aaABB ( A → aAB)
aaaABBB ( A → aC)
aaaaCBBB (CB → bCc)
aaaabCcBB (cB → Bc)
aaaabCBcB (cB → Bc)
aaaabCBBc (CB → bCc)
aaaabbCcBc (cB → Bc)
aaaabbCBcc (CB → bCc)
aaaabbbCccc (C → bc)
aaaabbbbcccc
```

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása:

A C utasítás fogalma BNF-ben való definiálása:

```
<utasítás> ::= <kifejezés> | <összetett_utasítás> | <feltételes_utasítás> | ←
    <while_utasítás> | <do_utasítás> | <for_utasítás> | <switch_utasítás> | ←
    <break_utasítás> | <continue_utasítás> | <return_utasítás> | < ←
    goto_utasítás> | <cimke_utasítás> | <>nulla_utasítás>

<kifejezés> ::= <értékkedás> | <függvényhívás>
<értékkedás> ::= <változó><szám>
<változó> ::= <betű>{<betű>}
<betű> ::= a-z
<szám> ::= <számjegy>{<számjegy>}
<számjegy> ::= 0|1|2|3|4|5|6|7|8|9
<függvényhívás> ::= <típus><függvénynév>
<típus> ::= <betű>{<betű>}
<függvénynév> ::= <betű>{<betű>}

<összetett_utasítás> ::= <deklarációlista> | <utasításlista>
<deklarációlista> ::= <deklaráció>{<deklaráció>}
<deklaráció> ::= <típus><változó>
<utasításlista> ::= <utasítás>{<utasítás>}

<feltételes_utasítás> ::= if<kifejezés><utasítás> | if<kifejezés><utasítás> ←
    else<utasítás>

<break_utasítás> ::= break
```

```
<while_utasítás> ::= while<kifejezés><utasítás> | while<kifejezés><utasítás <→
    ><break_utasítás>

<do_utasítás> ::= do<utasítás>while<kifejezés> | do<utasítás>while< ←
    kifejezés><break_utasítás>

<for_utasítás> ::= for([<kifejezés>] [<kifejezés>] [<kifejezés>])<utasítás> | ←
    for([<kifejezés>] [<kifejezés>] [<kifejezés>])<utasítás><break_utasítás>

<switch_utasítás> ::= switch<kifejezés><utasítás> | switch<kifejezés>< ←
    utasítás><case><kifejezés><default> | switch<kifejezés><utasítás>< ←
    break_utasítás>

<continue_utasítás> ::= continue

<return_utasítás> ::= return | return<kifejezés>

<goto_utasítás> ::= goto<azonosító>

<cimke_utasítás> ::= <azonosító>
<azonosító> ::= <cimke>
<cimke> ::= <betű>{<betű>}

<>nulla_utasítás> ::=
```

Kódcsipetek, melyek C89-ben nem, de C99-ben viszont lefordulnak:

```
for (int i = 0; i < 10; i++) {
    x=i*i;
printf("%d", x);
}
```

A C89-ben amikor megadtuk a ciklusok feltételeit, akkor ezzel egyidőben nem lehetett a változókat is deklarálni, hanem ezeket külön kellett megadni.

```
int csere (int a, int b) {
    a = a+b;
    b = a-b;
    a = a-b;      //változók cseréje
}
```

Ugyanez volt a helyzet a függvények esetében is. A példában szereplő x-et és y-t külön, a függvény deklaráása után, kellett bekérni.

A kódcsipetben látunk egy kommentet is (//változók cseréje). A //-es egy soros kikommentelés is azok a dolgok közé tartozik, amikre a C89 hibát adna futtatáskor.

3.4. Saját lexikális elemző

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

Megoldás videó: https://youtu.be/9KnMqrkj_kU (15:01-től).

Megoldás forrása:

A lexer egy olyan program amelyik megír, összeállít nekünk egy C programot. A lex programok úgymond több részből tevődnek össze, ezeket a részeket %%-al vannak elválasztva egymástól. A mi programunk most 3 részből áll. A programot ezen részletek alapján vizsgáljuk meg.

A forráskód megtalálható a következő linken is: [../Forraskodok/Chomsky/3.4/realnumber.l](#)

```
% {  
#include <stdio.h>  
int realnumbers = 0;  
%}  
digit [0-9]  
%%
```

A kapcsoszárójelek közötti részt a lexer, úgy ahogy van egy az egyben berakja a C programba. Az első része ismerős a C kódokból, az **include**-dal deklaráljuk az stdio header fájlt. Ezután egy deklarálás és egyben egy értékadás jön (a **realnumber** változó kezdetben a 0-ás értéket kapja). A a **realnumber** változóval azt számoljuk majd, hogy hány számot olvas be a program. A kapcsoszárójelezett rész után jönnek a definíciók, a mi esetünkben is van egy: a **digit** nevű definícióval egy szám csoportot adunk meg (0-tól 9-ig tartalmazza a számokat, tehát a számjegyeket tárolja). Szöglletes zárójelek között egy karaktereket tudunk megadni, de mi most mivel számokat írtunk közé, ezért nem karaktereket, hanem számokat fog tárolni. Majd mindenek után, %%-al lezárjuk ezt a részt.

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

Ebben, a második részben, jelennie meg a fordítási szabályok, ahol már használjuk is a definícióknál megadtakat, tehát egyfajta használati utasítást adunk meg hozzjuk. A **{digit}*** azt jelenti, hogy a digitből lehet nulla vagy bármennyi darab. Az informatikában a **.** karakter magában azt jelenti, hogy amit szeretnénk azt bármilyen karakterre rá lehet illeszteni. De mivel mi most a lexel azt akarjuk elérni, hogy tokeneket, valós számokat ismerjen fel (tehát olyan formájú számokat, mint például a 3.5, vagyis szám.szám), ezért a **".**-ot le kell védeni, amit a / jel segítségével érjük el. A levédett pont után egy újabb kifejezés következik (**{digit}+**), ami azt takarja hogy a pont után is számjegyek jönnek, bármennyi de legalább egy mindenépp (+). Ha ez a leírás megegyezik az egyik intup adattal, tehát ha talál egy ilyet, akkor (jön egy utasítássorozat ami {}-ek közé van téve) növeli a **realnumber** értékét, és ezt kiírja a képernyőre először stringként (%s), melyet el is tárolunk az **yytext**-be, majd számként kiiratjuk (%f), az **atof** függvényel az **yytext**-ben lévő stringet átkonvertáljuk double típusú számmá.

```
int  
main ()  
{
```

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

Az utolsó rész pedig már úgymond maga a tényleges program, ahol meghívjuk az előbb definiált lexikális yylex függvényt, elemzést és miután ez véget ér, akkor kiiratom a valós számok számát, vagyis a realnumber értékét.

C forráskód létrehozása: **lex -o realnumber.c realnumber.l**

Fordítás: **gcc realnumber.c -o realnumber -lfl**

Futtatás: **./realnumber**

Egy kép mely illusztrálja a programot működés közben (ha nem akarunk több inputot adni a programnak akkor a Ctrl+D-vel lelőjük)

```
3.4$ ./realnumber
The number of real numbers is 0
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Chomsky/
3.4$ ./realnumber
jhgfdhgfd
jhgfdhgfd

aksjdkfajskdfjkasdf
aksjdkfajskdfjkasdf
329fööjasdkfjasdf9j31fasdf
[realnum=329 329.000000]fööjasdkfjasdf[realnum=9 9.000000][realnum=31 31.000000]fasdf
333
[realnum=333 333.000000]
1.2.3.4
[realnum=1.2 1.200000][realnum=.3 0.300000][realnum=.4 0.400000]
The number of real numbers is 7
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Chomsky/
3.4$
```

3.2. ábra. Lexikális elemző

3.5. Leetspeak

Lexelj össze egy l33t cipher!

Megoldás videó: https://www.youtube.com/watch?v=06C_PqDpD_k

Megoldás forrása: [../nbatfai/bhax/blob/master/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/l337d1c7.1](#)

A Leet egy egyfajta al-abcje az angol nyelvnek, amit legtöbbször az internetes fórumokon, játékokban, tehát többnyire az informatika területén használják. A l33t a latin abc betűit, az ascii karakterek kombinációira cseréli fel, így létrejönnek az érdekesebbénél érdekesebb, furcsa, de ötletes átírások, megnevezések.

A forráskód megtalálható a következő linken is: [../Forraskodok/Chomsky/3.5/l33tsp3ak.l](#)

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

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

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

{'a', {"4", "4", "@", "/-\\"}}, {'b', {"b", "8", "|3", "|{}"}}, {'c', {"c", "(", "<", "{}"}}, {'d', {"d", "|)", "|]", "|{}"}}, {'e', {"3", "3", "3", "3"}}, {'f', {"f", "|=", "ph", "|#"}}, {'g', {"g", "6", "[", "[+"}}, {'h', {"h", "4", "|-", "|-", "[-]"}}, {'i', {"1", "1", "|", "!"}}, {'j', {"j", "7", "_|", "_/"}}, {'k', {"k", "|<", "1<", "|{"}}, {'l', {"l", "1", "|", "|_"}}, {'m', {"m", "44", "(V)", "|\\/|"}}, {'n', {"n", "|\\|", "/\\/", "/V"}}, {'o', {"0", "0", "()", "[]"}}, {'p', {"p", "/o", "|D", "|o"}}, {'q', {"q", "9", "O_", "(,)"}}}, {'r', {"r", "12", "12", "|2"}}, {'s', {"s", "5", "$", "$"}}, {'t', {"t", "7", "7", "'|'"}}}, {'u', {"u", "|_|", "(_)", "[_]"}}, {'v', {"v", "\\\/", "\\\/", "\\\/"}}}, {'w', {"w", "VV", "\\\/\\\/", "(/\\\)"}}, {'x', {"x", "%", ")(", ")("}}, {'y', {"y", "", "", ""}}, {'z', {"z", "2", "7_", ">_"}}, {'0', {"D", "0", "D", "0"}}, {'1', {"I", "I", "L", "L"}}, {'2', {"Z", "Z", "Z", "e"}}, {'3', {"E", "E", "E", "E"}}, {'4', {"h", "h", "A", "A"}}, {'5', {"S", "S", "S", "S"}}, {'6', {"b", "b", "G", "G"}}, {'7', {"T", "T", "j", "j"}}, {'8', {"X", "X", "X", "X"}},
```

```
{'9', {"g", "g", "j", "j"}}

// https://simple.wikipedia.org/wiki/Leet
};

%}
%%
```

Pont úgy, mint az előző lexiális elemző feladatban, a program szerkezete, felépítése úgyanaz (a három fő rész). Először is megadjuk a header fájloka, majd szintén következnek a definíciók. A definícióknál deklarálunk egy L337SIZE-ot, ami segítségével meg tudjuk majd határozni az input hosszát. Egy struktúra segítségével létrehozzuk a cipher-t ami megkapja a karaktert (amit majd átír), illetve a négy lehetőséget, stringet (amire majd cseréli). Majd ebből a struktúrából létrehozunk egy 1337d1c7 nevű tömböt (nem adjuk meg az elemeit, azt majd a program megszámol magának), aminek be is adagoljuk az elemeit a struktúra felépítettsége szerint (először a latin betű, majd pedig a négy lehetőség amire cserélheti a program). Ezek vége is az program első részének.

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

        if(1337d1c7[i].c == tolower(*yytext))
        {

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

            if(r<91)
                printf("%s", 1337d1c7[i].leet[0]);
            else if(r<95)
                printf("%s", 1337d1c7[i].leet[1]);
            else if(r<98)
                printf("%s", 1337d1c7[i].leet[2]);
            else
                printf("%s", 1337d1c7[i].leet[3]);

            found = 1;
            break;
        }

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

A második részben egy for ciklussal végig megyünk a kapott inputon. Az i-edig (tehár majd rendre minden egyik) latin karaktert átalakítja kisbetűvé, megkeresi az adott listában, random számot generál neki (körül-

belül 0-100 közöttit), majd az if résznél lévő vizsgálatok alapján (a random számot vizsgálva) dönti el hogy a latin betűnek melyik megfelelőjét írja ki a négy közül. Ha a random szám kisebb mint 91, akkor az első karaktert választja (az elsőnek a legnagyobb a valószínűsége), ha 91 és 94 közötti, akkor a második lehetőséget, ha 94 és 97 közötti, akkor harmadik lehetőséget, ha pedig 97-től nagyobb, akkor az utolsó karaktert választja és írja ki.

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

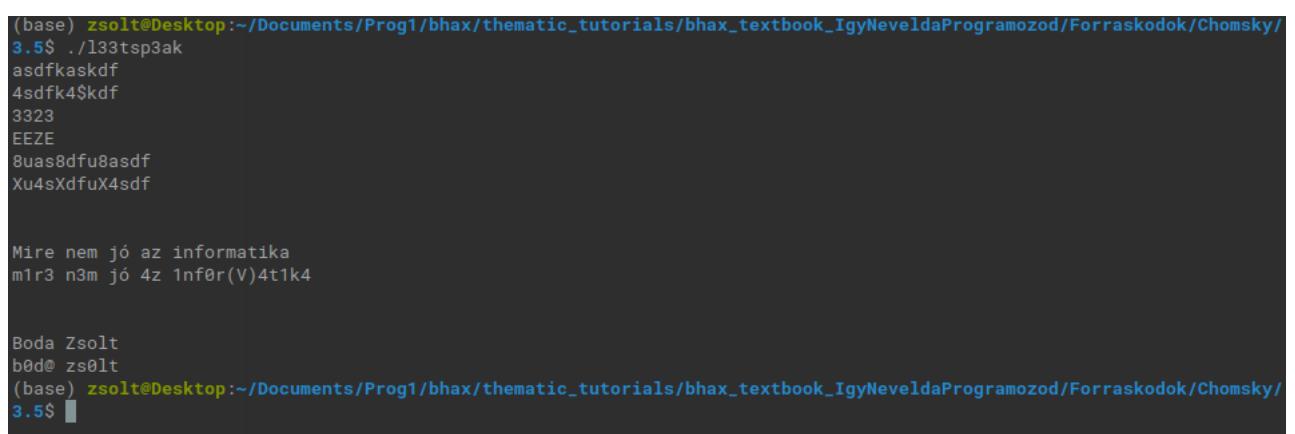
A program harmadik és egyben utolsó része egy c programrészlet, ahol a `yylex()` függvényhívással elindítjuk az input átváltoztatását.

C forráskód létrehozása: **lex -o l33tsp3ak.c l33tsp3ak.l**

Fordítás: **gcc l33tsp3ak.c -o l33tsp3ak -lfl**

Futtatás: **./l33tsp3ak**

Egy kép mely illusztrálja a programot működés közben (ha nem akarunk több inputot adni a programnak akkor a Ctrl+D-vel lelőjük)



```
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Chomsky/
3.5$ ./l33tsp3ak
asdfkaskdf
4sdfk4$kdf
3323
EEZE
8uas8dfu8asdf
Xu4sXdfuX4sdf

Mire nem jó az informatika
m1r3 n3m jó 4z 1nf0r(V)4t1k4

Boda Zsolt
b0d@ zs0lt
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Chomsky/
3.5$
```

3.3. ábra. Leetspeak

3.6. A források olvasása

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 jelkezelő függvény kezelje. (Miután a **man 7 signal** lapon megismertem a SIGINT jelet, a **man 2 signal** lapon pedig a használt rendszerhívást.)

Bugok

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

i.

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

Az if megvizsgálja, hogy a SIGINT figyelem kívül volt-e hagyva eddig, ha igen akkor maradjon is figyelem kívül, ha pedig nem volt akkor a *jelkezelo* függvény tegyen róla, kezelje.

ii.

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

Ez a for ciklus addig dolgozik az *i*-vel, amíg hamissá nem válik az a feltétel, hogy *i* kisebb mint az 5, *i*-t egyesével növeli (a ciklus 5 alaklomma fog lefordulni). Ha kiíratjuk az *i*-t a for cikluson belül, akkor a következő eredményt kapjuk: 0 1 2 3 4. A **++i** inkrementálja az *i* értékét, és ezt az értéket is tériti vissza, viszont ez, ha az egész for cikluson belül zajlik le, akkor nem látszik meg az eredményen.

iii.

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

Ez a for ciklus sokban hasonlít az előzőhez. Ha kiíratjuk az *i*-t a for cikluson belül, akkor most is a következő eredményt kapjuk: 0 1 2 3 4. A **i++** szintén inkrementálja az *i* értékét, de az eredeti, a növelés előtti értéket adja vissza, nem különbözik az előző eredménytől. Az hogy az *i*-t milyen formában növeljük, nem befolyásolja azt hogy a for ciklus hányszor fut le.

iv.

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

A for ciklus elméletben 5-ször fodul le, *i*-t 0-ról indítva. A *tomb* *i*-edik eleme mindenkor az *i* egyel nagyobb értékét veszi fel. Ez a ciklus első ránézésre furcsa már az utolsó argumentum miatt, és tényleg mert ez így egy programba ültetve nem fog lefutni.

v.

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

A for ciklus *i*-vel 0-ról indul, egyesével növelve ezt addig fut le amíg az *i* kisebb mint az *n* és amíg az a feltétel igaz, hogy a ***d++** egyenlő a ***s++**-al.

vi.

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

A **printf** függvény segítségével kiíratunk a standard kimenetre két egész értéket amelyeket az **f** függvény kétszeri meghívása ad vissza.

vii.

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

A `printf` függvény segítségével kiíratunk a standard kimenetre két egész értéket, először az `f` függvény visszatérítési értékét, abban az esetben, amikor az a változót kapja paraméterül, illetve magát az a értékét.

viii.

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

A `printf` függvény segítségével kiíratunk a standard kimenetre két egész értéket, először `f` függvény visszatérítési értékét, abban az esetben, amikor az a változó memóriacímét kapja paraméterül illetve magát az a értékét.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

Minden valós szám estén létezik egy másik olyan valós szám, ami nagyobb tőle és prím (tehát minden számtól van nagyobb prím).

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

Minden valós szám esetén létezik egy másik olyan valós szám ami nagyobb tőle, prím és a tőle kettővel nagyobb valós szám is prímszám (tehát véges sok ikerprím van).

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

Létezik olyan y valós szám minden x valós szám esetén, hogy ha az x prímszám, akkor az x kisebb mint az y (tehát véges sok prímszám van).

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

Létezik olyan y minden x esetén, hogy ha y kisebb mint az x, akkor az x nem prímszám (tehát ugyanaz mint az előző esetben, vagyis hogy véges sok prímszám van).

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

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

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket. / Mit vezetnek be a programba a következő nevek?

- egész

```
int a;
```

- egészre mutató mutató

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

- egész referenciaja

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

- egészek tömbje

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

- egészek tömbjének referenciaja (nem az első elemé)

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

- egészre mutató mutatók tömbje

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

- egészre mutató mutatót visszaadó függvény

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

- egészre mutató mutatót visszaadó függvényre mutató mutató

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

- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény

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

- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

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

3.9. Vörös Pipacs Pokol/csiga (diszkrét mozgási parancsokkal)

Írj olyan Minecraft MALMÖ python programot, amivel Steve csigavonalban járja be a pályát a diszkrét mozgási parancsok felhasználásával.

Megoldás videó: [Megoldó video](#)

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Chomsky/csiga_diszkret.py, <http://hackers.inf.unideb.hu:443/RedFlowerHell/>

A feladat nem változott a folytonos mozgásos csigához képest, minden össze az irányítást kell átszabni a diszkrét mozgási parancsokhoz. A while cikluson belül egy for ciklus lépteti Steve-et blokkonként kezdetben 9 blokkal, majd minden egyes "táv" megtétele után egyel nő a szamla, aminek értékét egy

`if` vizsgálja. Ha 4-gyel osztva 0 az érték, az azt jelenti, körbeértünk, tehát Steve ugrik egyet, előre megy még egy blokknyit és elfordul, a megteendő "távolságot" tároló uthossz változónk nő. Egyéb esetben csak fordul egyet és növeli a megtett oldalakat számoló szamlalo változót. Ezzel elérünk, hogy a diszkrét mozgási parancsokkal közel ugyanazt az utat járjuk be, mint a 2.9 feladatban.

```
class Steve:
    def __init__(self, agent_host):
        self.agent_host = agent_host

        self.nof_red_flower = 0

    def run(self):
        world_state = self.agent_host.getWorldState()
        uthossz = 9
        szamlalo = 0
        while world_state.is_mission_running:
            for i in range(uthossz):
                self.agent_host.sendCommand( "move 1" )
                time.sleep(.5)
            szamlalo = szamlalo + 1
            if szamlalo % 4 == 0:
                self.agent_host.sendCommand( "jumpmove 1" )
                time.sleep(.5)
                self.agent_host.sendCommand( "move 1" )
                time.sleep(.5)
                self.agent_host.sendCommand( "turn 1" )
                time.sleep(.5)
            uthossz = uthossz + 4
        else:
            self.agent_host.sendCommand( "turn 1" )
            time.sleep(.5)
        world_state = self.agent_host.getWorldState()
```

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

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

Megoldás videó: <https://youtu.be/1MRTuKwRsB0>, <https://youtu.be/RKbX5-EWpzA>.

Megoldás forrása: nbatfai/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramCaesar/tm.c

A forráskód megtalálható a következő linken is: [./Forraskodok/Caesar/4.1/tm.c](https://Forraskodok/Caesar/4.1/tm.c)

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

int
main ()
{
    int nr = 5;
    double **tm;

    printf("%p\n", &tm);

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

A program első soraiban létrehozunk egy `nr` változót, melynek értéket is adunk: 5. Ez az ötös szám arra utal, hogy a program futtatását követően kiírt alsó háromszögmátrix (az előző sorban eg elem lesz, a másodikban kettő, stb.) 5 soros lesz. A `double **tm` végrehajtódásakor deklaráljuk a `tm` nevű változót, a *-al jelezvén hogy ez egy pointer, egy mutató, és a program le is foglal neki monjuk 8 bájtnyi tárhelyet. Ezek után ki íratjuk a `tm` memóriacímét. A következőkben szerepel a `malloc` függvény, amelyik (mint a `man 3 malloc` paracs lekérése után is kiderül), helyet foglal a memóriában és egy `void *` pointert ad vissza, ami bármire mutathat, majd mi megadunk meg egy típust, hogy arra mutasson amire mi akarjuk.

A `malloc` paraméterül megkapja hogy mekkora területet kell lefoglaljon, most egy `sizeof` paramétert kap, ami a `double *` típus helyigényét adja vissza (tehát $\text{nr} * \text{sizeof}(\text{double} *) = 5 * 8 = 40$ bájtot kell lefoglaljon a `tm` számára). Az if-el at ellenőrizzük, hogy a `malloc` sikeresen lefoglalta-e a helyet a memóriában és hogy sikeresen vissza adta-e a `void *` mutatót. Ha ez nem sikerült, akkor egyenlő a `NULL`-al, vagyis nem mutat seholá és kilép a programból (ez akkor szokott bekövetkezni amikor nincs elegendő tárhelyünk és nem sikerül a memória foglalás). Tegyük fel, hogy nekünk ez most sikerült (ezért ki is íratjuk a `tm` `malloc` által visszaadott címét), ezért menjünk is tovább a program többi részére (a lefoglalt területet az ábrán a középső sor jelöli).

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

}
printf ("%p\n", tm[0]);
```

A következőkben egy for ciklussal úgymond végigmegyünk az 5 soron és minden egyik sorban úrja lesz egy memória foglalás. Nézzük pédának hogy mit fog csinálni abban az esetben amikor az `i=3`: mint mondottam a `malloc` egy `void *` pointert ad vissza, de mi kikötjük hogy ez nekünk most `double *` legyen. A `malloc` a `tm[2]-nek` most $(2+1 * 8)$ bájtot foglal, vagyis a harmadik sorban három 8 bájtnyi helyet foglal le, a harmadik `double *` egy olyan sorra mutasson ahol három doubleket van lefoglalva hely (az ábrán ezt az utolsó sor urolsó három cellára jelöli). Ez az egész egy if feltételeként szerepel, melyben ismét ellenőrzi hogy sikeres volt-e a pointer létrehozása és a hely lefoglalása, ha nem akkor egyenlő a `NULL`-al és megáll a program. De tegyük fel újra, hogy sikerrel jártunk, ezért kiíratjuk a `tm[0]`, az első sor memóriacímét, majd megyünk tovább.

```
for (int i = 0; i < nr; ++i)
    for (int j = 0; j < i + 1; ++j)
        tm[i][j] = i * (i + 1) / 2 + j;

for (int i = 0; i < nr; ++i)
{
    for (int j = 0; j < i + 1; ++j)
        printf ("%f, ", tm[i][j]);
    printf ("\n");
}
```

Egy következő for ciklussal megszerkesztjük az alsó háromszögmátrixot (ehhez két for ciklus szükséges, mivel így tudjuk meghatározni az elem sorát és oszlopát), azáltal, hogy az egyes, sor- és oszlopindexekkel jelölt elemek értéket kapnak (az `i` `nr`-ig, a `j` pedig `i`-ig megy, így biztosítva az alsó háromszögmátrix alakot). Lássuk, hogy hogy működik az értékadás, amikor az `i=1` és `j=0`: ekkor a második sor első elemének adunk egy $(1 * 2 / 2 + 0 =)$ 1-es (`double`) értéket. Az értékadások végeztével, újabb for ciklusok segítségével kiíratjuk a kapott alsó háromszögmátrixot.

```
tm[3][0] = 42.0;
(* (tm + 3))[1] = 43.0; // mi van, ha itt hiányzik a külső ()
```

```
* (tm[3] + 2) = 44.0;
* (* (tm + 3) + 3) = 45.0;

for (int i = 0; i < nr; ++i)
{
    for (int j = 0; j < i + 1; ++j)
        printf ("%f, ", tm[i][j]);
    printf ("\n");
}
```

Ebben a részben az első négy sor különböző leírások arra, hogy a negyedik sorban lévő négy értéket hogy változtassuk meg, tehát példa arra, hogy hogyan lehet hivatkozni a különböző elemekre ahhoz hogy valamit tudunk velük csinálni (ezesetben az értékmódosítás művelet végrehajtását). Az értékek megváltoztatása után újra kiíratjuk az alsó háromszögmátrixunkat.

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

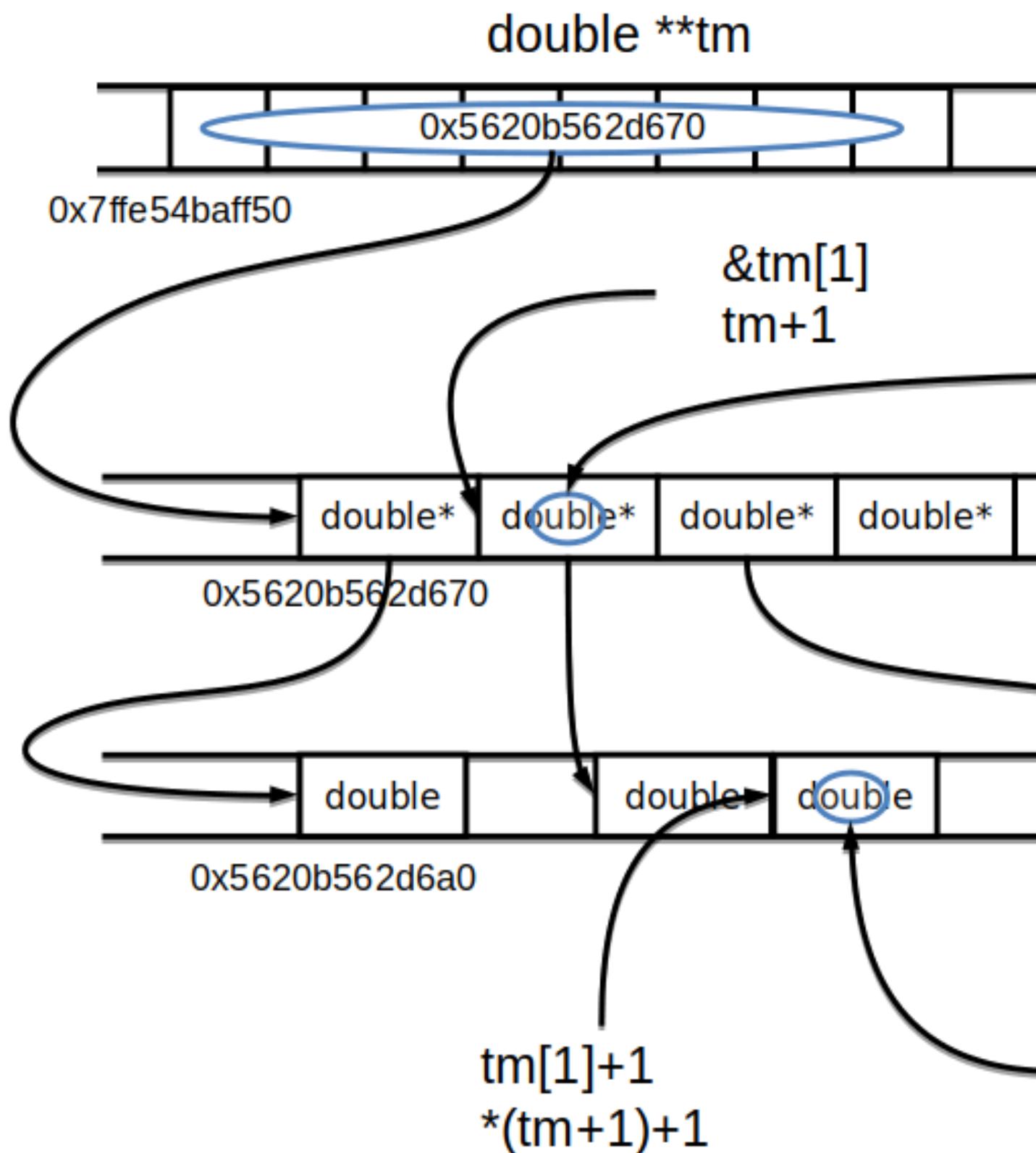
free (tm);

return 0;
}
```

A program utolsó részében a `free` függvény segítségével felszabadítjuk az egyes sorokban illetve az egész `tm` által foglalt memóriát.

Fordítás: **gcc tm.c -o tm**

Futtatás: **./tm**



4.1. ábra. A double ** háromszögmátrix a memóriában

```
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Caesar/4
.1$ ./tm
0x7ffd1e78c550
0x56380e776670
0x56380e7766a0
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
6.000000, 7.000000, 8.000000, 9.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
42.000000, 43.000000, 44.000000, 45.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Caesar/4
.1$
```

4.2. ábra. A program által kiírt eredmény

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása:

A forráskód megtalálható a következő linken is: [../Forraskodok/Caesar/4.2/e.c](#)

```
&#xA0;#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define MAX_KULCS 100
#define BUFFER_MERET 256

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

    char kulcs[MAX_KULCS];
    char buffer[BUFFER_MERET];

    int kulcs_index = 0;
    int olvasott_bajtok = 0;

    int kulcs_meret = strlen (argv[1]);
    strncpy (kulcs, argv[1], MAX_KULCS);

    while ((olvasott_bajtok = read (0, (void *) buffer, BUFFER_MERET)))
    {

        for (int i = 0; i < olvasott_bajtok; ++i)
```

```
{  
  
    buffer[i] = buffer[i] ^ kulcs[kulcs_index];  
    kulcs_index = (kulcs_index + 1) % kulcs_méret;  
  
}  
  
    write(1, buffer, olvasott_bajtok);  
  
}  
}
```

Egy **tiszta.szöveg** nevű fájl létrehozása, egy szöveg beletétele.

Fordítás: **gcc e.c -o e -std=c99**

Futtatás: **/e kulcs <tiszta.szöveg>titkos.szöveg** (a kulcs bármi lehet)

Dekódolás: **/e kulcs <titkos.szöveg** (a kulcs meg kell egyezzen a fent használt kulccsal)

Az exor titkosító, mint ahogy a neve is mondja, exor művelet segítségével szöveget titkosít a számunkra. A bevitt, titkosítandó szöveg emberi fogyasztásra alkalmas, de a titkosított viszont már annál kevésbé. A dekódolásnál lévő parancs segítségével képesek vagyunk a titkos szövegből visszaadni az eredeti szöveget. A program elején deklarálunk egy kulcsot, egy buffert (ami az olvasott bajtok egyes bajtjait veszi fel), majd bekérjük az olvasott bajtokat (a mi esetünkben a tiszta.szöveg nevű fájlból). A program egyessével végigmegy a beolvasható bajtokon, majd azokat össze xorozza az adott kulcs indexel ami a kulcsban következik, majd a kulcsban belül is továbblét a következő karakterre (ha a kulcs 4 karakter akkor sorban minden azon a 4 karakteren megy végig). Ezért játszik fontos szerepet a kulcs, mivel anélkül nem tudjuk feltörni a titkosított szöveget.

A program működését laboron kipróbáltuk, csoportban dolgozva átírtuk a Tanár Úr által kért kulcs karakterekre, majd az UDPORG Facebook csoportba posztoltuk és küldtük be a válaszokat.

Ebben a feladatban a következő személyek voltak a Tutoriáltjaim: Imre Dalma (<https://gitlab.com/imdal/bhax>), Fürjes-Benke Péter (<https://gitlab.com/fupn26/bhax>). Illetve Tutorom volt: Imre Dalma, Fürjes-Benke Péter.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

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

A forráskód megtalálható a következő linken is: [..Forraskodok/Caesar/4.3/exortitkosito.java](#)

A következő java program egy titkosító és egyben egy törő program is, mivel ha újra végrehajtjuk a titkosítást akkor visszakapjuk az eredeti szöveget. Ezt a műveletet az exortitkosito osztályban végezzük el.

```
public class exortitkosito {
```

```
public exortitkosito(String kulcsSzöveg,
                      java.io.InputStream bejövőCsatorna,
                      java.io.OutputStream kimenőCsatorna)
                      throws java.io.IOException {

    byte [] kulcs = kulcsSzöveg.getBytes();
    byte [] buffer = new byte[256];
    int kulcsIndex = 0;
    int olvasottBájt = 0;

    while((olvasottBájt =
           bejövőCsatorna.read(buffer)) != -1) {

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

            buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);
            kulcsIndex = (kulcsIndex+1) % kulcs.length;

        }

        kimenőCsatorna.write(buffer, 0, olvasottBájt);

    }

}

public static void main(String[] args) {
    try {
        new exortitkosito(args[0], System.in, System.out);
    } catch(java.io.IOException e) {
        e.printStackTrace();
    }
}
}
```

Mint mindegyik titkosító/törő programban van egy ciklus melyik apránként olvassa a bemenetet, ebben az esetben egy while ciklus, mely tömbönként olvas be. A titkosítás folyamata teljesen az mint az előző C exor titkosító programban. A `kulcsIndex`-et ráállítjuk a kulcs adott karakterére ami következik, majd ezen és a `buffer` által beolvasott karakteren elvégezzük a kizáró vagy műveletet. Az eredmény a `buffer` tömbbe kerül, a program végén ezt íratjuk majd ki.

A program futtatása (titkosítandó szöveg bekérése, titkosítása és visszaváltoztatása):

javac exortitkosito.java (létrejön az `exortitkosito.class` fájl)

java exortitkosito alma > titkositott.txt (a titkosítandó szöveg bevitele)

more titkositott.txt (kiíratódik a titkosított szöveg)

java exortitkosito alma < titkositott.txt (dekódoljuk a titkosított szöveget, visszakapva az eredetit)

```
.3$ java exortitkosito alma > titkos.txt
nagyon titkos dolog
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Formulas/4.4/titkos
.3$ more titkos.txt

[[M]enu
k
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Formulas/4.4/titkos
.3$ [[
```

4.3. ábra. A program működése

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása:

A forráskód megtalálható a következő linken is: [..//Forraskodok/Caesar/4.4/t.c](#)

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

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

double
atlagos_szohossz (const char *titkos, int titkos_meret)
{
    int sz = 0;
    for (int i = 0; i < titkos_meret; ++i)
        if (titkos[i] == ' ')
            ++sz;

    return (double) titkos_meret / sz;
}
```

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

    double szohossz = atlag_szohossz (titkos, titkos_meret);

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

A program elején lesz három nagyon fontos és beszédes definícióink: A titkos szöveg maximum mérete, a beolvasott buffer mérete és a kulcs mérete, ami, az előző programmal szemben, itt már nem lehet akármi, hanem ezt a program végén fogjuk ezt iterálni, most annyit tudunk hogy 8 hosszúságú. A `atlag_szohossz` függvénynek is beszédes neve van, ezt a függvényt a következő függvényben már használjuk is ami nemmás mint a `tiszta_lehet`. Ez a függvény úgymond megtippeli, hogy a szöveg amit kapott az megfelel-e a magyar nyelv néhány iratlan szabályának, mint például az hogy tartalmazza-e a leggyakoribb magyar szavakat (hogy, nem, az, ha), mivel ritka az az értelmes magyar nyelven íródott szöveg amikben ezek egyike legalább nem fordul elő.

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

    int kulcs_index = 0;

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

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

    }

}
```

A követező függvény az ami a tiszta szövegből titkosat csinál az exor művelet segítségével, ugyanazzal a módszerrel mint az Exor titkosító programban. A program egyessével végigmegy a beolvasott bájtokon, majd azokat össze xorozza az adott kulcs indexel ami a kulcsban következik, majd a kulcson belül is továbblép a következő karakterre.

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

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

    return tiszta_lehet (titkos, titkos_meret);

}
```

Nagy meglepődésünkre a törő függvény nem rendelkezik külön erre a célra kifejlesztett algoritmussal, hanem felhasználja az előbb bemutatott `exor` függvényt, mivel ha valamit exorozunk egy adott kulccsal, majd ugyanazzal a kulccsal újra megcsináljuk az `exor` műveletet akkor visszakapjuk az eredeti szöveget. Az `exor` eme tulajdonságát kihasználva az `exor_toro` elvégzi a titkos szöveg visszaváltottatását

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

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

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

    // osszes kulcs eloallitasa
    for (int ii = '0'; ii <= '9'; ++ii)
        for (int ji = '0'; ji <= '9'; ++ji)
            for (int ki = '0'; ki <= '9'; ++ki)
    for (int li = '0'; li <= '9'; ++li)
        for (int mi = '0'; mi <= '9'; ++mi)
            for (int ni = '0'; ni <= '9'; ++ni)
                for (int oi = '0'; oi <= '9'; ++oi)
    for (int pi = '0'; pi <= '9'; ++pi)
    {
        kulcs[0] = ii;
        kulcs[1] = ji;
        kulcs[2] = ki;
        kulcs[3] = li;
        kulcs[4] = mi;
        kulcs[5] = ni;
        kulcs[6] = oi;
        kulcs[7] = pi;
```

```

if (exor_tores (kulcs, KULCS_MERET, titkos, p - titkos))
    printf
("Kulcs: [%c%c%c%c%c%c%c] \nTiszta szöveg: [%s]\n",
ii, ji, ki, li, mi, ni, oi, pi, titkos);

// ujra EXOR-ozunk, így nem kell egy masodik buffer
exor (kulcs, KULCS_MERET, titkos, p - titkos);
}

return 0;
}

```

Következik a main, a főprogram ahol az előbbieket felhasználjuk. Beolvassuk a titkos szöveget, és itt látszik egy nagy különbség az előző C exor titkosító programmal, ez nem foglal le külön területet a tiszta illetve a titkos szövegnek, tehát a kódolt és a dekódolt szövegnek. A titkos szöveg bekérése után, megnézi hogy a kisebb e a mérete mint amennyit az elején lefoglalt neki a MAX_TITKOS-ba, ha igen akkor a maradék helyet kinullázza. Ezután előállítja a kulcsokat majd jöhet az exorozás, tehát a törés és a titkosítás művelete.

Fordítás: **gcc t.c -o t**

Futtatás: **./t**

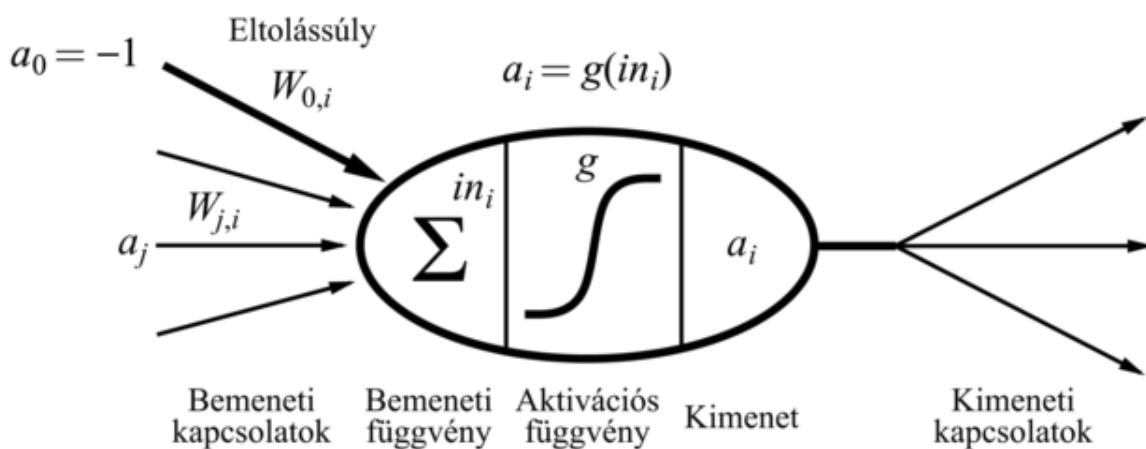
4.5. Neurális OR, AND és EXOR kapu

R

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

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

A neurális hálók irányított (bemeneti és kimeneti) kapcsolatokkal összekötött egységekből állnak. Az a(0-j) az inputról (más idegsejtekből) bejövő axonok, kapcsolatok (az "a"-k). Ezekből az "a"-kból rendre csinál egy eltolási súlyt, a W-t, ami meghatározza a kapcsolat erősséget és előjelét, majd ezt a megfelelő "a"-val összeszummázza, tehát i a bemenetek egy súlyozott összege lesz. A kapott összegre alkalmazunk egy aktivációs függvényt (g), ezáltal megkapva a kimeneti kapcsolatot.



4.4. ábra. A neurális háló illusztrálása

A forráskód megtalálható a következő linken is: [./Forraskodok/Cesar/4.5/neutoae.r](#)

```
# 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.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>
#
# https://youtu.be/Koyw6IH5ScQ

library(neuralnet)

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

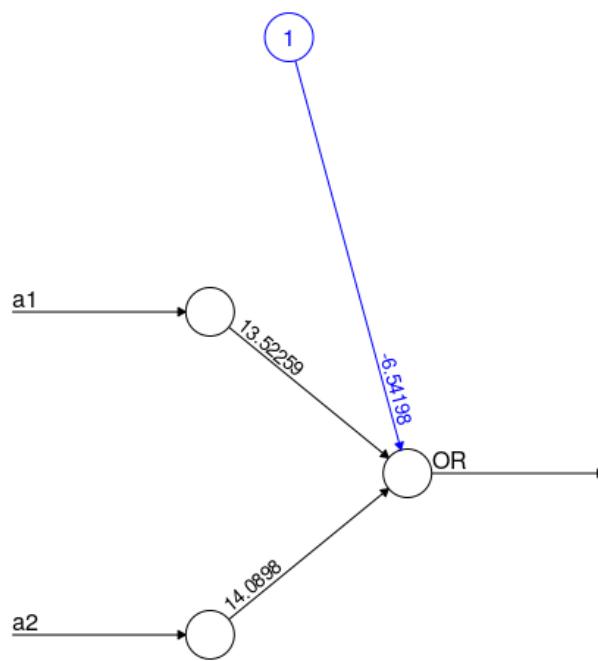
or.data <- data.frame(a1, a2, OR)

nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE, ←
                  stepmax = 1e+07, threshold = 0.000001)

plot(nn.or)

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

Az a1 és az a2-ben megadtunk 0-k és 1-sek sorozatát majd az OR-ba az "a1 or a2" művelet eredményét, majd ebből adatot csinálunk az or.data változóba. Az nn.or változóba használjuk a neuralnet függvényt, amely kiszámolja nekünk a neutrális hálót. Átadjuk neki az adatokat (amit az előbb előállítottunk az a1, a2 és az OR-ból), tehát megtanítjuk neki ezt az OR műveletet, majd saját magát továbbfejlesztve beállítja a súlyokat úgy hogy megtanulja a dolgok menetét. Fontos hogy a neuralnet függvényt nem a logikai műveletet végzi el, hanem tényleges tanulás után probálja megmondani az eredményt nekünk. A plot függvényel kirajzoltatjuk az első csatolt képen látható ábrát, ami illusztrálja a számításokat. Itt látható az ismétlések száma, hogy hányszor végezte a műveletet (steps) és hogy ezekből hányszor kapott rossz eredményt (errors, nagyon kicsi). A **compute** parancsal beadjuk neki az adatokat és számításra utasítjuk a függvényünket, tehát kikérdezzük tőle azt amit megtanítottunk neki az elején. És ahogy a második kép mutatja ügyesen megtanulta és vissza is adta a helyes eredményeket.



Error: 2e-06 Steps: 140

4.5. ábra. Neurális OR kép

```

+      bhax:git      x      4.5:Rscript      ...NeveldaProgramozod:make
neutoae.r Rplots1.pdf Rplots2.pdf Rplots3.pdf Rplots.pdf
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Caesar/4.5$ Rscript neutoae.r
dev.new(): using pdf(file="Rplots4.pdf")
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1
[4,] 1  1  1

$net.result
[,1]
[1,] 0.001294704
[2,] 0.999710396
[3,] 0.999010074
[4,] 1.000000000

dev.new(): using pdf(file="Rplots5.pdf")
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1
[4,] 1  1  1

$net.result
[,1]      [,2]
[1,] 4.340504e-05 2.589296e-09
[2,] 9.999737e-01 1.313634e-03
[3,] 9.999675e-01 1.247277e-03
[4,] 1.000000e+00 9.984262e-01

dev.new(): using pdf(file="Rplots6.pdf")
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1
[4,] 1  1  1

$net.result
[,1]
[1,] 0.5000041
[2,] 0.5000006
[3,] 0.5000006
[4,] 0.4999970

dev.new(): using pdf(file="Rplots7.pdf")
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1

```

4.6. ábra. Neurális OR eredmény

```

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

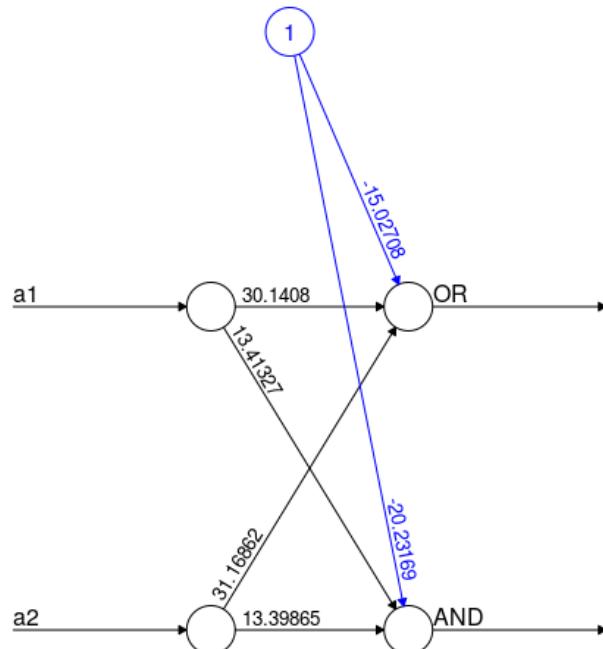
operand.data <- data.frame(a1, a2, OR, AND)

nn.operand <- neuralnet(OR+AND~a1+a2, operand.data, hidden=0, linear.output= FALSE, stepmax = 1e+07, threshold = 0.000001)

```

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

A következő rész annyiban különbözik az előző résztől, hogy most az OR művelet mellett az AND műveletet is megtanítjuk a programnak, majd mind a kettőt kiíratjuk. Mind a két képen láthatjuk, hogy az OR mellé társul az AND is (egy hálózatban lehet két kimenet is). A második kép bizonyítja, hogy az AND műveletet is tökéletesen elsajtottotta.



Error: 2e-06 Steps: 319

4.7. ábra. Neurális AND kép

```

4.5: Rscript
+       bhax.git      x     4.5: Rscript      ...NeveldaProgramozod: make
neutoae.r  Rplots1.pdf  Rplots2.pdf  Rplots3.pdf  Rplots.pdf
(base) zsoldt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Caesar/4.5$ Rscript neutoae.r
dev.new(): using pdf(file="Rplots4.pdf")
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1
[4,] 1  1  1

$net.result
[,1]
[1,] 0.001294704
[2,] 0.999710396
[3,] 0.999010074
[4,] 1.000000000

dev.new(): using pdf(file="Rplots5.pdf")
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1
[4,] 1  1  1

$net.result
[,1]      [,2]
[1,] 4.340504e-05 2.589296e-09
[2,] 9.999737e-01 1.313634e-03
[3,] 9.999675e-01 1.247277e-03
[4,] 1.000000e+00 9.984262e-01

dev.new(): using pdf(file="Rplots6.pdf")
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1
[4,] 1  1  1

$net.result
[,1]
[1,] 0.5000041
[2,] 0.5000006
[3,] 0.5000006
[4,] 0.4999970

dev.new(): using pdf(file="Rplots7.pdf")
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1

```

4.8. ábra. Neurális AND eredmény

```

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

exor.data <- data.frame(a1, a2, EXOR)

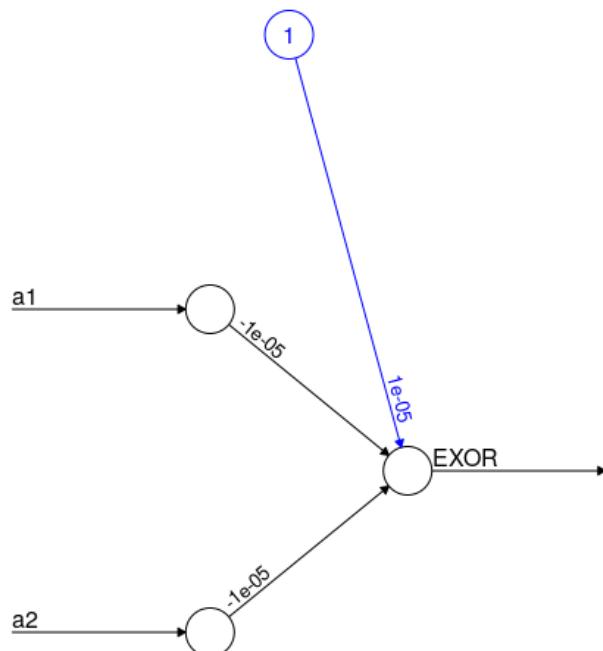
nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=0, linear.output=FALSE,   ↵
                      stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

```

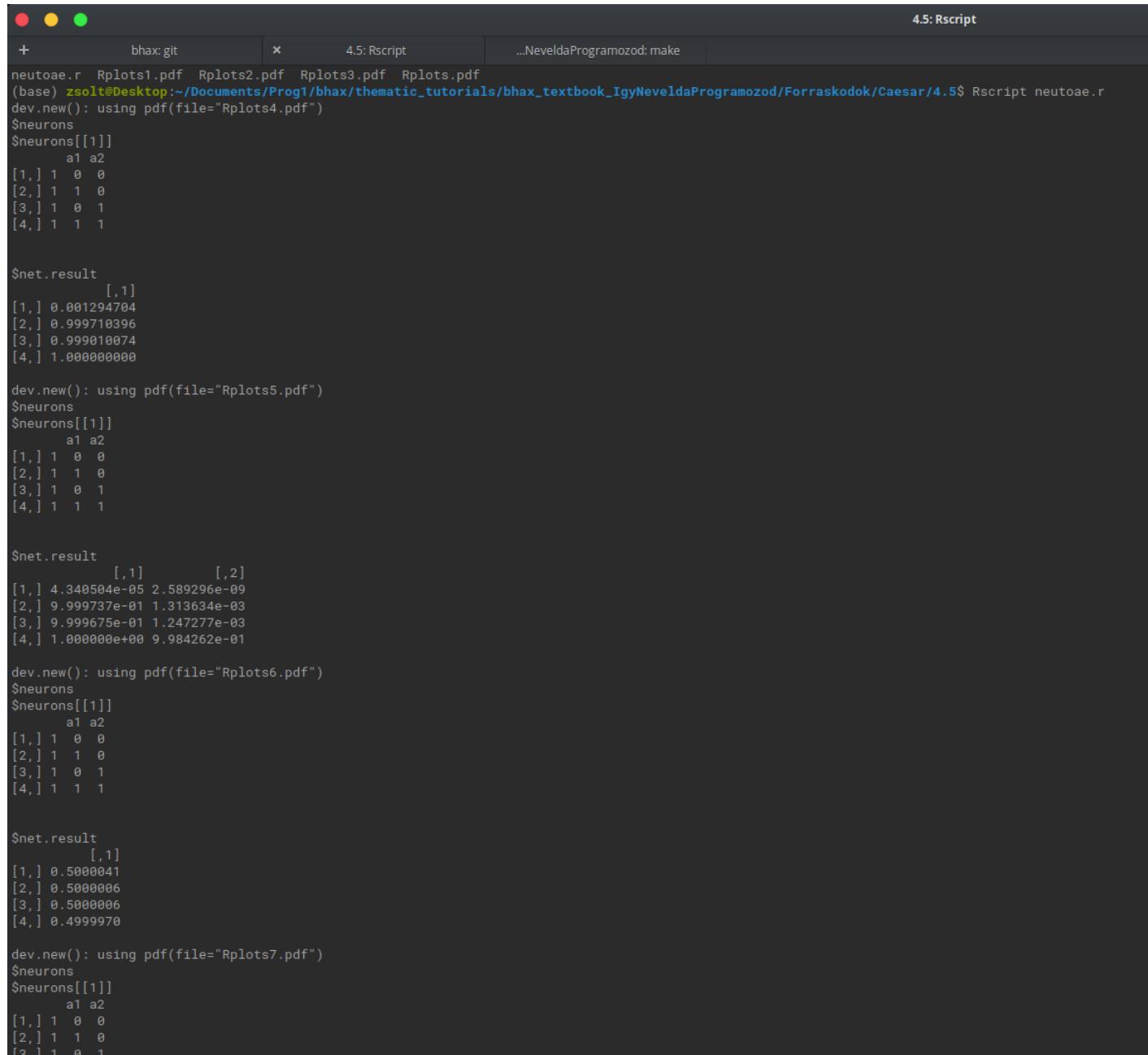
```
compute(nn.exor, exor.data[,1:2])
```

A harmadik részben kipróbaljuk, hogy a program képes elsajátítani-e az exor műveletet is. Ugyanazzokkal a lépésekkel tanítjuk meg neki a dolgokat. A futtatás után viszont nemvárt eredményt kapunk: a program helytelen eredményt adott vissza. Ez volt az a nagy katasztrófális hiba ami miatt sok időre le is fordultak a neurális háló tanulmányozásától. Az első képen látszik hogy a hiba az előzőekhez képest elég nagy. A második képen pedig az tűnik fel hogy a helyes eredmények helyett (0 vagy 1) mindeütt egy 0.5 közeli eredményt kapunk, nem tudja eldönteni hogy melyik a jó, 50-50% esélyt ad mindkettőnek.



Error: 0.5 Steps: 98

4.9. ábra. Neurális EXOR kép



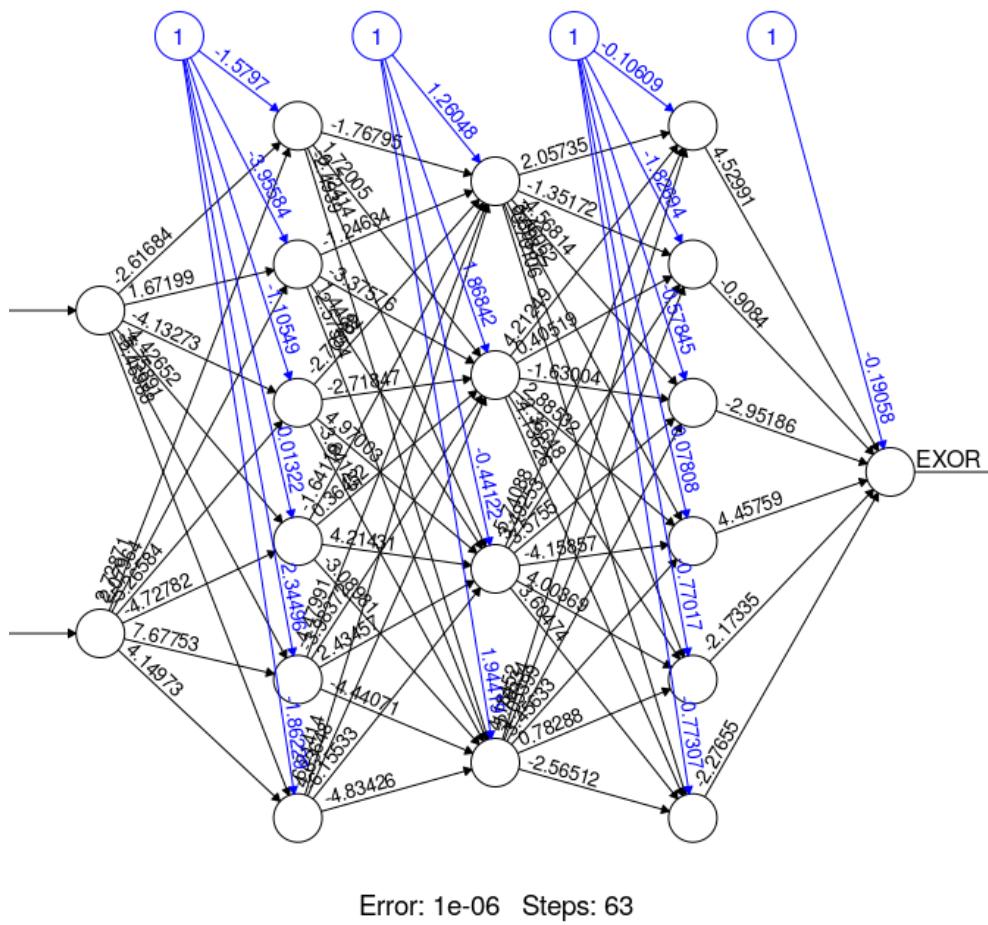
```
4.5: Rscript  
+       bhax:git      x     4.5: Rscript      ...NeveldaProgramozod:make  
neutoae.r Rplots1.pdf Rplots2.pdf Rplots3.pdf Rplots.pdf  
(base) zsolt@Desktop:~/Documents/Prog1/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Caesar/4.5$ Rscript neutoae.r  
dev.new(): using pdf(file="Rplots4.pdf")  
$neurons  
$neurons[[1]]  
  a1 a2  
[1,] 1  0  0  
[2,] 1  1  0  
[3,] 1  0  1  
[4,] 1  1  1  
  
$net.result  
[,1]  
[1,] 0.001294704  
[2,] 0.999710396  
[3,] 0.999010074  
[4,] 1.000000000  
  
dev.new(): using pdf(file="Rplots5.pdf")  
$neurons  
$neurons[[1]]  
  a1 a2  
[1,] 1  0  0  
[2,] 1  1  0  
[3,] 1  0  1  
[4,] 1  1  1  
  
$net.result  
[,1]      [,2]  
[1,] 4.340504e-05 2.589296e-09  
[2,] 9.999737e-01 1.313634e-03  
[3,] 9.999675e-01 1.247277e-03  
[4,] 1.000000e+00 9.984262e-01  
  
dev.new(): using pdf(file="Rplots6.pdf")  
$neurons  
$neurons[[1]]  
  a1 a2  
[1,] 1  0  0  
[2,] 1  1  0  
[3,] 1  0  1  
[4,] 1  1  1  
  
$net.result  
[,1]  
[1,] 0.5000041  
[2,] 0.5000006  
[3,] 0.5000006  
[4,] 0.4999970  
  
dev.new(): using pdf(file="Rplots7.pdf")  
$neurons  
$neurons[[1]]  
  a1 a2  
[1,] 1  0  0  
[2,] 1  1  0  
[3,] 1  0  1
```

4.10. ábra. Neurális EXOR eredmény

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

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

A negyedik példa az exor javított változata. A kettő közti különbség a rejtett neutronok számánál van: az előző feladatban ez 0 volt (hidden=0), most pedig egyből három értéket kap egy lista által, többrétegű lesz (hidden=c(6, 4, 6)). Mint látjuk, az eredményeknél itt már sokkal több értéket ír ki, de most mind jó értékeket, a képen pedig szintúgy látható a bal szélen az a1 és az a2, majd a az első rejtett (6 neuron), a második (4 neuron) és a harmadik rejtett réteg (6 neuron).



4.11. ábra. Neurális EXOR kép

4.6. Hiba-visszaterjesztéses perceptron

C++

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

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

Ez a program sokban kötődik az előző programhoz, mivel ebben is egyfajta tanítási módszert mutatunk be. A perceptronok, másnéven egyrétegű előrecsatolt neurális hálók, olyan hálók amelyekben az összes bemenet közvetlenül a kimenetekre kapcsolódik (minden súly csak egy kimenetre van hatással). A perceptronok

az alap logikai műveleteken túl képesek a bonyolultabbakat is bemutatni röviden. A hiba-visszaterjesztést többrétegű perceptron (MLP) esetén tudjuk csak alkalmazni. A többrétegű perceptron rétegekbe szervezett neuronokból áll. A rétegek mennyisége többnyire változó, minden esetben van egy bemeneti-, egy kimeneti- és a kettő között egy vagy több rejtett réteg. A következő programok együttese olyan algoritmus, amelyik megtanítja a gépnek a bináris osztályozást.

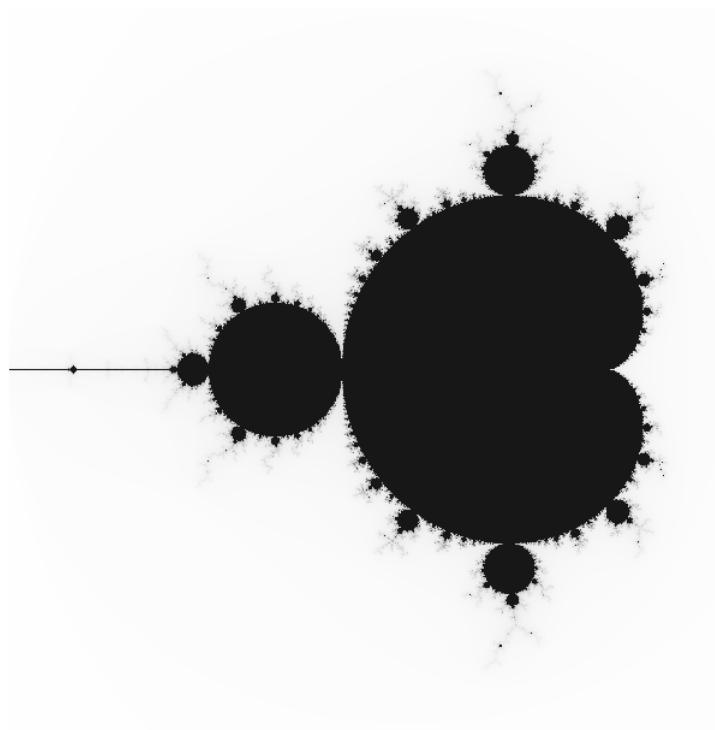
A forráskód megtalálható a következő linken: [./Forraskodok/Cesar/4.6/mandelpng.cpp](#)

A program itt nem kap nagy, részletező bemutatást, mivel a következő fejezetben erről illetve csak ilyenekről lesz majd szó.

Fordítás: **g++ mandelpng.cpp -o mandel -lpng**

Futtatás: **./mandel mandel_perceptron.png**

A kép megtekintése: **eog mandel_perceptron.png**



4.12. ábra. Mandelbrot kép

Szükségünk van még a következő programra, amelyik megtalálható a következő linken: [./Forraskodok/Cesar/4.6/perceptron.hpp](#)

A forráskód megtalálható a következő linken: [./Forraskodok/Cesar/4.6/main.cpp](#)

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

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

```
int size = png_image.get_width() * png_image.get_height();  
  
Perceptron* p = new Perceptron (3, size, 256, 1);  
  
double* image = new double[size];  
  
for (int i = 0; i<png_image.get_width(); ++i)  
    for (int j = 0; j<png_image.get_height(); ++j)  
        image[i*png_image.get_width() + j] = png_image[i][j].red;  
  
double value = (*p) (image);  
  
std::cout << value << std::endl;  
  
delete p;  
delete [] image;  
}
```

A fentebb említett perceptron.cpp programot a mainben meg is hívjuk header fájlként, átláthatóbbá téve a főprogramot. A fő számítások viszont a perceptron.hpp-ben vannak, a mainben az ott deklarált Perceptron osztályt hívjuk segítségül meg az eredmény kiszámolásának céljából. A mainben a png.hpp header fájl segítségével létrehozunk egy új png kiterjesztésű képet, úgyanolyan szélességgel és magasséggel mint a mandelbrotos kép volt. A két egymásbaágazódó for ciklus segítségével végigmegyünk a kép minden pixelén és az előzőekben lementett mandel_perceptron.png pixeleinek piros (red) komponenseit rámásoljuk a most létrehozott kép pixeleire. A program végén pedig kiíratjuk ezt a percceptor értéket a value változó segítségével.

Fordítás: **g++ perceptron.hpp main.cpp -o main -lpng -std=c++11**

Futtatás: **./main mandel_perceptron.png**

A kiadott eredmény: **0.731044**

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

Írj olyan Minecraft MALMÖ python programot, ami kiírja, hogy Steve milyen blokkokat érzékel maga előtt!

Megoldás videó: [Megoldó videó](#)

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Caesar/mitlatsteve.py, <http://hackers.inf.unideb.hu:443/RedFlowerHell/>

A belinkelt forrás alapján "felkosítjuk" az előző fejezetben megírt kódunkat, így Steve mozgás közben érzékeli az őt körülvevő blokkokat. Ezek is mind kiírathatók, de az alábbi kódban ki vannak kommentelve, így csak a LineOfSight látható, ami mindig azt a blokkot jelenti, amelyikre Steve néz. Ez fog nekünk segíteni abban, hogy bizonyos dolgok láttán különböző cselekvéseket rendelhessünk a karakterhez.

```
class Steve:  
    def __init__(self, agent_host):
```

```
self.agent_host = agent_host

self.nof_red_flower = 0

def run(self):
    world_state = self.agent_host.getWorldState()
    uthossz = 9
    szamlalo = 0
    self.agent_host.sendCommand( "look 1" )
    while world_state.is_mission_running:
        for i in range(uthossz):
            self.agent_host.sendCommand( "move 1" )
            time.sleep(.23)
            world_state = self.agent_host.getWorldState()

        if world_state.number_of_observations_since_last_state != ←
            0:

            sensations = world_state.observations[-1].text
            #print("      sensations: ", sensations)
            observations = json.loads(sensations)
            nbr3x3x3 = observations.get("nbr3x3", 0)
            #print("      3x3x3 neighborhood of Steve: ", nbr3x3x3)

            if "Yaw" in observations:
                self.yaw = int(observations["Yaw"])
            if "Pitch" in observations:
                self.pitch = int(observations["Pitch"])
            if "XPos" in observations:
                self.x = int(observations["XPos"])
            if "ZPos" in observations:
                self.z = int(observations["ZPos"])
            if "YPos" in observations:
                self.y = int(observations["YPos"])

            #print("      Steve's Coords: ", self.x, self.y, self.z)
            #print("      Steve's Yaw: ", self.yaw)
            #print("      Steve's Pitch: ", self.pitch)

            if "LineOfSight" in observations:
                lineOfSight = observations["LineOfSight"]
                self.lookingat = lineOfSight["type"]
                print("      Steve's <): ", self.lookingat)

    szamlalo = szamlalo + 1
    if szamlalo % 4 == 0:
        self.agent_host.sendCommand( "jumpmove 1" )
        time.sleep(.5)
        self.agent_host.sendCommand( "move 1" )
        time.sleep(.5)
```

```
    self.agent_host.sendCommand( "turn -1" )
    time.sleep(.5)
    uthossz = uthossz + 4
else:
    self.agent_host.sendCommand( "turn -1" )
    time.sleep(.5)
world_state = self.agent_host.getWorldState()
```

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

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

Megoldás forrása: banax.attention_raising/CUDA/mandelpngt.cpp nevű állománya.

Az ebben a fejezetben szereplő programok, mindenkorban szervesen kötődnek egymáshoz. Ha sikerül egy képet megcsinálni akkor az összes programot át tudjuk rá ültetni, tehát egy másik programváltozattal tudunk benne majd nagyítani, vagy úgy is át tudjuk majd írni, hogy a GPU-t használja, ezáltal nagyon meggyorsítva a számítási folyamatokat.

A forráskód megtalálható a következő linken: [..../Forraskodok/Mandelbrot/5.1/mandelpngt.cpp](https://www.inf.unideb.hu/~batfai/mandelpngt.cpp)

```
// mandelpngt.cpp
// Copyright (C) 2019
// Norbert Bátfai, batfai.norbert@inf.unideb.hu
//
// 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.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <https://www.gnu.org/licenses/>.
//
// Version history
//
// Mandelbrot png
// Programozó Páternoszter/PARP
```

```
// https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063 ←
//_01_parhuzamos_prog_linux
//
// https://youtu.be/gvaqijHlRUs
//
#include <iostream>
#include "png++/png.hpp"
#include <sys/times.h>

#define MERET 600
#define ITER_HAT 32000
```

A program elején, mint minden, megadjuk a header fájlokat. Ez esetben ami ismeretlen lehet az a png++/png.hpp és a sys/times.h fejlécek. Az elsőre azért van szükségünk, hogy létre tudjuk hozni a képet, hogy viziálisan is láthassuk a Mandelbrot halmazt eredményét, ami kép. A második header fájl pedig az idővel kapcsolatos számításokat, az idővel összefüggő függvények meghívását biztosítja számunkra. Idlletve efníciő megadaások is szerepelnek, melyek majd könnyítik a dolgunkat a program írása alatt.

```
void
mandel (int kepadat [MERET] [MERET]) {

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

    // számítás adatai
    float a = -2.0, b = .7, c = -1.35, d = 1.35;
    int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;

    // a számítás
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, reZ, imZ, ujreZ, ujimZ;
    // Hány iterációt csináltunk?
    int iteracio = 0;
    // Végigzongorázzuk a szélesség x magasság rácsot:
    for (int j = 0; j < magassag; ++j)
    {
        //sor = j;
        for (int k = 0; k < szelesseg; ++k)
        {
            // c = (reC, imC) a rács csomópontjainak
            // megfelelő komplex szám
            reC = a + k * dx;
            imC = d - j * dy;
            // z_0 = 0 = (reZ, imZ)
            reZ = 0;
            imZ = 0;
```

```
iteracio = 0;
// z_{n+1} = z_n * z_n + c iterációk
// számítása, amíg |z_n| < 2 vagy még
// nem értük el a 255 iterációt, ha
// viszont elértük, akkor úgy vesszük,
// hogy a kiinduláci c komplex számra
// az iteráció konvergens, azaz a c a
// Mandelbrot halmaz eleme
while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
{
    // z_{n+1} = z_n * z_n + c
    ujreZ = reZ * reZ - imZ * imZ + reC;
    ujimZ = 2 * reZ * imZ + imC;
    reZ = ujreZ;
    imZ = ujimZ;
    ++iteracio;
}

kepadat[j][k] = iteracio;
}
}

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;

}
```

A mandel függvény az ami majd kiszámolja nekünk a Mandelbrot halmazt. Egy időszámítás jön, ami segítségevel majd a program lefutás után kiírja azt az időt amit igénybe vett a számolás (az én gépen és egy i3-3227U-s processzor esetén ez körülbelül egy 15 másodpercbe telt). Ezek és a változódeklarálások után jöhetnek a nagyobb számítások. Létrehozunk egy dx szélességű és dy magasságú rácsot a két egymásba ágyazódó for ciklus segítségével, majd megvizsgáljuk, hogy a c komplex szám (mely megkapja a reC és a imC értékek által meghatározott pontot, pixelt) benne van-e a Mandelbrot halmazban, ha igen akkor azt a pontot beszínezi.

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

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

    int kepadat [MERET] [MERET];
```

```
mandel(kepadat);

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

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

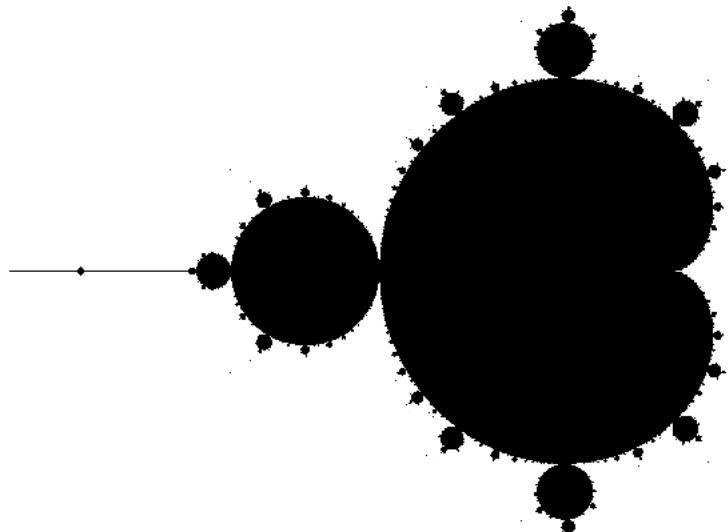
kep.write (argv[1]);
std::cout << argv[1] << " mentve" << std::endl;
```

A főprogram elején megvizsgáljuk, hogy a felhasználó jól használta-e a futtatási parancsot, tehát hogyha a parancs nem két argumentumból áll, akkor kiírunk egy hibaüzenetet a kimenetre. A kepadat változóba bekérjük a méreteket, tehát a kép szélességét és a magasságát majd ezeket átadjuk a mandel függvénynek, ami kiszámolja nekünk a mandelbrot halmazt, és ez alapján megalkotjuk a png kiterjesztésű képet. Miutána program befejezte a számításokat és sikeresen megcsinálta a képet, kiírja a kimenetre a "mentve" üzenetet a felhasználó számára, tudatva a sikert.

Fordítás: **g++ mandelpngt.cpp -lpng -O3 -omandelpngt**

Futtatás: **./mandelpngt mt.png**

Kép megnyitása terminálból: **eog mt.png**



5.1. ábra. A kiadott kép

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

Írj olyan C++ programot, amely kiszámolja a Mandelbrot halmazt!

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

Megoldás forrása: A **Mandelbrot halmaz** pontban vázolt ismert algoritmust valósítja meg a repó [bhax/attention_raising/Mandelbrot/3.1.2.cpp](#) nevű állománya.

A feladatban tutora voltam Schachinger Zsolt-nak.

Felvetődik egy kérdés: melyiket a szám amelyiket önmagával megszorozva 9-et kapunk (természetesen ez a 3). Ebből kiindulva a következő kérdés pedig az, hogy melyik az a szám amit ha megszorozunk önmagával -9-et kapunk? Ez pedig már nem lehetséges a valós számhalmazon, így jönnek képbe a komplex számok, melyek úgymond a valós számhalmaz továbbbővítése. A komplex számok alapja az "i" szám, melynek értéke a $\sqrt{-1}$, ennek a segítségével el lehet végezni a negatív számból való négyzetgyökvonást. Így már meg tudjuk válaszolni az előbb feltett kérdést, az "i" szám segítségével már ki tudjuk hozni a -9-et (tehát $3i^2$).

Ez a program és az előző közötti legnagyobb különbség az hogy a `c` amit vizsgálunk, hogy benne van-e a Mandelbrot halmazban, az előző programban egy változó, ebben pedig egy állandó. Így itt a `c` a rács minden vizsgálandó pontját befutja.

A forráskód megtalálható a következő linken is: [..../Forraskodok/Mandelbrot/5.1/mandelbrotcomplex.cpp](#)

// Verzio: 3.1.2.cpp

```
// Forditas:  
// g++ 3.1.2.cpp -lpng -O3 -o 3.1.2  
// Futtatas:  
// ./3.1.2 mandel.png 1920 1080 2040 ←  
// -0.01947381057309366392260585598705802112818 ←  
// -0.0194738105725413418456426484226540196687 ←  
// 0.7985057569338268601555341774655971676111 ←  
// 0.798505756934379196110285192844457924366  
// ./3.1.2 mandel.png 1920 1080 1020 ←  
// 0.4127655418209589255340574709407519549131 ←  
// 0.4127655418245818053080142817634623497725 ←  
// 0.2135387051768746491386963270997512154281 ←  
// 0.2135387051804975289126531379224616102874  
// Nyomtatás:  
// a2ps 3.1.2.cpp -o 3.1.2.cpp.pdf -l --line-numbers=1 --left-footer=" ←  
// BATF41 HAXOR STR34M" --right-footer="https://bhaxor.blog.hu/" --pro= ←  
// color  
// ps2pdf 3.1.2.cpp.pdf 3.1.2.cpp.pdf.pdf  
//  
//  
// Copyright (C) 2019  
// Norbert Bátfai, batfai.norbert@inf.unideb.hu  
//  
// 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.  
//  
// This program is distributed in the hope that it will be useful,  
// but WITHOUT ANY WARRANTY; without even the implied warranty of  
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
// GNU General Public License for more details.  
//  
// You should have received a copy of the GNU General Public License  
// along with this program. If not, see <https://www.gnu.org/licenses/>.  
  
#include <iostream>  
#include "png++/png.hpp"  
#include <complex>  
  
int  
main ( int argc, char *argv[] )  
{  
  
    int szelesseg = 1920;  
    int magassag = 1080;  
    int iteraciosHatar = 255;  
    double a = -1.9;  
    double b = 0.7;
```

```
double c = -1.3;
double d = 1.3;

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

A program elején a header fájlok deklarálása után (iostream, a cin, cout miatt főleg; png++/png.hpp, a kép létrehozása miatt; complex, a komplex iterációk és ahogy a címben is említve van a komplex osztályos megoldás miatt) jönnek a változók deklarálása, ezek a változók (szelesseg, magassag, iteraciosHatar, a, b, c és d) a futtaási parancsban is fontos szerepet játszanak hiszen az első két argumentum után (ami a ./futtató fájl neve illetve egy png féjlnév, ami a mentett kép neve lesz) ahogy deklarálva vannak, úgynilyen sorrenbe vannak megadva szóközzel elválasztva egymástól. Miután a változók deklarálva lettek, megvizsgáljuk hogy a felhasználó jól futtat-e a programot, tehét hogyha a parans 9 argumentumból áll akkor megadjuk a programban, hogy az egyes változók hányadik értéket kapják meg a parancsból, tehát hogy a program tudja hogy az egyes értékek mit jelentenek számára. Ha viszont a parancs nem 9 argumentumból áll, akkor kiíratunk az outputra egy üzenetet a felhasználónak, hogy tudja a helyes futtatási használatot, **return -1**-el pedig az operációs rendszernek is jelezük a hibát.

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

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

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

for ( int j = 0; j < magassag; ++j )
{
    for ( int k = 0; k < szelesseg; ++k )
    {
        reC = a + k * dx;
        imC = d - j * dy;
        std::complex<double> c ( reC, imC );

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

```
iteracio = 0;

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

    ++iteracio;
}

kep.set_pixel ( k, j,
                png::rgb_pixel ( iteracio%255, (iteracio*iteracio -->
) %255, 0 ) );
}

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

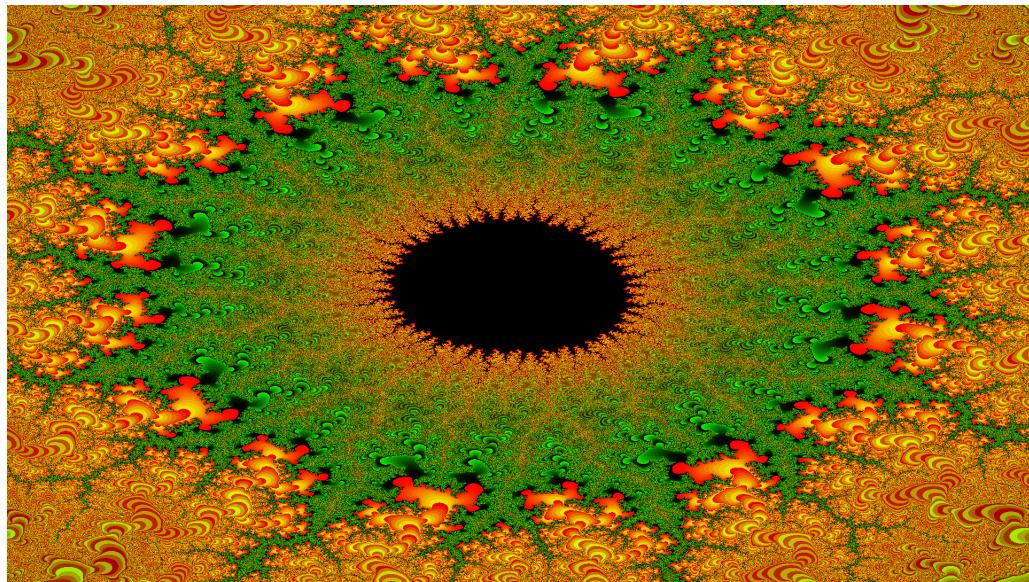
kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}
```

A számítások részben deklaráljuk a `reC`, `imC`, `reZ` és `imZ` (valós és imaginárius részű) változókat, melyekből már látszik is hogy a feladatban sokszerepet fognak játszani a komplex számok. A `c=(reC, imC)` a haló rácspontrajainak megfelelő aktualis pont. Úgyanúgy mint az előző programban a két for ciklus segítségével megcsináljuk a rácsot, elindulunk az origóban majd egy `c` pontra ugrunk a c^2+c majd ez az egész a négyzeten $+c$ és így tovább. Ha sikeres volt minden művelet, akkor kiíratunk a standard kimenetre egy "mentve" üzenetet, melyből a felhasználó tudja is hogy a kép sikeresen el lett készítve, lehet megtekinteni.

Fordítás: `g++ mandelbrotcomplex.cpp -lpng -O3 -o mandelbrotcomplex`

Futtatás: `./mandelbrotcomplex mandelcomplex.png 1920 1080 1020 0.412765541820958925534057470940750.4127655418245818053080142817634623497725 0.2135387051768746491386963270997512154281 0.21353`

Kép megnyitása terminálból: `eog mandelcomplex.png`



5.2. ábra. A program áltak megcsinált kép

5.3. Biomorfok

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

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

A bimorfos algoritmus pontos megismeréséhez ezt a cikket javasoljuk: https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9_Iss5_2305-2315_Biomorphs_via_modified_iterations.pdf. Az is jó gyakorlat, ha magából ebből a cikkből from scratch kódoljuk be a sajátunkat, de mi a királyi úton járva a korábbi **Mandelbrot halmazt** kiszámoló forrásunkat módosítjuk. Viszont a program változóinak elnevezését összhangba hozzuk a közlemény jelöléseivel:

A cikkben arról olvashatunk, hogy Pickover amikor felfedezte a biomorfokat, teljesen meg volt győződve arról hogy felfedezte a természet törvényeit, tehát hogy hogyan néznek ki és alakulnak ki az élő orgazmusok. A cikkben látjuk azt is, hogy a különböző képeket különböző függvények segítségével hozunk létre, hasonlóan a Mandelbrot halmazhoz, elindul a rácson és azon végzi el a képhez tartozó függvényt. A program a komplex számsíkon dolgozik, tehát van "i" számunk, és a c egy állandó .Próbáljuk ki és nézzük meg hogy hogy is működik a program.

A forráskód megtalálható a következő linken is: [./Forraskodok/Mandelbrot/5.1/biomorfok.cpp](#)

```
// Verzio: 3.1.3.cpp
// Forditas:
// g++ 3.1.3.cpp -lpng -O3 -o 3.1.3
// Futtatas:
// ./3.1.3 bmorf.png 800 800 10 -2 2 -2 2 .285 0 10
// Nyomtatas:
// a2ps 3.1.3.cpp -o 3.1.3.cpp.pdf -l --line-numbers=1 --left-footer="←
// BATF41 HAXOR STR34M" --right-footer="https://bhaxor.blog.hu/" --pro= ←
// color
```

```
//  
// BHAX Biomorphs  
// Copyright (C) 2019  
// Norbert Batfai, batfai.norbert@inf.unideb.hu  
  
//  
// 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.  
  
//  
// This program is distributed in the hope that it will be useful,  
// but WITHOUT ANY WARRANTY; without even the implied warranty of  
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
// GNU General Public License for more details.  
  
//  
// You should have received a copy of the GNU General Public License  
// along with this program. If not, see <https://www.gnu.org/licenses/>.  
  
//  
// Version history  
//  
// https://youtu.be/IJMbqRzY76E  
// See also https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9\_Iss5\_2305--2315\_Biomorphs\_via\_modified\_iterations.pdf  
  
#include <iostream>  
#include "png++/png.hpp"  
#include <complex>  
  
int  
main ( int argc, char *argv[] )  
{  
  
    int szelesseg = 1920;  
    int magassag = 1080;  
    int iteraciosHatar = 255;  
    double xmin = -1.9;  
    double xmax = 0.7;  
    double ymin = -1.3;  
    double ymax = 1.3;  
    double reC = .285, imC = 0;  
    double R = 10.0;  
  
    if ( argc == 12 )  
    {  
        szelesseg = atoi ( argv[2] );  
        magassag = atoi ( argv[3] );  
        iteraciosHatar = atoi ( argv[4] );  
        xmin = atof ( argv[5] );  
        xmax = atof ( argv[6] );  
    }  
}
```

```
    ymin = atof ( argv[7] );
    ymax = atof ( argv[8] );
    reC = atof ( argv[9] );
    imC = atof ( argv[10] );
    R = atof ( argv[11] );

}

else
{
    std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c ←
        d reC imC R" << std::endl;
    return -1;
}
```

Header fájlként megadjuk a képkészítéshez a png++/png.hpp fejléket, majd ami új az pedig a complex header fájl ami természetesen a komplex számokkal való számolás miatt kell. Ezek után már kezdjük is a főprogramot, melyben legelőször deklaráljuk a változókat, majd meghatározzuk a magasságot, szálességet és a rácsban való mozgáshoz szükségez koordinátákat. Ez a kép kirajzoltatásához a c állandó értékében most nincs "i", ezért a c -nek a valós részéhez (reC) adunk meg 0-tól különböző értéket, az imaginárius, képzett részéhez (imC) pedig 0-t.

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

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

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

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

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

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

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

            int iteracio = 0;
            for (int i=0; i < iteraciosHatar; ++i)
            {

                z_n = std::pow(z_n, 3) + cc;
                //z_n = std::pow(z_n, 2) + std::sin(z_n) + cc;
                if (std::real ( z_n ) > R || std::imag ( z_n ) > R)
                {
```

```
        iteracio = i;
        break;
    }

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

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

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

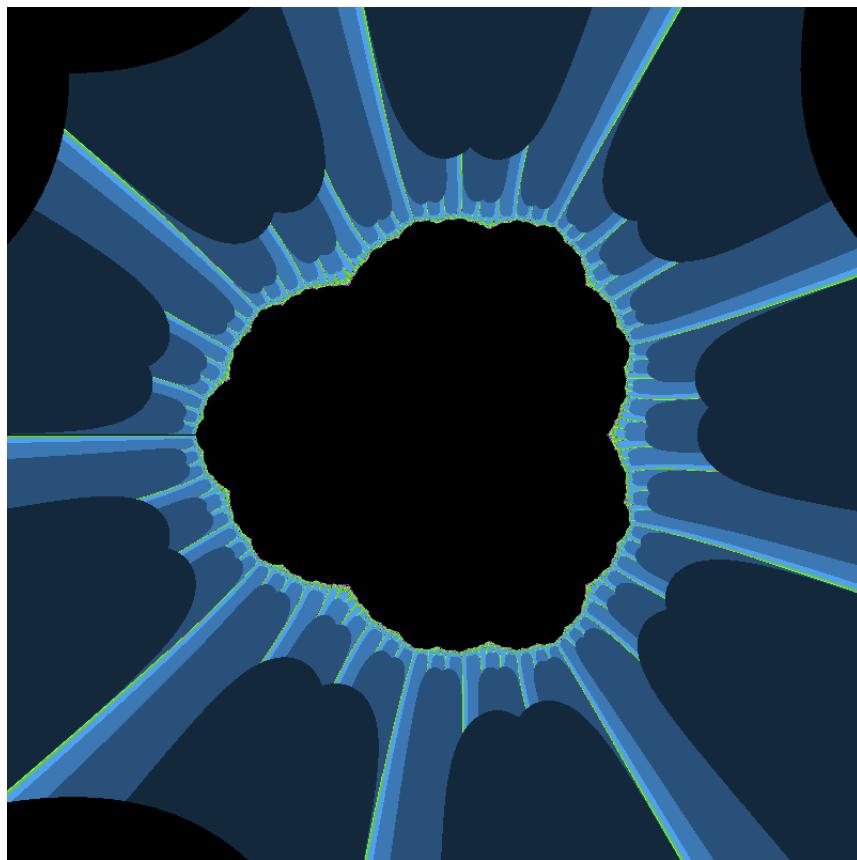
}
```

A következő programrészben, az első pixelre állunk, majd elkezdjük a számításokat. A két egymásbaágyszó for ciklussal rácsot teszünk a komplex számsíkra (ahol a j-vel a soron, a k-al pedig az oszlopon megyünk végig). Az előző complex osztállyal megvalósított Mandelbrot halmazzal szemben, itt a cc nem változik, hanem minden vizsgált z rácspontra ugyanaz, állandó lesz (a program eme változata a Júlia halmaz része). A mi programunkban a függvény a következőképpen néz ki: $z_n^3 + c$. A függvény változtatásával a készített kép is fog változni.

Fordítás: **g++ biomorfok.cpp -lpng -O3 -o biomorfok**

Futtatás: **./biomorfok biomorf.png 800 800 10 -2 2 -2 2 .285 0 10**

Kép megnyitása terminálból: **eog biomorf.png**



5.3. ábra. Biomorf kép

5.4. A Mandelbrot halmaz CUDA megvalósítása

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

Megoldás forrása: bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu nevű állománya.

A feladatban tutorom volt Racs Tamás.

A forráskód megtalálható a következő linken is: [./Forraskodok/Mandelbrot/5.4/mandelpngc_60x60_100.cu](#)

```
// mandelpngc_60x60_100.cu
// Copyright (C) 2019
// Norbert Bátfai, batfai.norbert@inf.unideb.hu
//
// 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.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
```

```
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <https://www.gnu.org/licenses/>.
//
// Version history
//
// Mandelbrot png
// Programozó Páternoszter/PARP
// https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063 ↫
// _01_parhuzamos_prog_linux
//
// https://youtu.be/gvaqijHlRUs
//

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

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

#define MERET 600
#define ITER_HAT 32000

__device__ int
mandel (int k, int j)
{
    // számítás adatai
    float a = -2.0, b = .7, c = -1.35, d = 1.35;
    int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;

    // a számítás
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, reZ, imZ, ujreZ, ujimZ;
    // Hány iterációt csináltunk?
    int iteracio = 0;

    // c = (reC, imC) a rács csomópontjainak
    // megfelelő komplex szám
    reC = a + k * dx;
    imC = d - j * dy;
    // z_0 = 0 = (reZ, imZ)
    reZ = 0.0;
    imZ = 0.0;
    iteracio = 0;

    while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
    {
```

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

++iteracio;

}

return iteracio;
}

__global__ void
mandelkernel (int *kepadat)
{

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

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

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

void
cudamandel (int kepadat[MERET][MERET])
{
    int *device_kepadat;
    cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));

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

    cudaMemcpy (kepadat, device_kepadat,
               MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost);
    cudaFree (device_kepadat);
}

int
main (int argc, char *argv[])
{
    // Mérünk időt (PP 64)
    clock_t delta = clock ();
    // Mérünk időt (PP 66)
    struct tms tmsbuf1, tmsbuf2;
    times (&tmsbuf1);

    if (argc != 2)
```

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

Ugynúgy mint a legelső Mandelbrotos feladat, a mandelbrot halmazt számolja ki, hasonló műveletekkel, viszont van egy nagy különbség a kettő között: míg az első a CPU-t használja a számítások közben, addig ez a program a GPU-t használja. A program futtatása után a számok melyek az időt mutatják magukért beszélnek. Míg a CPU-t használó program 15 másodpercet vett igénybe, addig CUDA-s program mintegy 0.15 másodperc alatt kiszámolja nekünk ugyanazzt az eredményt. A két program leírásban nagyon hasonlít az előző feladathoz, mint mondtam itt a számítási idők mások, ezt az időt a program során számoljuk és iratjuk ki, hogy legyen ű vizsgálati alapunk. Mivel nekem nincs CUDA kártyám ezért nem tudom élezni ezt a gyors programfutás által gerjeszett élményeket, de a fent belinkelt videón minden megtapasztalható.

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

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

Megoldás forrása:

A forráskód megtalálható a következő linken is: [..../Forraskodok/Mandelbrot/5.5/mandelbrotNagyito.cpp](#)

A Mandelbrot halmaz kiszámítása mellett ezzel a programmal a kiadott képen nagyítást is tudunk végrehak-tani. A kép megnyitása után az egerünk görgőjének segítségével tudunk benne nagyítani, illetve touchpad-en a kétújas mozdulattal.

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

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

    if ( argc == 9 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
        a = atof ( argv[5] );
        b = atof ( argv[6] );
        c = atof ( argv[7] );
        d = atof ( argv[8] );
    }
    else
    {
        std::cout << "Hasznalat: ./3.1.1 fajlnev szelesseg magassag n a b c d" <<
            std::endl;
        std::cout << "Most az alapbeallitasokkal futtatjuk " << szelesseg << "
        "
        << magassag << " "
        << iteraciosHatar << " "
        << a << " "
        << b << " "
        << c << " "
        << d << " " << std::endl;
```

```
//return -1;  
}
```

A program eleje ismerős hiszen úgyanaz mint az előző programok esetében: header fájlok dekalrálása, majd a változók beadagolása (szelesseg, magassag, iteraciosHatar, a, b, c és d). Most is megvizsgáljuk hogy a parancs amivel a felhasználó futtatta a programot, az tényleg 9 argumentumból áll-e. Ha igen akkor az az egyes argumentumokat, értékeket átadjuk az egyes változóknak, ha viszont az argumentumok száma nem felel meg, akkor használati utasítást adunk a programot futtatónak hogy mi lenne a helyes mód, illetve még segítségképp megmutatjuk azt is hogy mik voltak az általunk adott alap beállítások.

```
png::image < png::rgb_pixel > kep ( szelesseg, magassag );  
  
double dx = ( b - a ) / szelesseg;  
double dy = ( d - c ) / magassag;  
double reC, imC, reZ, imZ;  
int iteracio = 0;  
  
std::cout << "Szamitas\n";  
  
for ( int j = 0; j < magassag; ++j )  
{  
    for ( int k = 0; k < szelesseg; ++k )  
    {  
        reC = a + k * dx;  
        imC = d - j * dy;  
        std::complex<double> c ( reC, imC );  
  
        std::complex<double> z_n ( 0, 0 );  
        iteracio = 0;  
  
        while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )  
        {  
            z_n = z_n * z_n + c;  
  
            ++iteracio;  
        }  
  
        iteracio %= 256;  
  
        kep.set_pixel ( k, j,  
                        png::rgb_pixel ( iteracio%255, 0, 0 ) );  
    }  
  
    int szazalek = ( double ) j / ( double ) magassag * 100.0;  
    std::cout << "\r" << szazalek << "%" << std::flush;  
}  
  
kep.write ( argv[1] );  
std::cout << "\r" << argv[1] << " mentve." << std::endl;
```

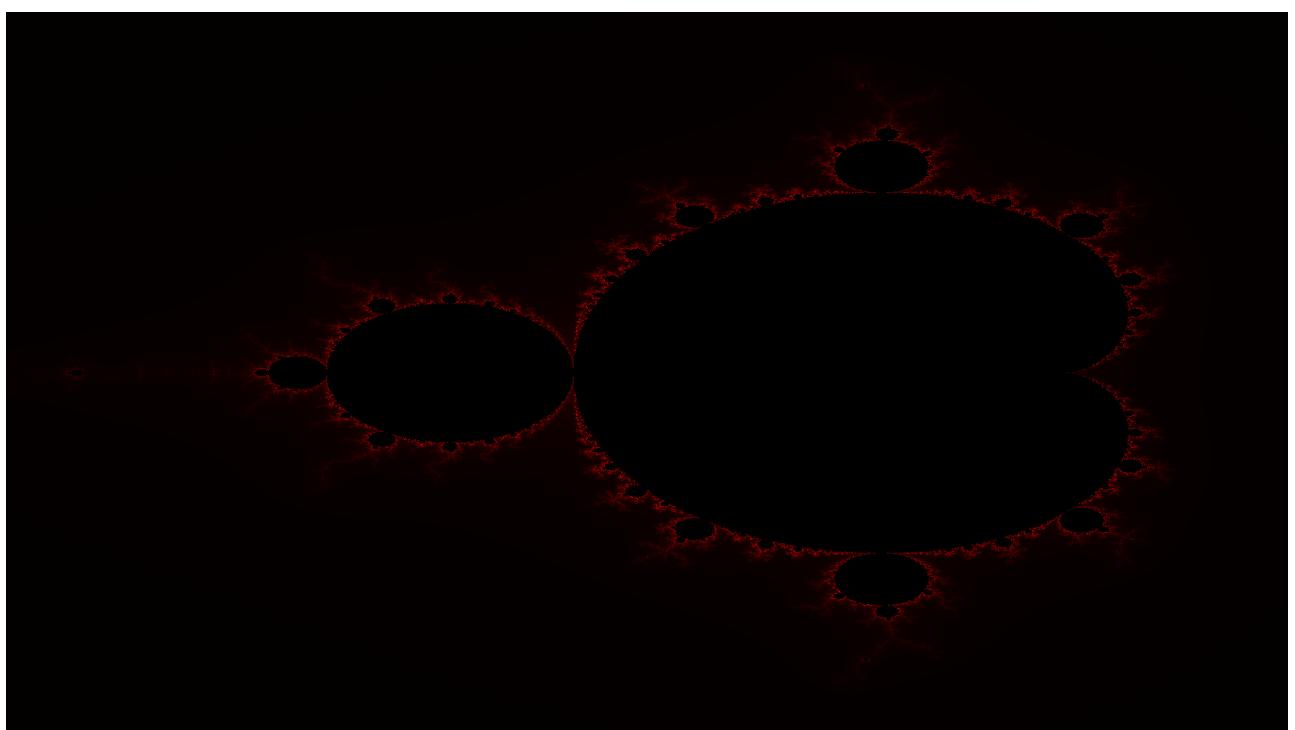
```
}
```

A program utolsó része ugynőgy komplexszámos megoldással van csinálva. A két for ciklussal végigmegyünk a rácson j-vel a soron, k-val az oszlopon. A a reC, imC, reZ és imZ (valós és imaginárius részű) változókat használva végigmegyünk minden rácsponton úgy, hogy a c=(reC, imC) a haló rácspontjainak megfelelő komplex szám. A megadott képletet használva megalkotjuk a várt képet, megkapjuk a "mentve" üzenetet, megnyitjuk a képet és tetszésünk szerint nagyítgathatunk a kapott képben aszerint ahogy a feladat legején bemutattam.

Fordítás: **g++ mandelbrotnagyito.cpp -lpng16 -O3 -o mandelbrotnagyito**

Futtatás: **./mandelbrotnagyito mandelnagytocpp.png 1920 1080 1020 0.4127655418209589255340574709407 0.2135387051768746491386963270997512154281 0.2135387051804975289126531379224616102874**

Kép megnyitása terminálból: **eog mandelnagytocpp.png**



5.4. ábra. C++ mandelbrot nagyító és utazó

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás videó: <https://youtu.be/Ui3B6IJnssY>, 4:27-től. Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a_mandelbrot_nagyito_uttazoval/

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#id570518>

A program célja ugyanaz mint az előző programé: nagyítsunk a kapott képben. A java-s program úgy van megírva, hogy itt a bal egérgomb segítségével kell kijelöljük azt a részt amit pontosabban, "közelebbről"

megszeretnénk tekinteni. Ezt a műveletet nagyon sokszor tudjuk megcsinálni egymás után. A programban commentként rengeteg magyarázatot találunk, melyek a különböző sorok, parancsok működését írják le.

A forráskód megtalálható a következő linken is: [./Forraskodok/Mandelbrot/5.6/MandelbrotHalmazNagyito.java](#)

```
/*
 * MandelbrotHalmazNagyító.java
 *
 * DIGIT 2005, Javat tanítok
 * Bátfai Norbert, nbatfai@inf.unideb.hu
 *
 */
/**
 * A Mandelbrot halmazt nagyító és kirajzoló osztály.
 *
 * @author Bátfai Norbert, nbatfai@inf.unideb.hu
 * @version 0.0.1
 */
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {
    /** A nagyítandó kijelölt területet bal felső sarka. */
    private int x, y;
    /** A nagyítandó kijelölt terület szélessége és magassága. */
    private int mx, my;
    /**
     * Létrehoz egy a Mandelbrot halmazt a komplex sík
     * [a,b]x[c,d] tartománya felett kiszámoló és nyígtani tudó
     * <code>MandelbrotHalmazNagyító</code> objektumot.
     *
     * @param a           a [a,b]x[c,d] tartomány a koordinátája.
     * @param b           a [a,b]x[c,d] tartomány b koordinátája.
     * @param c           a [a,b]x[c,d] tartomány c koordinátája.
     * @param d           a [a,b]x[c,d] tartomány d koordinátája.
     * @param szélesség   a halmazt tartalmazó tömb szélessége.
     * @param iterációsHatár a számítás pontossága.
     */
    public MandelbrotHalmazNagyító(double a, double b, double c, double d,
                                    int szélesség, int iterációsHatár) {
        super(a, b, c, d, szélesség, iterációsHatár);
        setTitle("A Mandelbrot halmaz nagyításai");
        addMouseListener(new java.awt.event.MouseAdapter() {
            public void mousePressed(java.awt.event.MouseEvent m) {
                x = m.getX();
                y = m.getY();
                mx = 0;
                my = 0;
                repaint();
            }
            public void mouseReleased(java.awt.event.MouseEvent m) {
                double dx = (MandelbrotHalmazNagyító.this.b
                             - MandelbrotHalmazNagyító.this.a)
                            /MandelbrotHalmazNagyító.this.szélesség;
            }
        });
    }
}
```

```
        double dy = (MandelbrotHalmazNagyító.this.d
                      - MandelbrotHalmazNagyító.this.c)
                      /MandelbrotHalmazNagyító.this.magasság;
        new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+ ←
            x*dx,
            MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
            MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
            MandelbrotHalmazNagyító.this.d-y*dy,
            600,
            MandelbrotHalmazNagyító.this.iterációsHatár);
    }
});
```

A program elején úgyanúgy mint ahogy megszoktuk jönnek a változók deklarálásai, ahol az `x` és az `y` változókkal határoljuk be a nagyítandó terület bal felső sarkát, illetve az `mx` és `my` pedig a kijelölt nagyítandó terület magassága és szélessége. Víjuk az ōs osztály konstruktőrét, beállítjuk az ablak címét (amiben majd nagyítunk) és deklaráljuk az egér kattintás műveletét. Az azonnal következő `mousePressed` függvény segítségével deklaráljuk a nagyítandó terület bal felső sarkát, szélességé és magasságát. A következő `mouseReleased` függvénytellyel az a művelet aktivizálódik, amikor kattintjuk az egert, vonszolással kijelöljük a nagyítani kívánt területet és amikor elengedjük az egérgombot akkor a kijelölt terület újraszámítása kezdődik el, illetve megalkotódik a kinagyított terület, az lesz az aktuális kép amit az ablakban látunk és továbbnagyítunk majd.

```
addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
    public void mouseDragged(java.awt.event.MouseEvent m) {
        mx = m.getX() - x;
        my = m.getY() - y;
        repaint();
    }
});
```

Itt deklaráljuk a fentebb már használt függvényt, ami arra szolgál hogy követni lehessen a eglr mozgását, kattintásának feldolgozását, a kijelölés menetét mindenkorral hogy közben megadjuk a `mouseDragged` függvényt (ami nem más mint a vonszolás, a kijelölés menete, az új szélesség és a magasság bellítása és az újraméretezés által).

```
public void pillanatfelvétel() {
    java.awt.image.BufferedImage mentKép =
        new java.awt.image.BufferedImage(szélesség, magasság,
            java.awt.image.BufferedImage.TYPE_INT_RGB);
    java.awt.Graphics g = mentKép.getGraphics();
    g.drawImage(kép, 0, 0, this);
    g.setColor(java.awt.Color.BLUE);
    g.drawString("a=" + a, 10, 15);
    g.drawString("b=" + b, 10, 30);
    g.drawString("c=" + c, 10, 45);
    g.drawString("d=" + d, 10, 60);
    g.drawString("n=" + iterációsHatár, 10, 75);
    if(számításFut) {
```

```
        g.setColor(java.awt.Color.RED);
        g.drawLine(0, sor, getWidth(), sor);
    }
    g.setColor(java.awt.Color.GREEN);
    g.drawRect(x, y, mx, my);
    g.dispose();
    StringBuffer sb = new StringBuffer();
    sb = sb.delete(0, sb.length());
    sb.append("MandelbrotHalmazNagyitas_");
    sb.append(++pillanatfelvételszámláló);
    sb.append("_");
    sb.append(a);
    sb.append("_");
    sb.append(b);
    sb.append("_");
    sb.append(c);
    sb.append("_");
    sb.append(d);
    sb.append(".png");
    // png formátumú képet mentünk
    try {
        javax.imageio.ImageIO.write(mentKép, "png",
            new java.io.File(sb.toString()));
    } catch(java.io.IOException e) {
        e.printStackTrace();
    }
}
```

Mint a sima nagyítás nélküli programban, itt is lehetővé tesszük a felhasználó számára a pillanatfelvétel készítésének a lehetőségét. Ebben az esetben figyelme vesszük azt is, hogy az adott képernyőfotó hányadik volt, mivel hogy könnyebben tudjunk majd eligazodni a képek között, a kép nevében szerepelni fog hogy mikor csináltuk azt, hányadikként. A fájl nevébe beleveszünk, hogy melyik tartományban találtuk a halmazt:

```
public void paint(java.awt.Graphics g) {
    g.drawImage(kép, 0, 0, this);
    if(számításFut) {
        g.setColor(java.awt.Color.RED);
        g.drawLine(0, sor, getWidth(), sor);
    }
    g.setColor(java.awt.Color.GREEN);
    g.drawRect(x, y, mx, my);
}

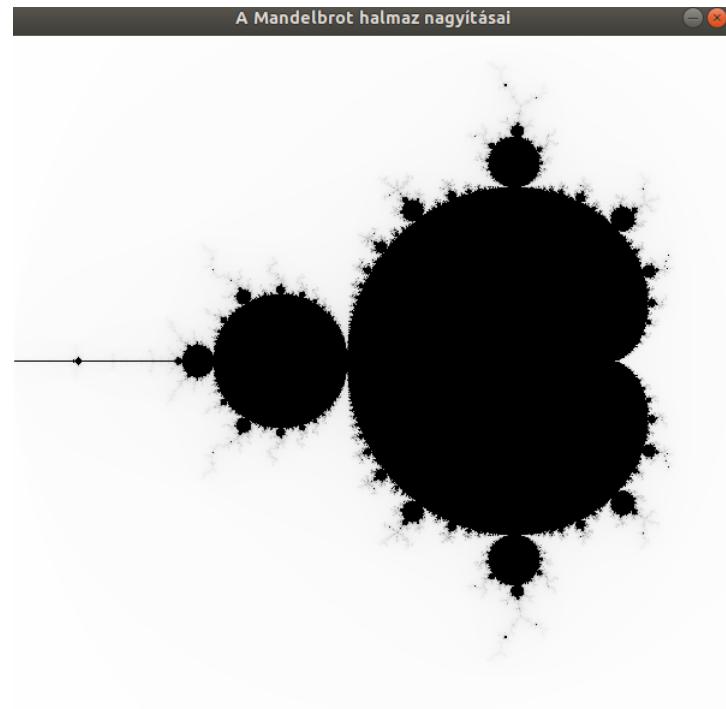
public static void main(String[] args) {
    new MandelbrotHalmazNagyító(-2.0, .7, -1.35, 1.35, 600, 255);
}
```

A paint segítségével legelőször is kirajzoljuk a Mandelbrot halmazt. Ha a számítások éppen futnak, akkor azt hogy melyik sorban tart egy vörös csíkkal jelöljük. Végül pedig kirajzoltatjuk a nagyítandó

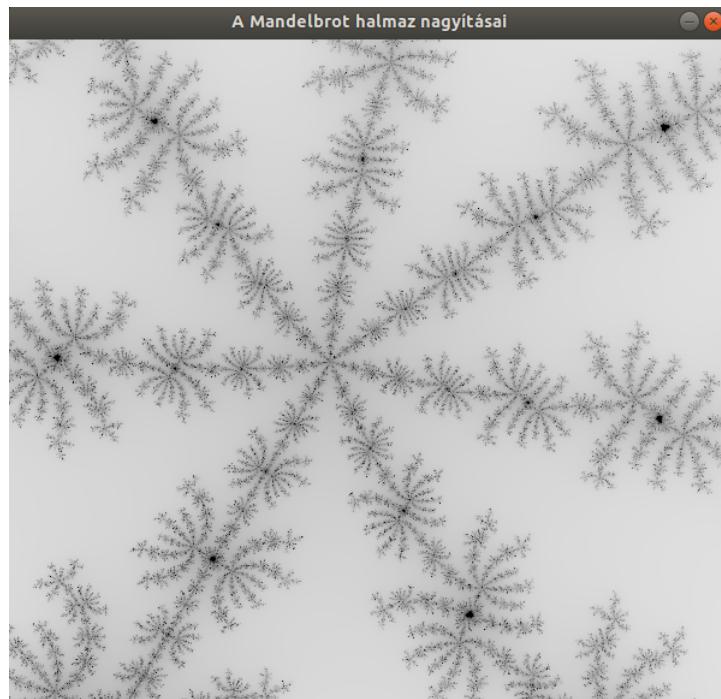
terület körvonalát zöld színnel.

Fordítás: **javac MandelbrotHalmazNagyító.java**

Futtatás: **java MandelbrotHalmazNagyító**



5.5. ábra. Java mandelbrot nagyító és utazó



5.6. ábra. Többszöri nagyítás után

5.7. Vörös Pipacs Pokol/fel a láváig és vissza

Írj olyan Minecraft MALMÖ python programot, ami felküldi a láváig Steve-et, majd visszafordítja és leküldi onnan.

Megoldás videó: [Megoldás videó](#)

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Forraskodok/Mandelbrot/lava_fel_le.py, <https://github.com/nbatfai/RedFlowerHell>

Mivel már Steve képes érzékelni a környezetében lévő objektumokat, nincs nehéz dolgunk. Elindítjuk a felfelé, majd amint érzékeli maga előtt a folyó lávát, kétszer 90 fokot fordul és elindul lefelé, a pálya közepére. Ekkor a program megáll.

```
class Steve:
    def __init__(self, agent_host):
        self.agent_host = agent_host

        self.nof_red_flower = 0

    def run(self):
        world_state = self.agent_host.getWorldState()
        self.agent_host.sendCommand( "look 1" )
        while world_state.is_mission_running:
            for i in range(100):
                self.agent_host.sendCommand( "move 1" )
                time.sleep(.5)
```

```
    self.agent_host.sendCommand( "jumpmove 1" )
    time.sleep(.5)
    world_state = self.agent_host.getWorldState()
    i=i+1

    if world_state.number_of_observations_since_last_state != 0:
        sensations = world_state.observations[-1].text
        #print("    sensations: ", sensations)
        observations = json.loads(sensations)
        nbr3x3x3 = observations.get("nbr3x3", 0)
        print("    3x3x3 neighborhood of Steve: ", nbr3x3x3)

        if "Yaw" in observations:
            self.yaw = int(observations["Yaw"])
        if "Pitch" in observations:
            self.pitch = int(observations["Pitch"])
        if "XPos" in observations:
            self.x = int(observations["XPos"])
        if "ZPos" in observations:
            self.z = int(observations["ZPos"])
        if "YPos" in observations:
            self.y = int(observations["YPos"])

        #print("    Steve's Coords: ", self.x, self.y, self.z)
        #print("    Steve's Yaw: ", self.yaw)
        #print("    Steve's Pitch: ", self.pitch)

        if nbr3x3x3[18] == "flowing_lava":
            self.agent_host.sendCommand( "turn 1" )
            time.sleep(.2)
            self.agent_host.sendCommand( "turn 1" )
            time.sleep(.2)
            for i in range(100):
                self.agent_host.sendCommand( "move 1" )
                time.sleep(.02)
            quit()

    world_state = self.agent_host.getWorldState()
```

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás videó:

Megoldás forrása:

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

6.2. LZW

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

Megoldás videó:

Megoldás forrása:

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása:

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása:

6.5. Mutató a gyökér

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

Megoldás videó:

Megoldás forrása:

6.6. Mozgató szemantika

Írj az előző programhoz mozgató konstruktort és értékkadást, a mozgató konstruktor legyen a mozgató értékkadásra alapozva!

Megoldás videó:

Megoldás forrása:

7. fejezet

Helló, Conway!

7.1. Hangyszimulációk

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

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

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

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

Tanulságok, tapasztalatok, magyarázat...

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Minecraft-MALMÖ

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

Tanulságok, tapasztalatok, magyarázat...

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Tanulságok, tapasztalatok, magyarázat...

10. fejezet

Helló, Gutenberg!

10.1. Magas szintű programozási nyelvek 1

A programozási nyelveket sok tulajdonságuk alapján tudjuk besorolni őket különböző osztályokba, de ténylegesen 3 különböző szintet különítünk el amikor beakarunk sorolni egy nyelvet. Ezek a gépi, assembly szintű és magas szintű nyelvek, utóbbival foglalkozunk jelenleg (2018/2019/2 félév). A magas szintű nyelvvel elkészített programokat forrásprogramnak, illetve forrásszövegnek(source code) nevezzük. Ezek megírásának szabályait szintaktikai szabályoknak nevezzük, értelmezési szabályokból épülnek fel a szemantikai szabályok.

Tudjuk számítástechnikai ismereteinkből, hogy minden egyes feldolgozó egység csak az ő saját nyelvén, saját dialektusán megírt programokat tudja végrehajtani, ebből adódik az, hogy a magas prog. nyelvek forráskódját át kell alakítani valahogyan, például interpreteres(értelmező) technikával vagy fordítóprogramos technikával. A fordító(program) tárgykódot állít elő, ami gépi kód.

A fordító működése során a következő lépéseket hajtja végre, jó esetben:

- lexikálás elemzés
- szintaktikai elemzés
- szemantikai elemzés
- kódgenerálás

Lexikális egységre bontás után ellenőrizzük, hogy szintaktikailag helyes-e a fkód mivel csak azokat tudjuk lefordítani. A futtatható programot a szerkesztő vagy kapcsolatszerkesztő (linker) állít elő. A betöltő viszi a tárba, válik processzussá a program, és a felügyelést a futtató rendszer felügyeli (pl Out Of Bound kivétel dobása).

Nyelvről nyelvre fordítás történik, léteznek olyan nyelvek (pl C) ami egy előfordító segítségével először a forrásprogramból egy adott nyelvű programot készít ami majd feldolgoztatja magával a nyelv fordítójá.

Minden nyelvnek van szabványa, ami az idő során fejlődhet/módosulhat. Ezek leírják, hogy hogyan lehet használni az adott nyelvet (C89 C99). Sokféle fordítót használhatunk pl GNU Compiler Collection fordítás során, és sokféle forráskód szerkesztőt, szövegszerkesztőt is például Kate, vagy integrált fejlesztési környezetet pl Visual Studio, Code::Blocks.

Az interpreteres tech. nem készít tárgykódot, hanem értelmezi és végrehajtja a kódost sorrol sorra.

Imperatív nyelvek A hacker algoritmusokat ír és ez működteti majd a processzort. A programja utasítások sorozata, változókat használhat(tárt érheti el vele) Alcsoportjai eljárásorientált/objektumorientált nyelvek.

Deklaratív nyelvek Ezek nem algoritmikus p. nyelvek, a problémát írjuk le velük aminek megoldási lépései a nyelvi implementációban van beépítve. Nincsenek memóraműveletek elérhetőek a programozók számára, vagy csak nagyon korlátozottan. Funkcionális és logikai nyelvek alcsoportjai. Ezenkívül lehet még többbe is sorolni őket, a nyelveket.

Forrásszövegek legkisebb részei a karakterek, tehát mindegyiknek van karakterkészlete. Eljárásorientált nyelvek nyelvi elemei: lexikális/szintaktikai/program/fordítási egységek, utasítások, a program. Általában az angol ABC betűit használják a programnyelvek(például van orosz is), és az arab számokat használják.

A lexikális egységeket ismeri fel a fordító. Az operátorok(például ++ C/C++/.. stb) ilyen egységek, de ilyenek például a változók azonosítója (m_e) ezek nyílván korlátozottak az ABC-re és lehet hosszhatáruk is. A nyelv fenntart kulcsszavak is, például if elágazás C-ben, ezek nem definiálhatók újra a user által. Megjegyzéseket is támogathat egy nyelv, ezeket a fordító ignorálja, segítik a user áltla írt kód megértését. A konstansok nem módosíthatóak(értékük).

A sorokban az utasítások nyelvtől függően tartalmazhat csak 1 vagy több utasítást, utasításokat ált. a ';' választja el de lehet akár '\n' is..

Egy adattípus 3 dolog ad meg(tartomány, műveletek, reprezentáció). A tartományuk azokat az elemeket tartalmazzák, melyet a megadott típusú konkrét programozási eszköz fölvetet értékként, akár ezek literálok is lehetnek. A műveletek ezeken az elemek használjuk, belső ábrázolási módot az implementáció határozza meg(például szértszórt, folytonos ábrázolás.) Saját típushoz is lehet definiálni egyes nyelvekben például C.

A nevesített kontansoknak 3 komponense van: név, típus, érték, minden deklarálni kell! Konstans, tehát értéke nem változtatható meg, sem helye az operatív tárban. Például az úgynevezett magic numbers helyett használjuk. Létrehozásuk például C-ben a #define preprocessor direktívával vagy const.

Változók(komponensek: név, attribútumok, cím, érték). Név azonosító, attribútumok a futás közben viselkedésüket befolyásolja(típus). Explicit/implicit vagy automatikus módon deklarálhatjuk őket(változókat). Automatikus esetnél a fordítóprogram rendel attribútumot azokhoz a változókhöz, amelyknél nincsen explicit módon ez deklarálva, másik két esetben a programozó feladata. A címkomponens a tárbeli helyét tartalmazza az értéknek, a futási idő azon részét, amikor rendelkezik ezzel a változó élettartamának nevezük. A tárkiosztás lehet statikus/dinamikus, vagy elől futás előtt a címe vagy csak futás alatt, de a programozó is rendelhet futási idő alatt(abszolút/relatív címzés, vagy megadja mikor legyen cím hozzárendelve a rendszer által)

A C-nek vannak aritmetikai(egyszerű) és származtott(összetett) típusai, illetve void. Aritmetikai lehet integrális(egész,karakter,felsorolásos) és valós(float, double). A származtatott: tömb, függvény, mutató, struktúra, union. A C csak egydimenziós tömböket kezel(darabszám megadás szükséges), és azt minden mutató típusként, van auto dek. alapértelmezés az int.

Két résziük van a kifejezéseknek: érték és típus. Operandusok, operátorok, kerek zárójelek az összetevői. Legegyszerű kif. csak egyetlen operandusból áll. Léteznek unáris, bináris, ternáris operátorok, tehát, hogy hány operanduson dolgoznak. Alakjuk lehet prefix(+ a b), infix(a + b), postfix(a b +). A végrehajtási sorrend lehet balról-jobbra, fordítva, vagy precedencia tábla szerint ballról-jobbra. Az operandusok értékének meghatározásának sorrendje vagy szabályozza a nyelv vagy nem pl a C nem. Zárójelezés felülírhatja a precedencia táblázat szabályainak alkalmazását, ugye minden azt kell kiértékelni először. Teljesen zárójelezett infix egyértelmű. Logikai kifejezéseket kilehet értekelni rövidzár módszerrel(konjunkció első tagja

hamis, akkor már nem kell a másik részt vizsgálni) teljes kiértékeléssel. A kif. típusának meghatározásánál kétféle elv: típusegyenértékűség vagy típuskényszerítés. Első esetben egy kétoperandusú operátornak csak azonos típusú operandusai lehetnek, nincs castolás ekkor, vagy eldöntheti az operátor is(==). Második esetnél lehetnek különbözők, de mivel csak azonos ábrázolású operanduszok között végezhetőek műveletek konverzió van. A nyelv definiálja a típust ekkor. C enged például valóst egésszé vagy fordítva kasztolni. A C eleve a típuskényszerítés elvét vallja. A muatókon értelmezett a kivonás és összeadás, egésznek tekinthetők(unsigned int).

Utasítások alkotják az algoritmus lépésein illetve ez alapján készül a tárgykód. Két csoportja van a deklarációs és a végrehajtható utasítások. Utóbbi csak a fordítóprogramnak kotypog, mindenféle igényt kérnek, mint például üzemmód beállítása. A hacker csak a névvel rendelkező ő általa birtokolt prog. eszközeit tudja deklarálni.

Megkülönböztetünk: értékadó, üres, ugró, elágaztató, ciklusszervező, hívó, vezérlésátadó, I/O és egyéb utasításokat.

Értékadó utasítás feladata beállítani, frissíteni egy változó értékkomponensét akármikor futási idő alatt. AZ üres utasítás hatására a processzor üres gépi utasítást hajt végre. A ugró utasítás egy megcímkezett utasításra adhatjuk a vezérlést. NEM BIZTONSÁGOS, ÁTLÁTHATATLAN KÓDOT VONHAT MAGA UTÁN!!

Az elágaztató utasítás révén egy adott ponton két aktivitás közül választhatunk végrehajtani. Ált. felépítése if feltétel then tevékenység else tevékenység. A feltétel logikai kifejezés, a tevékenység utasításainak száma függ a nyelvtől, lehet egyetlen egy vagy egy blokknyi utasítás. Az IF-utasítások tetszőlegesen egymásba vihetők. Felmerülhet a csellengő ELSE probléma ekkor, azaz 'if then if then else' esetén kihez tartozik az ELSE? Válasz soféle lehet: Elkerülhető a probléma, ha minden hosszú IF-utasítást írunk, vagy implementáció függő, vagy a szintaktika alapján egyértelmű.

Többirányú elágaztató utasítás: amikor olyan ponton vagyunk ahol diszjunkt tevékenységek közül egyet kell végrehajtanunk kifejezés alapján. C-en ez a switch statement

```
switch (kifejezés)
{
    case egész_konstans_kif : [tevékenység]
        ...
    default: tevékenység
}
```

A kifejezésnek egészre konvertálhatónak kell lennie! Case ágak kifejezései DISZJUNKTAKNAK KELL LENNIÜK!! Default bárhol szerepelhet. Kiértékelődik a kifejezés majd sorrendre összehasonlításra kerül a case ágak értékeivel, van egyezés végrehajtódik, ha nem akkor a default fog. Ha nincs default akkor egy üres utasítás fog. Akárhogy is, a case és default ágakban szereplenie kell a break utasításnak, azzal lépünk ki a switchből.

A ciklusszervező utasítások lehetővé teszik, hogy a program egy adott pontján egy bizonyos aktivitást többször is megismételjünk. Egy ciklus ált. felépítése: fej mag vég. Fejben vannak az ismétlésre vonatkozó információk vagy a végben. Mag tartalmazza az ismételni kívánt utasítások halmazát. Két radikális típusa van a ciklusoknak végtelen és üres, előbbi soha nem áll le, másik egyszer sem fog végrehajtódni. Ciklusfajták: feltételes, előírt lépésszámú, felsorolásos, végtelen és összetett.

A feltételesnél az ismétlődést nyílván egy igaz/hamis érték fogja meghatározni. Van kezdőfeltétel és vég-feltételes feltételes ciklusok. Rendre: a feltétel a fejben, a feltétel a végben(ált.). Első verzió kiértéklődik először, ha igaz végrehajtódik, és ahányszor/ameddig igaz lesz addig végre is fog hajtódni a ciklusmag. Kell lennie valahol emiatt a magban olyan utasítás ami változtat a feltétel értékét. Végtelen vagy üres lehet.

A végfeltételes ciklusok először hajtónak végre, majd kiértékelődik a feltétel és ha az igaz lesz akkor megint végrehajtódik és ezt folytatjuk amíg hamissá nem értékelődik ki. DE vannak olyan nyelvek, ahol a fordítottja megy végbe, addig hajtódik végre a mag amíg igazzá nem válik a feltétel. Soha nem lehet üres, mert egyszer lefut minden. Végtelenség engedélyezett.

AZ előírt lépésszámu ciklus azz ismétlésre vonatkozó információk a fejben vannak. minden esetben tartozik hozzá van ciklusváltozója, és erre felvett értékekre fut le a mag. Egy előírt tartományból vehet fel értékeket, ezt a fejben adjuk meg, mint kezdő és végértéket. Lépésköz ált 1, tehát minden elemet felvesz, de ez módosítható. A tartományt befuthatja növekvőleg/csökkenően.

Lehet előtesztelős vagy hátultesztelős, a működés implementáció függő. Előbbi esetében működés kezdetén definiálva lesznek a ciklusparaméterek, majd a futattó rendszer megnézi, hogy a megadott tartomány üres-e. Ha az, üres ciklus, különben lefut a mag miután a kezdőértéket felvette az ciklusváltozó. Ezután megnézi van-e még olyan érték amit felvehet, ha van újra lefut a mag, és ezek a lépések ismétlődnek. Hátultesztelős ugyanúgy meghatározódnak a paraméterek, de előbb lefut a mag, majd értékelődik ki a feltétel..

A felsorolásos ciklusnak van ciklusváltozója és annak értéke amit a fejben adunk meg, minden felvett érték mellett végrehajtódik a mag. Nem lehet üres, végtelen.

Végtelen ciklusnak sem fejben sem végben nincs információ az ismétlődésre vonatkozóan. Üres ciklus nem lehet. Használatánál a magban kell olyan utasítást alkalmazni, ami megállítja.

Az összetett ciklus 4 ciklusfajta kombinációiból tevődik össze. A fejben bármennyi ismétlésre értetendő információk sorolhatóak fel, jelentések pedig szuperponálódnak.

3 vezérő utasíása van a C-nek a fentebb említettek mellet. Ez a continue; break; és return[kifli]. A continue; ciklusmagban használatos, kihagyja az ót követő utasítások végrehajtását és megvizsgálja a ciklusfeltételt és az alapján kilép vagy újabb cikluslépés végrehajtásba kezd. A break is a magban lehet, befejezteti a cklust, kilép az elágaztató utasításokból. A return; szabványosan bejezteti a függvényt és a hívónak adja vissza a vezérlést.

A programok több egymástól valamennyire független részekből áll az eljárásorientált típusú nyelveknél, ezeket nevezzük programegységeknek. Ebből adóóan kérdések fogalmazhatóak meg velük kapcsolatban.

Ha több egység áll egy program akkor egyben kell az egészet fordítanunk vagy lehetőség van-e arra, hogy egyenként, mivel függetlenek, fordítsuk le őket?

Erre a válasz a nyelv típusától, a nyelv adta lehetőségektől függ. Egyes nyelvekben ténylegesen van lehetőség arra, hogy külön fordítsuk le őket, hiszen fizikailag önálló programokból áll maga a "főprogram". Ezek nem strukturáltak.

Nem ilyen nyelvek esetében csak arra van mód, és ez kötelező, máshogy nem lehet, hogy egyetlen egy egysékként legyen fordítva a program, strukturálható ekkor a program kód, viszont a programegységek fizikailag nem függetlenek.

A fenti kettő együttese is valós opció lehet, ekkor függetlenek az egységek, de akármilyen struktúrával bíró egységek vannak.

Emellett feltehetünk olyan kérdésket is, minthogy ha külön fordulnak az egységek, mi adja az önálló fordítási egységet, vagy milyen és milyen viszony van a programegységek között, hogyan kommunikálnak..

Az eljárásorientált nyelvekben a következő programegységekről beszélünk: alprogram, blokk, csomag, taszk.

Eljárásorientált nyelvknél az alprogram egy absztrakciós réteg. Azért mondjuk rá, hogy absztrakciós réteg vagy absztrakciós eszköz, mert formális paraméterekkel látjuk el. Ez egy általánosítást jelent, tehát absztrakció ment végbe.

A feladata az alprogramnak az, hogy a bemeneti adatoknak egy halamzát képezi le kimeneti halmazba, úgy hogy csak a leírás(specifikáció) az adatoknak csak leírását adja meg, az implementálásból de a hátérben zajló leképezésről nincs információink.

Az alprogramot, absztrakciós volta miatt, olyan helyeken használjuk a programunkban, ahol egy állandóan ismétlődő műveletet szeretnénk kiváltani vele. Azaz nem akarjuk újra meg újra leírni a műveletet, hanem egyszer megírjuk és amikor épp az a művelet szükséges hivatkozunk rá.

Az alprogram áll fejből, törzsből és végből. Első kettőt rendre szoktuk hívni specifikációinak és implementációinak is. Ez a formális felépítése.

Ha mint eszköz gondolunk rá, akkor név, formális paraméter lista, törzs és környezet komponensek építik fel.

A név, azonosító, minden a fejben megtalálható.

A paraméter lista azonosító lista, egy absztrakt/általán műveltet írnak le, a hívás helyén konkrét értéket kell megadnunk nekik.

Kezdetben a formális paraméter listán csak paraméterek nevei lehetettek, később bővült azzal, hogy olyan információkat is megadhatunk amik változtatásokat visznek végbe a paraméterek viselkedésében, futási időben. A lista lehet üres is.

A törzsben utasításokból deklarációs és végrehajtható utasítások lehetnek, néhány nyelv esetében előfordulhat az, hogy kötelező elkülöníteni a kétfajta, előbb említett, utasítást, így 2 része lesz a törzsnek. Illetve vannak nyelvek amikben szabadon keverhetőek.

A globális változók együttese az alprogram környezete. Alprogram lehet eljárás a függvény. A kettő között az a különbség, hogy az eljárás nem tér vissza visszatérési értékkel, hanem paramétereit és/vagy környezetét frissíti, a függvény pedig olyan alprogram aminek van visszatérési értéke, típusa tetszőleges de ez hozzátarozik a függvény leírásához. A függvény is befolyásolhatja paramétereit, ha ez megtörténik mellékhatalsról beszélünk.

Az eljrást bárhonnan ahol utasítás állhat meghívható. A meghívás is nyelv függő, valamelyik nyelv esetében explicit meg kell mondani, hogy meghívni akarjuk az eljárást, máshol automatikus az eljárás első sorára kerül a vezérlés. Szabályosan vagy szabálytalanul fejeződhet be egy eljárás, szabályos az amikor az eljárás végére értünk vagy egy kulcsszó használatával kilépünk. Szabálytalan az, nyelv függő ez is, amikor gotoval kiugrunk egy nem benne lévő címkehez vagy valamilyen eszköz hatására az egész program leáll és az operációs rendszeré lesz a vezérlés.

Csakis kifejezésben lehet függvényt meghívni, ha szabályosan fejeződött be akkor a kifejezésbe tér vissza a vezérlés folytatva annak kiértékelését.

A visszatérési érték meghatározása több módszerrel történhet:

A függvénynév változóként használató a törzsében, a visszatérési érték az utoljára kiszámított érték.

A függvény nevéhez kell az értéket rendelni, itt is az utoljára kapott érték a visszatérési érték.

Külön utasítással adjuk vissza a visszatérési értéket a törzsben, ez befejezeti a függvényt.

Minden megírt programnek kell lennie egy olyan különleges egységének amit főprogramnak nevezünk. A betöltő ennek adja a vezérlést és a többi egységet is ő manipulálja, szabályosan befejeződik a program ha ez szabályosan ér végét.

Szabályosan fejezül be, ha elértek a végét és van visszatérési érték, vagy olyan utasítást használunk ami befejezte és van visszatérési érték, vagy olyan utasítást használunk ami be is fejezeteti és meg is határozza a visszatérési értéket.

Azokban az esetekben amikor nem goto-t használunk visszatérünk a kifejezés kiértékeléséhez.

Nem szabálosan fejezül be: ha goto utasítást használtunk, nincs visszatérési érték.

A programegység képességei tartozik az, hogy újabb programegységet hívhatnak meg, az így kialakulú láncot nevezük hívási láncnak. A lánc feje mindenkor a főprogram. A lánc felfogható stacknek is(verem), hiszen mindenkor az utoljára "betett" program vége először be működését(szabályos esetben). A lánc dinamikusan alakul.

Rekurzióról beszélünk ha egy program önmagát hívja meg vagy amikor már egy hívási láncban lévő egységet hív meg. Minden rekurziós megoldás átírható iteratív algoritmussá, ez a kevesebb memória foglalás miatt gyorsabb programot eredményez. Nem minden nyelvben értelmezett a rekurzió.

Előfordulhat olyan funkció egyes nyelvek esetében, amikor megengedett egy másodlagos belépési pont használata, ekkor a törzsnek csupán egy tartománya hajtódik végre.

Olyan programegységet ami egy programegységen belül van blokknak nevezünk.

A blokkoknak van kezdetük, törzsük és végük. Törzsben ugyanúgy lehetnek deklarációs és végrehajtható utasítások, mint előző esetekben, és keverhetők vagy nem keverhetők. Nincs paraméterük, nevük létezése nyelv függő. Ott helyezhető el, ahol végrehajtható utasítás állhat. Vezérlést átadni gotoval van lehetőség vagy automatikusan oda kerül a vezérlés a program működése során. Ugyanígy befejezhetődhet. Hatáskörök elhatárolására használatos.

Paraméterkiértékelésről beszélünk, amikor az aktuális paraméterek a formális paraméterekhez csatolódnak az alprogram meghívásakor. Ekkor határozódnak meg, olyan információk, amelyek kommunikációt adják paraméterek megadásánál.

Mindig a formális lista elsődleges, egy darab van belől de aktuális listából(paraméter) annyi van ahányszor meghívjuk a programot, a paraméterkiértékelésnél.

Az dönti el, hogy melyik formális p-hoz melyik akruális paraméter, hogy milyen kötésről beszélünk.

Sorrendi kötés sorrenbeli hozzárendelést jelent. Ez az alapértelmezett a legtöbb nyelv esetében.

Név szereinti kötés mi adhatjuk meg a hozzárendelést az aktuális paraméterlistában. Ezt a technikát néhány nyelv ismeri. E két kötés kombinációjának használata is megtörténhet olyan módon, hogy aktuális plista elején sorrendi, majd név szerinti kötés van.

Ha fix számosságú a formális paraméterek listája akkor vagy meg kell egyeznie az aktuális paraméterek számának megadáskor, vagy kevesebb (csak értéki szerinti megadás során) és ekkor alapértelmezett érték rendelődik a meg nem adott helyekre.

Dinamikus számosság esetén a aktuális paraméterek száma is dinamikus, azaz tetszőleges mennyiségű.

A két paraméterlista elemeire további megszorítások is értelmezve vannak, például meg kell egyezniük a paraméterek típusainak vagy legalábbis az aktuális paraméterlista elemeinek konvertálhatóak kellenek lenniük a formális paraméter típusára.

A paraméteradás kommunikációt valósít meg programegységek között, minden van hívó és hívott, rendre kiad és kap.

Létezik érték, cím, eredmény, érték-eredmény, név és szöveg szerinti átadás.

Érték szerinti esetben a formális paraméter címkomponenssel rendelkeznek , az aktuális paraméterek értékkomponenssel kell rendelkezniük. E érték meghatározódik a pkiértékelés folyamán és majd végén a címkomponensre kerül. Egyírányban áramlik az információ, mivel értékmásolásról beszélünk, ez hosszadalmas folyamat, minél bonyolultabb struktúrát szeretnénk átadni.

Címes esetben nincsen címkomponensük lefoglalva a hívott alprogram helyén a formális paramétereknek, de címkomponenssel rendelkezniük kell az aktuális paramétereknek a hívó félen. A formális paraméter címkopense az akt paraméter címe, amit átadásra kerül, lesz. Kétirányú információ áramlás, alprogram hívót eléri, ez veszélyes!

Kötelezően rendelkezniük kell címkomponenssel az aktuális paramétereknek eredményes alapú paraméterátadásnál. Az alprogram futási idő alatt nem használja a neki átadott kiértékeléskor megkapott aktuális paraméter címét, de működése végén átmosolja a formális paramétert a címkomponensre. Egyírányú kommunikáció.

Az érték-eredmény verziójánál az előző megkötést kiegészíti, hogy az alprogramnak értékkomponenssel is rendelkezniük kell. Kiértékeléskor a hívotthoz adódik az akutális paraméter címe és értéke, de a címét nem használja az alprogram. Végén a formális paraméter átmásolódik az aktuális paraméter címére. Kétirányú, értékmásolás kétszer.

Név szerintinél rögzül az alprogram szövegének környezete, az információ áramlás iránya a szövegkörnyezettől függ.

Szöveg szerintinél amikor a formális paraméter neve legelőször fordul elő a szövegben(alprograméban) akkor kerül rögzítésre a szövegkörnyezet és felülírása a formális paraméterek. A hívás utáni azonnal munkához lát az alprogram. C-ben csak egyetlen (érték szerinti) paraméterátadási mód van.

Az input paraméterek az alprogramnak adnak át információt a hívótól. Output paraméter ennek fordítottja, és van a kettő kombinációja amikor oda-vissza megy az információ.

Blokkok csak programegységen belül lehetnek, vannak nekik formálisan kezdetük, törzsük és végük. Kezdetet alapszó jelöli, törzsben utasítások sorozata lehet.

Nevük csak egyes nyelvekben lehet, no paraméter. Ott helyezhető el, ahol a végrehajtható utasítás is.

Vagy rákerül szekvenciálisan a vezérlés, vagy használhatunk GOTO utasítást egy felette lévő címkére. Nevek hatáskörének elkülönítésére használatos.

Hatáskör alatt azt értjük, hogy a szöveg egy részében egy név ugyanarra a eszközre hivatkozik, tehát láthatóságról beszélünk.

Programegységen belül beszélünk lokális, amiket nem ebben deklarálunk pedig szabad névnek nevezzük.

Ha megállíptjuk egy név hatáskörét, akkor hatáskörkezelésről beszélünk. Hatáskörből kétféle van, statikus és dinamikus hatáskörkezelés.

Fordítási időben történik a statikus hatáskörkezelés, fordító végzi, addig rekurzívan halad a programegységből kifelé ameddig meg nem találja lokális névként. Ha nem volt deklarálva kint akkor hibát dob, ha olyan nyelvről beszélünk ami szerint minden deklarálni kell, ellenben automatikus deklaráció fog végrehajtódni.

A hatáskör csak befelé terjedhet! Globális név az, ami az adott programegységen nincs deklarálva, de kívül igen.

A dinamikus hatáskörkezelés futási időben zajlik le, hívási láncot felhasználva megy vissza a láncon ameddig lokális névként nem találja a nevet. Ugyanazok az esetek zajlanak le, mint előbb, vagy hiba, vagy automatikus deklaráció.

Név hatásköre amiben dekláráltuk programegység dinamiku shatáskörkezelésnél, és azok amik ebből induló hívási láncokban helyezkednek el.

Statikusnál a forrásszöveg alapján egyértelműen megállapítható a hatáskör. Dinamikus esetben változhat futási időben. Eljárásorientált nyelvek statikus hatáskörkezelést implementálnak legtöbbször.

A C ismeri a blokkotkat és a függvényt, lambda kon kívül nem ágyazhatóak be függvények más programegységekbe, blokkokat tehtjük akárhogyan.

Megjegyeztehű, hogy a függvényeknek Cben az alapértelmezett visszatérési értéke jelölt egész. A függvénykből a return kulcsszóval térünk vissza.

Az extern kulcsszóval rendelkező nevek hatásköre az egész program, illetve a futási idő végéig létezni fognak.

Az auto attribútum miatt hatáskörkezelésük statikus, de láthatóak deklarációjuktól kezdve, dinamikus élettartam, nincs kezdőérték automatikusan.

A register hatására értéke regiszterben tárolódik, ha van szabad.

A static hatására élettartamuk a futási idő, fordítási egység a hatáskörük, van automatikus kezdőérték.

Az absztrakt adattípus enkapsulációt valósít meg, amely következetében a logikailag összeillő adatokat egy struktúrában tudjuk kezelní. Nem ismerjük az implementációt, interfészeken keresztül tudjuk felhasználni programozás során.

A procedurális asbztrakció eszköze a generikus programozási paradigma. Bármely nyelvbe beépíthető, lényege, hogy paraméterezhető forrásszöveg-mintát adunk meg. A mintaszövegből az aktuális paraméterek segítségével állítjuk ki a mintaszövegből a konkret szöveget, amit majd lefordítunk. Típus paraméterrel paraméterezzük.

Input/Output

Az I/O területének kezelése nagyon eltérő programozási nyelvenként, hiszen az I/O kezelése az operációs rendszer feladata amikból rengeteg féle létezik(Haiku, Linux disztrók, BSD disztrók...) és mindegyik másképp kezeli az I/O kérdést. Ahhoz, hogy egy programozási nyelv hordozható legyen leválasztották a nyelvektől(némelyiknél) az I/O, így az I/O implementációját róbizzák az operációs rendszerre, ők csak egy interfész bocsátanak ki a programozó számára.

Az I/O a perifériákkal való kommunikációért felelős, amely az operatív tárba vár és onnan küld adatokat. Ez minden állományokkal történik meg amiből van logikai és fizikai. A logikai állomány olyan egy reláció aminek van azonosítója(neve) és attribútumai. A fizikai állomány meg a OS szintű perifériákon lévő tényleges adatállomány.

Az input állományból olvasni lehet csak, tehát léteznie kell a műveletek véghajtása előtt.

A output állomány az előbbi ellentettje, csak írni lehet bele, feldolgozás során jön létre a lemezen.

A kettő kombinációja az input-output állomány e kettő ötvözete.

Folyamatos adatátvitelnél eltér az ábrázolási mód tár és periféria között, a nyelvek ekkor folyamatos karakterláncnak nézik a periféria adatait, de megfelelő belső reprezentálás valósul meg a tárban. Konverzió történik, azaz olvasáskor meg kell mondani, hogyan osszuk fel az adatokat, íráskor pedig melyik helyen mennyi karakterrel jelenjen meg az adat a karakterláncban.

Megadásra pár alapeszköz:

Formátumos módú adatátvitel: meg kell adni az adathoz rendelt formátum segítsével az érintett karakterek darabszámát és a típust

Szerkesztett módú adatátvitel: maszkot adunk meg amely szerkesztő és átvivendő karakterekből áll. Szerkesztő karakterek adják meg, hogy a pozíció milyen kategóriájú karakternek kell lennie. A masz elemszáma adja meg az érintett karakterek mennyiségét.

Listázott módú adatátvitel: Karaktreslánca vannak beletéve a tördelő speciális karakterek.

Nincsen konverzó a bináris adatátvitel esetén, azaz a tábeli és periférián lévő reprezentáció megegyezik.

Ahhoz, hogy állományokkal tudunk dolgozni deklarálás, összerendelés, állomány megnyitás, feldolgozás és lezárás műveleteket kell végrehajtanunk.

A nyelv szintaxisának megfelelően deklaráljuk a logikai állományt, adunk neki nevet és attribútumokat(nyelv függő).

Összerendelésnél megfelejtetjük a logikai állományt a fizikaival. EZ nyelvi eszközzel történik, vagy operációs rendszer szintjén végezzük el.

Műveletek elvégzése előtt meg kell nyitnunk a fájlt, erre operációs rendszer speciális hívásai zajlanak le, az operációs rendszer nyílván tartja, hogy mely fájlok vannak megnyitva. Egy nyitott fájlból akármennyi processzus olvashat, de írni csak egy írhat egy időben.

Feldolgozáskor írunk, olvasunk a fájlból az előbb felsorolt módok alapján a megfelelőnek megfelelően megadjuk a nevet, formátumot, változólistát, maszkot. Kiíró eszközrendszerben a név mellé kifejezésítést is szerepelhetni kell, ezek kerülnek kiírásra kiértékelés után, szükséges emelett megadni a maszkot/formátumot. Binárisnál rekordot kell adnia a kifejezésnek.

Lezárásnál megint rendszerhívások fognak lefutni, le kell zárni az output-intput, output állományokat és az inputot is. Megszűnik az összerendelés logikai és fizikai állomány között.

Amikor például billentyűzetet használunk vagy a képernyőn megjelenik adat akkor implicit állománnal dolgoztunk, hiszen maga a billentyűzet és a monitor is fájlként van számon tartva a rendszeren(UNIX alatt minden fájl!), de ez el van "rejtve" a felhasználók elől. Létezik tehát logikai és fizikai állomány is ennél az esetnél. Szabványos perifériákról beszélünk itt(0 STDIN, 1 STDOUT, 2 STDERR).

Ahogy említve volt a C I/O rendszere nem a nyelvhez tartozik, hiszen akkor nem lenne hordozható operációs rendszerek között. Létezik neki bináris és folyamatos módú átvitele, utóbbi keveréke egy szerkesztett átvitel és formátumos átvitelnek. Az I/O függvények minimálisan egy karakter(bájt) vagy karaktercsoport(bájtcsoport) olvasását/írását biztosítják.

Kivételkezelésről általában

Kivételkezeléssel minden program rendelkezik, egy szinten. A kivételkezelés kivételkezelőrendszerrel történik, ez lényegéb olyan, mint operációs rendszer szintjén a megszakításkezelés és annak kezelője. A kivételek olyan történések amik pontosan az előbb említett dolgot váltják ki, megszakítást.

Minden kivétel besorolható valamilyen csoportba, például vannak olyan kivételek amik végtelen ciklust okoznának, ha erre nem lenne eleve egy alapértelmezett kezelés írva, tehát mondjuk a 0-val történő osztás. Vagy annak memórakezelésre vonatkozó kivételek, például kilépünk a processzunkhoz tartozó címtartományból megsértve ezzel más területét.

Mivel megszabhatjuk/megfigyelhetjük a kivételeket, figyelmen is kívül hagyhatunk néhányat(amit engednek) nyelvi szinten is. A letiltás is kivételkezelésnek minősül. Nem vesz ez esetben tudomást a programunk a kivétel által kiváltott megszakítás legenerált jelre.

A kivételeknek vannak neveik és azonosítóik.

Kivételkezelésről számos kérdést feltehetnénk, nézzünk meg néhányat:

Milyen beépített kivételkezelések lehetnek egy nyelvben? Például 0-val való osztás, vagy out-of-bounds memóriaelérés,..stb

Lehetnek saját kivételeink? Természetesen, ellenben elég gyakran kéne új nyelveket megalkotnunk.

Van-e a hatóköre a kivételkezelésnek? Van, de akár az egész programra is bevezethetnénk kivételkezelést. Javaban pl a kivételkezelő egy blokk.

Kivételkezelésután hogyan fut tovább a program? Futhat tovább, de hiából függően le fog, vagy le kell állnia, utóbbi esetben kilövi a kernel. Ignorálhatjuk is a kivételt, ha olyan típusú.

A nyelveknek lehetnek beépített kivételkezelői, például a Javanak a virtuális gépe is képes erre.

Ha Java kód során valamilyen speciális kivétel bekövetkezik akkor létrejön a megfelelő kivétel-osztálynak egy példánya/objektuma. Ekkor a kivétel a Java virtuális géphez kerül, aminek feladata, hogy a megfelelő típusú kezelőnek átadja a kivétel kezelését. Javabána kivételkezelő egy blokk, tehát van láthatósága és azt meg is szablya.

10.2. KERNIGHANRITCHIE

A C nyelv kevés adattípushoz használ minősítők is hozzájuk. (char, int, float, double) Minősítők pl (short(16 bit), long(32 bit) int). Bevezetésük oka az volt, hogy más hosszúságú egészekkel is dolgozhasson a programozó. A signed/unsigned minősítők az előjel meglétére/hiányára utal, ha előjeles az egész vagy char, akkor -határ,+határ intervallumban vehet fel értéket az adott típus, ha előjelmentes, akkor 0,2*határ intervallumban. Valós típusoknál a pontosságot lehet növelni a long-gal.

Állandók is rendelkeznek a fenti típusokkal, minősítőkkel, pl: 2018 signed int is lehet. Léteznek karakterállandók pl 'a', értéke a gépi karakterkészletbeni kódszáma. '0'=48. Karakterorszarak is lehetnek állandók "alma", bizonyos karakterek pedig escape sorozattal adhatóak meg pl '\a' csipog, '\n' új sor. A '\0' null-karakter karakterorszat-állandót zár le. Az állandó kifejezés csak állandókat tartalmaz, fordítás során értékelődnek ki. #define MAX AZ enum felsorolt állandó, egész értékek listája. pl enum boolean{no, yes}; Explicit mód megadhatjuk a legelső egész, pl enum wat {e=5, g,h,z};, avgy összesnek. Állandók neveinek különbözőeknek kell lennie.

A C-ben változók neveinek első karaktere betűnek kell lennie(_ még jó, de nem ajánlott ezzel kezdeni a konyvtári eljárások gyakran ezt használják). A nyelv kulcsszavai nem lehetnek változónévek. Egész típusok(int, char) lebegőpontosak(float,double), lebegőpontosak pontosság eltérő, double pontosabb, de nagyobb hrlyet igényel. A nyelv megengedi, hogy ezek között konvertálunk, de ekkor például ha floatot konvertálunk egésszé a "pont utáni rész" leválik. Ha egy kifejezésben egy lebegőpontos utasítás van, tehát egyik operandus lebegőpontos akkor az eredmény is az lesz, az egész típus lebegőpontossá alakul.

A vezérléstátadó utasítások a végrehajtás sorrendiségét adják meg. Egy olyan kifejezés C-ben, mint x=666 utasítássá válik, ha ';' rakunk utána pl x=666; ';' utasításlezáró jel tehát. A {} zárójelekkel deklarációk és utasítások halmazát fogjuk be egyetlen egy blokkba vagy összetett utasításba, ez nyelvtanliag egy utasítás lesz. Ilyen pl a függvények utasításait behatároló {} vagy if,else,while,for...

Az if-else utasítást döntéshelyzet kifejezésére alkalmazzuk.

```
if (kif)
    utasítás
else
    utasítás
```

A else rész elhagyható, utasítás a kifejezés kiértékelődse szerint az első utasítást hajtja végre, ha az igaz volt, ellenben 2.-at. Az if-else else ága egymásba ágyazott if-else szerkezetnél opcionális, de előfordulhat olyan, hogy nem világos, hogy a jelenlévő else ég melyik if utasításhoz tartozik. Például:

```
if (n>0)
    if (x>y)
        z = x;
    else
        z = y;
```

A fenti forráskóban az else a benső if tulajdonában áll, tagolás is szemlélteti, tehát az else minden a hozzá legközelebb eső nem else ágat tartalmazó ifhez tartozik. Zárójelezéssel megoldhatjuk, hogy más ághoz tartozzon.

Az else-if utasítás

```
if (kif)
    utasítás
else if (kif)
    utasítás
else
    utasítás
```

Többszörös elágazások programozásának egyik legáltalánosabb módszere, a gép sorra értékeli ki a kifejezéseket és ha valamelyik igaz lesz akkor a hozzátartozó utasítást hajtja végre, majd befejezi a további vizsgálatokat. Például ilyet használhatunk a bináris keresés implementálásának:

```
int binsearch(int x, int v[ ], int n)
{
    int a, f, k;

    a = 0;
    f = n - 1;
    while (a <= f) {
        k = (a + f) / 2;
        if (x < v[k])
            f = k - 1;
        else if (x > v[k])
            a = k + 1;
```

```
        else
            return k;
    }
    return -1;
}
```

Másik eszköz erre(többirányú elágazás) a switch utasítás. Ez összehasonlítja egy kifejezés értékét több egész értékű állandó kifejezés értékével, végül végrehajtva a hozzáartozó utasítást.

```
switch(kif)
{
    case állandó_kif: utasítás
    case állandó_kif: utasítás
    .
    .
    default: utasítások
}
```

Mindegyik case ágban egész konstans(konstans értékű állandó), ha ez egyezik a switches fejében lévő értékkel akkor az utasításai végrehajtásra kerülnek. Default ha egyikével sem egyezik meg, default elhagyható. Az utolsó case-ben lennie kell egy break; utasításnak, ha a felette lévőkben nem volt. Nincs switch break nélküli. A case-k mint címkék működnek, utasításaik végrehajtása után a köv. címkére akar ugrani a switch, de ezt break; utasítással kizárjuk.

Ciklusszervezés while és for utasításokkal: minden fejében van a ciklusfeltétel, ami ha igazzá értékelődik ki végrehajtjuk a magot és ezeket a lépéseket addig ismétljük amíg a feltétel igaz. A for fejében a ciklusváltozónak kezdőértéket adunk, megadjuk a feltételt, és egy ciklus utáni értékadó kifejezést is.

```
for(int i=0; i<6; i++) ; //c99-től
int j=-1;
while(j < 0)
    j++;
```

A for lehet végtelen `for(;;)`; és a while is `while(1);..`

A programozó dönthet arról, hogy while vagy for használ nincs előírva.

```
void shellsort(int v[ ], int n)
{
    int t, i, j, atm;

    for t = n/2; t > 0; t /= 2
        for (i = t i < n; i++)
            for (j = i-t;
                j >= 0 && v[j] > v[j + t]
```

```
        j -= t
        atm = v[j];
        v[j] = v[j+t];
        v[j+t] = atm;
    }
}
```

A for-ban használhatunk ',' operátort is, ez lehetőséget ad arra, hogy egyes helyeken több kif. kerüljön elhelyezésre, pl indexek párhuzamos feldolgozásánál használhatunk ilyet. pl: megfordítjuk a karakterláncot

```
#include <string.h>

void reverse(char s[ ])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

A fv-k argumentumait, deklarációkban változókat elválasztó vessző nem azonos a vesszőoperátorral.

A do-while utasítás hárultesztelős utasítás, azaz egyszer mindenkor végre fog hajtódni a ciklusmag és csak utána nézzük meg, hogy fennáll-e a ciklusfeltétel igaz volta.

```
do
    utasítás
while(kif);
```

A break; continue; utasítások a ciklus működését befolyásolják. A break; hatására kilünk a ciklusból, a continue; hatására kihagyjuk a continue; utasítás utáni utasításokat és új ciklusba kezdünk.

A C nyelvben is van goto utasítás amivel a programkódban létrehozott címkékhez ugorhatunk és ott adjuk át a vezérlést. Nem használják.

```
for(....)
    for(....) {
        ...
        if (err)
            goto error;
    }
    ...
```

```
error:  
  a hiba kezelése
```

A címkéket minden kettőspont zárja, a goto és a címke bárholt lehet, de a gotoval csak azonos függvényben lévő címke érhető el.

Összefoglalva: utasításuk leírásuk szerint hajtódnak végre, egymás után, kivéve ha nem, jelezzük. Vannak címkézett, kifejezés, összetett, kiválasztó, iterációs és vezérlésátadó utasítások.

A címkézett utasítások nyelvtana:

```
címkézett_utasítás  
  azonosító: utasítás  
  case állandó_kifejezés: utasítás  
  default: utasítás
```

Azonosító címkét csak a goto utasítás céljaként használhatunk, címke csak a blokkjában érvényes, nem deklarálhatók újból. switch használja a case és default címkéket.

A kifejezésutasítások értékadások vagy függvényhívások legtöbbször. Ha nincs benne a kifejezés rész, akkor null-utasítás. Az összetett utasítás(blokk) több utasítást csoportosít így alkotba egyetlen egy utasítást, például a fvdefiníció magja. Egy blokkban csak 1 azonosítót lehet deklarálni, és nem szereplhet blokkon kívül.

A kiválasztó utasítások a lehetséges végrehajtási utak egyikét választja.

```
kiválasztó_utasítás:  
  if ( kifejezés ) utasítás  
  if ( kifejezés ) utasítás else utasítás  
  switch ( kifejezés ) utasítás
```

A if-s kifejezésnek mutató vagy aritmetikai kifejezésnek szükséges lennie, nyílván ha igazzá értékelődik ki végrehajtódik az if-s utasítások ellenben ha van else akkor azok. Az else ág nem egyértelműségét több if esetében már tárgyaltuk fentebb. switch utasítás csak egész értékekkel működik és a case részknél konstans értékeket kell használnunk. Ha végrehajtódott egy case utasításai break; utasítást illik végrehajtani amivel kilépünk a switchből, ellenben "átfolunk" az alsó caserekre. Default címkéből csak 1 lehet. Végrehajtáskor a switches kif. összehasonlításra kerül a case-k konstansaival, ha egyezés van akkor végrehajtódnak a megfelelő utasítások, ellenben default vagy üres utasítás. Iterációs utasítások és vezérlésátadó utasításokat fentebb már tárgyaltuk. ELőltesztelek ciklusok: for, while. Hátultesztelek: do .. while. goto címkéhez ugrik, a címke a gotoval egy függvényben kell lennie, continue a ciklusfeltétel kiértékelését idézi elő ezzel kihagyva utasításokat utána, break ciklusból lép ki, return pedig függvényből térít vissza.

10.3. Programozás

[BMECPP]

Bevezetés: A C++ lehetővé teszi az objektmoriemtált és generikus programozást. A C++-t Bjarne Stroustrup fejlesztette ki. A C++ a C nyelvre alapszik, A C a C++ részhalmazának tekinthető.

2.1.1. C-ben a függvényhívások esetén a oaraméterlistát üresen is hagyhatjuk. A C++ esetén a üres paraméterlista egynértékű azzal, hogy megadunk egy void paramétert (a zárójelek közé).

2.1.2. C++-ban kétfajta main függvénymegadás van: int main(), illetve int main (int argc, char* argv[]), ahol az argc a parancssorban megadott parancs argumentumainak száma, az argv pedig maga a parancs argumentumait kapja meg. A return hsználata nem kötelező

2.1.3. A bool típus a logikai igaz vagy hamis (true vagy false) értékeket képes visszaadni. A C-ben ez a bool még nincs, helyette az int vagy az enum típust használjuk erre a célra, úgy lehet megkülönböztetni int típus esetén hogy a 0-ás érték

2.1.4. A C++ már header fájlok terén is bővült, illetve a C-ben lévő wchar típus a C++-ban beépített pípus lett

2.1.5. A C++-ban megjelent a másik nagy és hasznos újítás, ami az hogy cílusok feltételei megadása közben illetve a függvények paramétereinek magadásakor is definiálhatunk változókat.

2.2. C++-ban lehetőségünk van úgyanolyan nevű függvényeket létrehozni, abban az esetben ha az argumentumaiak különböznek.

2.4. Ha van egy olyan függvénydeklarálásunk hogy void f(int i) és a mainben f(a)-val hívjuk meg akkor az i szimbólisan az a lemasolt értékére fog hivathozni. A következő példában az adott érték az a változó címe lesz, egy pointer, melyre a függvényben a pi szimbólummal hivatkozunk, a *operátorral tudunk hozzáérni az a változóhoz és változtatni is azon. Létezik cím- és értékszerinti paraméterátadás

3. Az osztály alapelve az egységezés, mivel egy adott doleg struktúrájának leírását foglalja magába, úgy kell elpézelní mint egyfajta kategóriát. Az osztályoknak lehetnek példányai, önálló egyedi amiket objektumoknak nevezünk. Az osztályok esetén tudjuk biztosítani, hogy az objektumok tulajdonságaihoz a program többi rész ene férjen hozzá, ne tudja változtani azokat, ezt a védekezést adatrejtésnek hívjuk. A példákon keresztül lathatjuk hogy a struktúrák esetében nemcsak tagváltozói hanem tagfüggvényei is lehetnek. A this poniterrel tagváltozókra tudunk hivatkozni.

Adatrejtés esetén a private kulcsszót hívjuk segítségül, melyet a struktúra vagy osztály definíciójába írunk. A private: után felsorolt tagváltozók és függvények csak az osztályon belül lesznek láthatóak majd. A C++ esetében érdemesebb az ostályz használni a struktúra helyett, struktúrákat a régebbi C nyelvben használtunk.

A konstruktur olyan specilis tagfüggvény melynek neve megegyezik az osztály nevével és automatikusan meghívódik amikor példányosítjuk az osztályt. Ha nem írunk mimagunk konstruktorát az osztályba, attól még van ott egy alapértelmezett konstruktur ami nem csinál semmit. Tehát a konstruktur végezi el az inicializálást, ezzel ellentétben pedig van a destruktur ami pedig a felszabadításra szolgál. A destruktur a hullám jellel való kezdésről könnyű felismerni, melyet az osztály neve követ. A destruktur akkor hívódik meg amikor megszűnik az objektum.

A dinamikus memoriakezelésre C-ben a malloc és a free függvények állnak rendelkezésünkre. C++-ban viszont már nem is függvényeket hanem operátorokat használunk ezek helyett, ez pedig a new és a delete. A new a lefoglalt típusra mutató poniterrel tér vissza, használat után ezt a delete-el szebadítjuk fel. Tömbök lefoglalása és felszabadítása esetén úgyanígy kell eljárni hozzájéve a tömb "jelet": []. Dinamikus adattagok térolása esetén megvalósítható a FIFO, ahol ha őj elemet adunk hozzá akkor meg kell növelnünk a dinamikus területet, amikor meg kiveszünk, akkor egyel csökkentjük. A másolókonstruktur esetén az új

objektumot egy már meglévő alapján inicializáljuk, másolatot akrunk látrehozni. Ha nem írunmk másoló konstruktort akkor bitenként másolunk, ellenkező esetben amásolót hívjuk meg.

A sablonok alatt olyan osztálysablnokat és függvénysablonokat értünk, amelyek deklarációja esetén bizonyos elemeket paraméterként kezelünk. A paramétereket explicit vagy implicit módon adhatjuk meg. A függvénysablonok deklarálását a template kulcsszóval kezdjük, majd kacsacsőrök között felsoroljuk a sablonparamétereit (pl class T vagy name T vagy type T) vesszővel elválasztva.

A hagyományos hibakezelés továbbfejlesztése a kivételkezelések, melyekkel sokkal átláthatóbbá tudjuk tenni a programunkat. C++-ban a kivételkezelést try-catch blokkal oldjuk meg (a 190-es oldalon tökéletes szemléltetést létünk erre) ami már a Pici könyv olvasónaplója esetében bemutattam.

10.4. Python bevezetés

Forrás <http://users.atw.hu/progmat/letoltesek/Bevezetes%20a%20mobilprogramozasba.pdf>

Általános információk

A Python egy magas szintű, általános célú programozási nyelv. A szkriptnyelvek családjába szokták sorolni. Guido van Rossum 1990-ben alkotta meg. Egyik legnagyobb erőssége a funkcionális standard könyvtára, jól bővíthető.

Python-futtatókörnyezet számtalan különféle rendszerre létezik, így több mobilplatformra is (Apple iPhone, Palm, Windows Phone).

A nyelv népszerűségét nagymértékben az egyszerűségének köszönheti. Elsősorban kliensszoftverek, prototípusok készítésére alkalmas.

A Python nyelv jellemzői

Amikor alkalmazásokat készítünk és szükség van egy olyan program rész megírására ami a probléma szempontjából irrelváns, elkészítése mégis sok időt venne igénybe akkor a Python egy nagyon hasznos opció. Nem kell fordítani, elég az értelmezőnek a Python forrást megadni és az automatikus futtatja is az adott alkalmazást. A Pythonra tekinthetünk valódi programozási nyelvként is, mivel sokkal többet kínál, mint az általános szkript nyelvek vagy batch file-ok.

A Python tulajdonságai

A Python nyelvhez szorosan kapcsolódó standard Python kódkönyvtár rengeteg újrahasznosítható modult tartalmaz, amelyek meggyorsítják az alkalmazásfejlesztést. Ilyen modulok találhatóak például fájlkezelés-rendszerkezelésre, különféle rendszerhívásokra és akár felhasználói felület kialakítására is. Illetve az internetre is egyre több Python példakód és leírás található amely segít a nyelveldajtásában.

A Python egy köztes nyelv, nincs szükség fordításra se linkelésre, az értelmező interaktívan is használható. A Python segítségével tömörebb mégis olvashatóbb programokat készíthetünk amelyek rövidebbek (általában), mint a velük ekvivalens C,C++ vagy Java programok. Ennek okai a következők:

A magas szintű adattípusok lehetővé teszik, hogy összetett kifejezéseket írunk le egy rövid állításban.

A kódcsoporthoz egyszerű tagolással (új sor, tabulátor) történik, nincs szükség nyitó és zárójelezésre.

Nincs szükség változó vagy argumentumdefiniálására.

A Python nyelv bemutása

Alap vető szintaxis

A kód szerkesztése

A Python nyelv legfőbb jellemzője , hogy behúzásalapú a szintaxisa. A programban szereplő állításokat az azonos szintű behúzásokkal tudjuk csoportba szervezni, nincs szükség kapcsos zárójelrevagy explicit kulcsszavakra. A sor végéig tart egy utasítás. Ha egy utasítás csak több sorban fér el ezt sor végére írt '\' -el kell jelezni.

AZ értelmező minden sort tokenekre bont. A token különböző fajtái a következők: azonosító, kulcsszó, operátor, delimiter, literál. A kis és nagybetűket Pythonban megkülönböztetjük. A Pythonban vannak lefoglaltak kulcsszavak, azok megtekinkhetőek a forrásban.

Típusok és változók

Típusok

Pythonban minden adatot objektumok reprezentálnak. Az adatokon végezhető műveleteket az objektum típusa határozza meg. Nincs szükség változók típusának megadására, azt a rendszer automatikus "kitalálja". Adattípusok a következők lehetnek: számok, sztringek, ennesek, listák, szótárak. A Pythonban is van a NULL értéknek megfelelő típus, itt None a neve.

Változók és alkalmazásuk

Pythonban a változók alatt az egyes objektumokra mutató referenciákat értünk. Maguknak a változóknak nincsenek típusai. A változó értékkadása egyszerűen '=' jel segítségével történik. A Python nyelvben egyaránt létezenek globális és lokális változók.

A nyelve eszközei

Például print metódus, amivel sztringet vagy más változót írhatunk ki a konzolra.

A nyelv támogatja a más nyelvekben megszokott if/elifelse kulcsszavakkal.

Természetesen támogatja a különféle ciklusok kezelését is, mint a for ciklus, while ciklus. Illetve támogatja a C-ben megismert break és continue kulcsszavakat is.

Címkeket a label parancsal helyezhetünk e a kód egyes részeiben , és ezekhez a goto parancssal ugorhatunk.

Függvények

Pythonban függvényeket a def kulcsszóval definiálhatunk. A függvények rendelkeznek paraméterekkel, amelyeknek a szokásos megkötésekkel és szintaxissal alapértelmezett értékeket is adhatunk. A függvényeknek egy visszatérési értékük van azonban visszatérhetnek például ennesékkel is.

Osztályok és objektumok

A Python nyelv támogatja a klasszikus, objektum orientált eljárásokat. Definiálhatunk osztályokat, amelyek példányai objektumok. Az osztályoknak lehetnek attribútumaik: objektumok, illetve függvények. Ez utóbbiakat metódusnak vagy tagfüggvénynek is hívjuk. Ezenkívül az osztályok örökölhetnek más osztályokvól is. Az osztályoknak lehet egy speciális, konstruktur tulajdonságú metódusa, az __init__.

Modulok

A Python a fejlesztés megkönnyítése érdekében sok szabványis modult tartalmaz. Például: appuifw, messaging,syinfo,camera,audio.

Kivételkezelés

A Python nyelv támogatja a váratlan helyzetek kezelésére az úgynevetett kivételeket. Ebben,egyszerű esetben a try kulcsszó után írva szerepel az a kódblokk, amelyben a kivételes helyzet előállhat, majd ezt az expect blokk követi , amelyre a hiba esetén kerül a vezérlés, illetve opcionálisan egy else ág. Az utóbbi kettőt kiválthatja egyetlen finally blokk.

III. rész

Második felvonás

**Bátf41 Haxor Stream**

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

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

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

11.4. C

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

11.5. C++

[BMECPP] Benedek, Zoltán & 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 a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPORG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEAHCackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésekért és hasznos észrevételeikért.

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