

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Dr. Bátfai Norbert

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

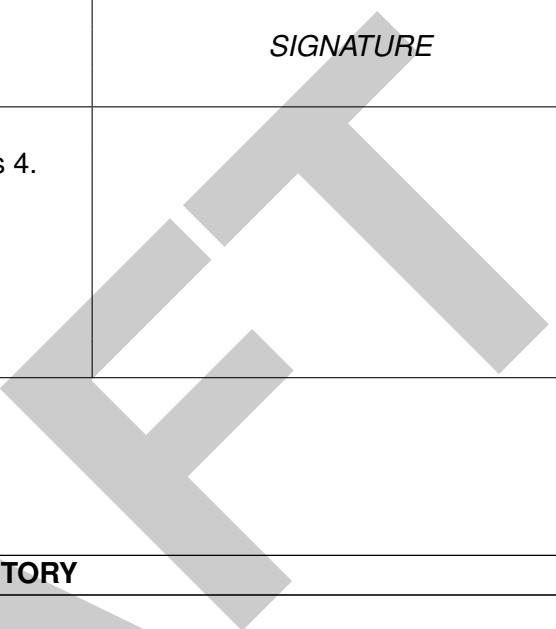
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is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

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

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

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

COLLABORATORS

	<i>TITLE :</i>		
	Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, ÁCs Bátfai, Margaréta	2019. május 4.	

REVISION HISTORY

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

Ajánlás

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

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

DRAFT

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	7
2.4. Labdapattogás	8
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	10
2.6. Helló, Google!	10
2.7. 100 éves a Brun téTEL	12
2.8. A Monty Hall probléma	12
3. Helló, Chomsky!	14
3.1. Decimálisból unárisba átváltó Turing gép	14
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	14
3.3. Hivatalos nyelv	15
3.4. Saját lexikális elemző	16
3.5. Leetspeak	17
3.6. A források olvasása	19
3.7. Logikus	20
3.8. Deklaráció	21

4. Helló, Caesar!	25
4.1. int *** háromszögmátrix	25
4.2. C EXOR titkosító	26
4.3. Java EXOR titkosító	26
4.4. C EXOR törő	27
4.5. Neurális OR, AND és EXOR kapu	28
4.6. Hiba-visszaterjesztéses perceptron	28
5. Helló, Mandelbrot!	30
5.1. A Mandelbrot halmaz	30
5.2. A Mandelbrot halmaz a std::complex osztállyal	31
5.3. Biomorfok	33
5.4. A Mandelbrot halmaz CUDA megvalósítása	34
5.5. Mandelbrot nagyító és utazó C++ nyelven	35
5.6. Mandelbrot nagyító és utazó Java nyelven	35
6. Helló, Welch!	36
6.1. Első osztályom	36
6.2. LZW	37
6.3. Fabejárás	37
6.4. Tag a gyökér	38
6.5. Mutató a gyökér	39
6.6. Mozgató szemantika	39
7. Helló, Conway!	40
7.1. Hangyszimulációk	40
7.2. Java életjáték	41
7.3. Qt C++ életjáték	41
7.4. BrainB Benchmark	42
8. Helló, Schwarzenegger!	44
8.1. Szoftmax Py MNIST	44
8.2. Mély MNIST	45
8.3. Minecraft-MALMÖ	45

9. Helló, Chaitin!	46
9.1. Iteratív és rekurzív faktoriális Lisp-ben	46
9.2. Gimp Scheme Script-fu: króm effekt	47
9.3. Gimp Scheme Script-fu: név mandala	47
10. Helló, Gutenberg!	48
10.1. Programozási alapfogalmak	48
10.2. Programozás bevezetés	49
10.3. Programozás	50
III. Második felvonás	51
11. Helló, Arroway!	53
11.1. A BPP algoritmus Java megvalósítása	53
11.2. Java osztályok a Pi-ben	53
IV. Irodalomjegyzék	54
11.3. Általános	55
11.4. C	55
11.5. C++	55
11.6. Lisp	55

Ábrák jegyzéke

5.1. A Mandelbrot halmaz a komplex síkon 30

DRAFT

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

DRAFT

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

DRAFT

**Bátf41 Haxor Stream**

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

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Végtelen ciklusok véletlen és direkt is létre jöhetsznek. Direkt alkalmaznak végtelen ciklust például a programok futása közben, hiszen nincsen előre megadott kilépési feltétel, de a bezárás gomb megszakítja a ciklust.

Egy szál 100%-on: A `for(;;)` és a `while(1)` is olyan végtelen ciklust hoz létre amely egy magot 100%-on használ ki, gyakorlatban ha célunk egy végtelen ciklus a `for(;;)` használatos, ez ugyanis egyértelművé teszi más programozók számára mit akartunk vele elérni és nem figyelmetlenségből van ott.

```
int main ()
{
    while(int i=1)
        for(;;

    return 0;
}
```

Egy szál 0%-on: 0%-os CPU használtságot a `sleep(0);` parancs segítségevel tudunk elérni. A 0 végtelen ideig "altatja" a folyamatot, ezt másodpercben megadva használjuk.

Több szál 100%-on: Ehhez párhuzmosítanunk kell a programunkat, amit a `#pragma omp parallel` használatával érhetünk el. Igy a program minden szálat 100%ban ki tud majd használni.

Fordítás: `gcc infinite_p100 -o infinite_p100 -fopenmp`

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

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

Megoldás videó:

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

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

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

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

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

main(Input Q)
{
    Lefagy2(Q)
}

}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

Tanulságok, tapasztalatok, magyarázat...

T100nak ha saját magát adjuk meg azt írja ki hogy nincs benne végtelen ciklus pedig az argumentuma maga egy végtelen ciklus

A T1000nek pedig nem tudja eldönteni magáról hogy micsoda ugyanis ha nem fagy le bekerül egy végtelen ciklusba amitől lefagy.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Csere segédváltozóval: Itt létrehozunk egy ideiglenes változót amiben eltároljuk az első változó értékét, majd az első változót egyenlővé tesszük a másodikat. Ezután a másodikat egyenlővé teszzük az ideiglenes változóban lévő értékkel

```
#include <stdio.h>
int main()
{
    int a=42, b=666;

    int c=a;
    a=b;
    b=c;

    return 0;
}
```

Ezen kívül lehetséges változó csere kivonás-összeadással és EXOr-ral is: //!!!

```
a = a - b; //Csere összeadás-kivonással
b = a + b;
a = b - a;

a = a ^ b; //Csere EXOR-ral
b = a ^ b;
a = a ^ b;
```

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 videónkon.)

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

If-el: x és y lesz a pálya mérete, lx és ly a labda helyzete, mx és my pedig a labda mozgásához kell majd. A labda pattogását egy végtelen ciklus és a kiir eljárás teszi lehetővé. Először növelem a labda jelenlegi helyzetét mx és my-al majd megnézem, hogy elérte e valamelyik falat és ha igen megfordítom a mozgatás előjelét. Késlelteni kell a kiirást amit a usleep-pel érek el.

```
int main(void)
{
    int x=90,y=10, //pálya mérete
        lx=1,ly=1, //labda helyzete
        mx=1,my=1; //labda mozgatása
```

```
char labda='o';

for(;;)
{
    lx+=mx;
    ly+=my;

    if (x-1<=lx) {
        mx*=-1;
    }

    if (y<ly) {
        my*=-1;
    }

    if (lx<0) {
        mx*=-1;
    }

    if (ly<0) {
        my*=-1;
    }

    kiir(lx,ly,labda);
    usleep (100000);

}
}
```

If nélkül: Itt a labda pattogását maradékos osztással és abszolút értékkel érjük el. Plus szükség van egy változóra amit mindenkorán többet növeljük majd szorozzuk a pálya méretének kétszeresével, ezt pedig kivonjuk a pálya magasságából/szélességből. Igy megkapjuk a labda kordinátáját és ha a változó nagyobb lesz a pályánál a kivonás miatt az abszolút értéke újra kicsi lesz.

```
x=abs(szelesseg-lepteto%(2*szelesseg));
y=abs(tmagassag-lepteto%(2*tmagassag));
rajzol(palya,x,y,labda);
lepteto++;
```

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:

Tanulságok, tapasztalatok, magyarázat...

A gépi szóhossz méretét a bitshift operátorral tudjuk megnézni. Mi a leftshifet alkalmazzuk ami a megadott szám bináris értékében az egyeseket balra tolja és pótolja a nullákat. Igy az egyet szépen elkezdjük tologatni balra és számoljuk hogy hányszor mozdítottuk el amíg elérjük az int végét és a ciklus leáll. A végen a változó érteke 31 lesz, de a szóhossz mégis 32 mivel az elsőt nem számolta.

```
int szam = 1, a = 0;

while (szam <=1) {
    a++;
}

printf("%u bites a gépi szó \n", a+1);
```

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:

Tanulságok, tapasztalatok, magyarázat...

A pagerankot a Google fejlesztette ki és arral szolgál, hogy megmutassa egy weboldal "értékét". Ezt az oldalra mutató linkek alapján számolják ki, minnél nagyobb Pagerankú oldal mutat a te oldaladra annan több "pontot" ad. Az alap elgondolás az volt, hogy emberek azért linkelnek be más oldalakat mert hasznosak.

Manapság már nem tölt be olyan fontos szerepet a google algoritmusában mert linkfarmokkal és fórumokon komment spameléssel manipulálható volt.

Létrehozzuk a kapcsoltati gráfot ami megmutatja hogy melyik link melyik oldalra mutat(halott linkek kezelésétől eltekintük)

```
int main(void)
{
    double L[4][4]={
        {0.0 , 0.0 , 1.0 / 3.0 , 0.0},
        {1.0 , 1.0 / 2.0 , 1.0 / 3.0 , 1.0},
```

```
    {0.0 , 1.0 / 2.0 , 0.0 , 0.0},
    {0.0 , 0.0 , 1.0 / 3.0 , 0.0}
};

double PR[4] = {0.0 , 0.0 , 0.0 , 0.0};
double PRv[4] = {1.0 / 4.0 , 1.0 / 4.0 , 1.0 / 4.0 , 1.0 / 4.0};
    4.0};
}
```

Ezután elindítunk egy végtelen ciklust ami akkor fejeződik be ha a pagerank kisebb lesz mint a csillapító tényező(damping factor, jelen esetben 0.00001).Ebben csinálunk egy for ciklust ami a kapcsolati gráfot tömb sorait megszorozza a pagerankkal, az értéket pedig betölti egy ideiglenes pagerankba, amíg ki nem lép a végtelen ciklusból.

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

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

if ( tavolsag(PR,PRv, 4) < 0.00001)
    break;

}
```

Ennek a feltétele pedig az hogy a tavolság függvény visszatérési értékenek kisebbnek kell lennie mint a csillapító tényező. Ez a függvény argumentumként megkapja a végleges és ideiglenes pagerankot és az oldalak számát, majd a két tömb i-edik elemének az értéket és ezek abszolút értékét összeadja.

```
double tavolsag(double pr[], double pr_temp[], int n)
{
    double osszeg= 0;
    for(int i=0; i<n; i++)
        osszeg += (pr_temp[i]-pr[i])*(pr_temp[i]-pr[i]);

    return osszeg;
}
```

2.7. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

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

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

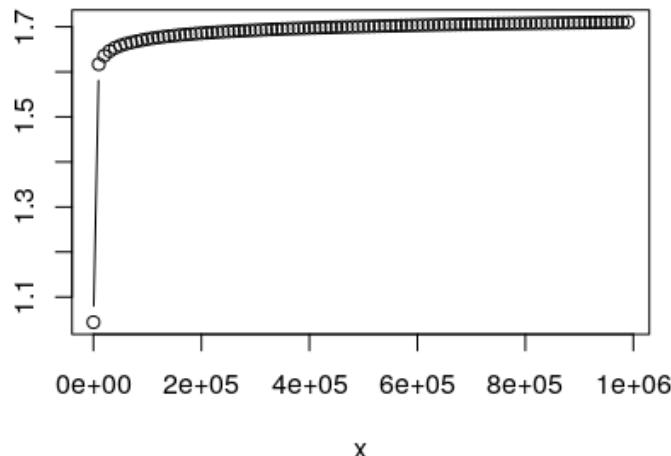
A prímek olyan számok amelyeknek csak két osztójuk van: 1 és saját maguk. Az ikerprímek pedig olyan prímek melyek különbsége kettő (pl:5 és 7). Így nézzünk egy programot ami kiszámol és ábrázol x prím között található ikerprímet.

```
primes = primes(x)
diff = primes[2:length(primes)]-primes[1:length(primes)-1]
idx = which(diff==2)
t1primes = primes[idx]
t2primes = primes[idx]+2
```

A program egy megadott x értékig kikeresi a prímekeket. Majd megnézi a köztük lévő differenciát (diff), ahol ez a differencia 2, annak az indexét egy tömbben(idx) tárolja(de csak az ikerprímpár első tagjának indexét, ezért kell a t2primes-nál a +2) tehát a prímek közül kiszűri, hogy melyek ikerprímek.

Az stp függvényben a megadott x-ig kiszedi az összes prímet és ha a különbségük 2 (diff==2), annak az indexét az idx-ben tárolja, de csak az első tagját így csinálunk két tömböt az első és második tagnak (t1 és t2primes). Az rt1plus2 összeadja ezek reciprokát, majd ennek az összegét adja vissza a függvény.

A Seq-el beosztjuk az x tengelyt(13-tól 1000000.ig, 10000-es lépésszámmal). A sapply pedig az y értékekhez rendeli a visszatérési értékeket. Pláttal pedig kirajzoljuk az ábrát.



2.8. A Monty Hall probléMA

Írj R szimulációt a Monty Hall problémára!

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

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

Tanulságok, tapasztalatok, magyarázat...

A Monty hall probléma egy statisztikai puzzle:

- Van 3 ajtó, 2 mögött egy-egy kecske, a harmadik mögött pedig egy autó.
- Te választassz egy ajtót (hívjuk A ajtónak). Természetesen te azt szeretnéd választani mi mögött az autó van.
- A játékmester megnézi a másik két ajtót (B és C) és kinyit egyet ami mögöt egy kecske van.

A játék a következő: Maradsz az eredeti ajtónál vagy átváltasz a másik zártra? Az ember először azt hinné, hogy minden esetben 50-50% a nyeremény esélye, valójában ha váltassz az esetek 2/3-ában nyersz! A kulcs az indoráció mennyiségeiben van: az elején semmit nem tudunk a az ajtókról, így az esélye hogy az auto az egyik ajtó mögött van 1/3, ha maradok az első választásomon ez ennyi marad, nem lehet sehogy a győzelmem esélye. E szerint a logika szerint a másik ajtónál kell legyen a maradék 2/3 esély. Sokkal jobban látszik a lényeg ha az ajtók mennyiségeit 3-ról 100-ra emeljük. Ugyanúgy kiválsztunk egy ajtót, a játékvezető pedig kinyit 98-at ami mögött biztos kecske van. Most maradsz-e az eredeti ajtónál (1/100) vagy a legjobb ajtót választod a maradék 99 közül ?

Ha a statisztikai magyarázat nem győz meg, egy R szimulációban kiszámoljuk melyik a jobb választás:

A kísérletek_száma változóban adjuk meg a próbálkozások számát, azaz hogy hányszor fusson le a szimuláció. A kísérlet és a játékos tömbököt feltöljük 1 és 3 közötti számokkal. A musorvezető egy vektor aminek a merete megegyezik a kísérletek számával

Egy for ciklussal bejárjuk a kísérletek_szamat és ha a játékos jól tippelt, akkor a mibol tömbbe a másik két ajtó kerül. Ezután Monty 'megtámaszt' egy ajtót, vagyis kiválaszt egyet a mibol tömbből. Aztán megnézzük hányszor nyerne a játékos ha az eredeti választásánál marad. Vagyis ide azok az elemek kerülnek amikor a játékos és a kísérlet azonos. Végül kiíratjuk a statisztikát, hogy mikor járnánk jobban ha minden változatnánk vagy ha maradnánk az első ajtónál.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Decimálisból unárisba, úgy váltunk, hogy annyi 1 est írunk le, amennyi a szám értéke vagy másképp fogalmazva amilyen messze van a 0-tól. Pl.: $n=90$ esetén 90 db 1 est kell leírnunk. Tehát itt pozitív számokat tudunk ábrázolni. A megvalósítás 2 féle lehet vagy indítunk egy for ciklust 0-tól és minden egyes lépésnél egy stringbe összefűzzük az egyeseket. Vagy pedig a számtól indítunk egy ciklust 0-ig és ugyan ezt tesszük.

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:

Tanulságok, tapasztalatok, magyarázat...

Forrás: <https://gyires.inf.unideb.hu/KMITT/b24/ch03s02.html>

A környezetfüggő grammatika olyan szabályok összessége, amely segítségével a nyelvben minden jelsorozatot képesek vagyunk előállítani.

I. $N = (S, X)$, $T = (a, b, c)$, ahol N nem terminális, T pedig terminális szimbólumok. A nyelvtan szabályai:

- 1. $S \rightarrow abc$
- 2. $S \rightarrow aXSc$
- 3. $Xb \rightarrow bb$

- 4. $Xa \rightarrow aX$

És ebből egy generált $a^2b^2c^2$ nyelv.

$S \rightarrow aXSc \rightarrow aXabcc \rightarrow aaXbcc \rightarrow aabbcc$

II. $N = (S, X, Y)$, $T = (a, b, c)$

- 1. $S \rightarrow abc$
- 2. $S \rightarrow aXbc$
- 3. $Xb \rightarrow bX$
- 4. $Xc \rightarrow Ybcc$
- 5. $bY \rightarrow Yb$
- 6. $aY \rightarrow aaX$
- 7. $aY \rightarrow aa$

És ebből egy generált $a^3b^3c^3$ nyelv.

$S \rightarrow aXbc \rightarrow abXc \rightarrow abYbcc \rightarrow aYbbcc \rightarrow aaXbbcc \rightarrow aabbXcc \rightarrow aabbYbcc \rightarrow aaYbbbccc \rightarrow aaabbccc$

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 forrása:

Tanulságok, tapasztalatok, magyarázat...

C-ben az utasítások egymás után leforduló parancsok, a nyelv alapjai. Tartalmaz alap egysoros utasításokat, többsoros utasítás blokkokat, iterációkat(for,while,do-while), operátorokat(--,++,!=,stb...) és vezérlő szerkezeteket(if,switch) is.

Váltani a két változat között a következő módon lehet: **gcc -std="c99/89" hivatkozas.c-**

A következő kódrészlet csak c99 alatt fordul le mivel c89-ben még nem lehetett for cikluson belül változót deklarálni.

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

int main()
{
```

```
for(int i=0; i<5; i++) {
    printf("\n");
}

return 0;
}
```

És a következő hibaüzenetet generálja:

3.4. Saját lexikális elemző

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

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

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/realnumber.l

```
#include <stdio.h>
int realnumbers = 0;
%
digit [0-9]
%%
{digit}*(\.{digit}+)? {++realnumbers;
    printf("[realnum=%s %f]", yytext, atof(yytext));}
%%
int
main ()
{
    yylex ();
    printf("The number of real numbers is %d\n", realnumbers);
    return 0;
}
```

Tanulságok, tapasztalatok, magyarázat...

LEXben az első rész tartalmazza a dekralációkat és a könyvtárakat és ennek a végén definiálunk és adjuk meg a karaktereket amiket keresni akarunk a szövegből. Mi itt megadjuk hogy számokat keresünk 0-9ig. A részeket a %% jelek választják el, ezután jön a második rész.

Itt megmondjuk mit csinálunk a karakterekkel ha megtalálja őket a lexer. Ha valós számot talál növeljük a számláló értéket és megmondjuk neki hogy írja ki a karaktersorozatot.

Ez után jön a main rész ahol meghívjuk az yylex() függvényt és kiirjuk hany számot találtunk.

Fordítás:

lex -o lexing.c lexing.l

gcc .o lexing lexing.c -lfl

Ezután elkezdhetünk számokat irni a terminálba, ha meguntuk pedig a CTRL+D-vel állíthatjuk meg a programot.

3.5. Leetspeak

Lexelj össze egy l33t cipher!

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

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/l337d1c7.1

```
% {  
    #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  
};  
  
%}  
%%  
. {  
  
    int found = 0;  
    for(int i=0; i<L337SIZE; ++i)  
    {  
  
        if(l337d1c7[i].c == tolower(*yytext))  
        {  
  
            int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0));  
  
            if(r<91)  
                printf("%s", l337d1c7[i].leet[0]);  
            else if(r<95)  
                printf("%s", l337d1c7[i].leet[1]);  
            else if(r<98)  
                printf("%s", l337d1c7[i].leet[2]);  
            else  
                printf("%s", l337d1c7[i].leet[3]);  
  
            found = 1;  
            break;  
        }  
  
        if(!found)  
            printf("%c", *yytext);  
    }  
}
```

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

Tanulságok, tapasztalatok, magyarázat...

Elsőként definiáljuk a lexer struktúra méretét. Ezt a define L337SIZE ,ez mutatja meg a sorok számát

Majd létrehozunk egy struktúrát. Itt definiálunk egy karakter változót és egy 4 elemű mutató tömböt. Ezután létrehozzuk a struktúra fő részét itt az első elem karakter típusú, amelyet majd vizsgálunk, a 2. elem egy 4 elemű tömb, amelyben a karakter helyettesítési lehetőségeit tároljuk.

Következőnek az utasítás része jön a lexernek. Itt behozunk egy változót, amely azt jelzi, hogy megtalálta e a karaktert a struktúrában ha nem akkor visszaadja majd a lent lévő if magát a karaktert. Majd indítunk egy fort, amely átnézi a struktúrát keresve a beolvasott karaktert.

Ha megtaláltuk a karaktert akkor egy random számot generálunk, amely segít, hogy véletlenszerűen válasszunk a 4 opció közül, amelyet a 4 if segítségével érünk el és visszaadjuk, hogy megtaláltuk a karaktert.

Az utolsó részben találjuk az srandomot, amely a randomot hívja . A random generálásához az időt használja és hozzáadja a getpidet, ez azért kell, hogy jobban generáljon random számokat, vagyis nagyobb legyen a számok randomitása. Végül meghívjuk a yylex() függvényt.

3.6. A források olvasása

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

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

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

Bugok

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

i.

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

ii.

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

For ciklus ami nullától ötig megy, a `++i` pre-incrementet jelent, vagyis a ciklus legutára előtt növeli a változó értékét

iii.

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

For ciklus ami nullától ötig megy, a `++i` post-incrementet jelent, vagyis a ciklus legutára után növeli a változó értékét

iv.

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

For ciklus ami nullától ötig megy, tömb i-edik eleme egyenlő lesz a léptetővel, kivéve az elso elemet.

v.

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

For ciklus ami addig megy meg i kisebb mint n, és a d tömbmutatót kicseréli arra amire az s mutat.

vi.

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

Kiirja az a változót kétszer, először kettővel majd eggyel növelve.

vii.

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

Kiirja az a változót az f fügvénnyel való módosítás után és változtatás nélkül.

viii.

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

Kiirná az a pointert és az á változót, de pointereket nem `%d`-vel hanem `%p`-vel kel kiiratni.

Megoldás forrása:

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat...

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

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

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

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

Tanulságok, tapasztalatok, magyarázat...

1. Bármely x -hez létezik olyan y , hogy az y nagyobb mint az x és y prím.
2. Bármely x -hez létezik olyan y , hogy y nagyobb mint x és y prím és $y+2$ is prím.
3. Létezik olyan y , hogy bármely x esetén ha x prím akkor az x kisebb mint y .
4. Létezik olyan y , hogy bármely x esetén ha y kisebb mint x akkor x nem prím.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

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

Mit vezetnek be a programba a következő nevek?

- `int a;`
- `int *b = &a;`
- `int &r = a;`
- `int c[5];`

- `int (&tr) [5] = c;`
- `int *d[5];`
- `int *h();`
- `int *(*l)();`
- `int (*v (int c)) (int a, int b)`
- `int (*(*z) (int)) (int, int);`

Megoldás videó:

Megoldás forrása:

Az utolsó két deklarációs példa demonstrálására két olyan kódot írtunk, amelyek összehasonlítása azt mutatja meg, hogy miért érdemes a **typedef** használata: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr.c](#), [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr2.c](#).

```
#include <stdio.h>

int
sum (int a, int b)
{
    return a + b;
}

int
mul (int a, int b)
{
    return a * b;
}

int (*sumormul (int c)) (int a, int b)
{
    if (c)
        return mul;
    else
        return sum;
}

int
main ()
{
```

```
int (*f) (int, int);

f = sum;

printf ("%d\n", f (2, 3));

int (*(*g) (int)) (int, int);

g = sumormul;

f = *g (42);

printf ("%d\n", f (2, 3));

return 0;
}
```

```
#include <stdio.h>

typedef int (*F) (int, int);
typedef int (*(*G) (int)) (int, int);

int
sum (int a, int b)
{
    return a + b;
}

int
mul (int a, int b)
{
    return a * b;
}

F sumormul (int c)
{
    if (c)
        return mul;
    else
        return sum;
}

int
main ()
{
    F f = sum;

    printf ("%d\n", f (2, 3));
```

```
G g = sumormul;  
f = *g (42);  
printf ("%d\n", f (2, 3));  
return 0;  
}
```

Tanulságok, tapasztalatok, magyarázat...

DRAFT

4. fejezet

Helló, Caesar!

4.1. int *** háromszögmátrix

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Az alsó hároszög mátrix egy olyan kvadratikus (megegyező sor- és oszlopszámú) mátrix aminek főátlója fölött csak 0-ák szerepelnek. Az alsó háromszögmátrixokat sorfolymatonosan egy vektorban szoktuk tárolni.

nr-ben megadjuk a mátrix sor és oszlop számát és egy double*ra mutató mutatót. Az ifben double* nr-szeresát foglaljuk le, ha a malloc nem tud helyet foglalni akkor -1-es hibaüzenettel tér vissza. Majd a következő malloc-kal a sor elemeinek foglalunk helyet, mivel a double 8 bájtos az első sorban 1x8 majd következőben 2x8 és így tovább bájtot foglalunk le.

```
int nr = 5;
double **tm;

if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
{
    return -1;
}
```

Majd egy for ciklus feltölti a tömböt 0-14ig számokkal, egy másik pedig a megadott alsó háromszög mátrix formában kiírja a terminálba.

```
greg@greg-X510UAR:~/Prog1$ gcc haromszog_matrix.c
greg@greg-X510UAR:~/Prog1$ ./a.out
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,
```

Végül a free(tm)-el felszabadítjuk a malloc által lefoglalt helyet.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Az exor titkosítás egy egyszerű, de hasznos titkosítási forma, aminek a biztonsága a megadott kulcs hosszával növekszik, mivel a feltörésnél annál több lehetőséget kell majd megnézni.

Mivel a szöveg minden karakteren xor-t hajtunk végre, az adott karaktert a kulcs adott karakterével xorozzunk, de mivel a kulcs rövidebb lesz mint a titkosítandó szöveg egy idő után elkezd ismétlődni.

A programban a while beolvassa a szövegfájlt, majd a for ciklus minden egyes karakter titkosít exorral, ha nagyob a szöveg mint a kulcs ismétlődésével titkosít.

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

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Létrehozunk egy Exor osztályt, majd létrehozunk egy stringet, amelyben a kulcsot tároljuk és nyitunk két csatornát egy bejövőt és egy kimenőt az olvasás íráshoz. A throws a hibakezeléshez kell ha nem sikerül a beolvasás vagy kiiratás akkor hibát dob. Ezután definiálunk egy byte változót a buffernek és egy kulcs indexet, amely a kulcs első karakterére hivatkozik kezdetben és egy olvasott bajtokat, amely a beomenetről beolvasott szöveg hosszával egyenlő. Aztán a while-al olvassuk be a szöveget és letároljuk a bufferben közben a méretét az olvasott bajtokba. Aztán a forban titkosítunk a maradékos osztás azért szükséges, hogy ha a szöveg hosszabb mint a kulcs akkor a kulcs index ismét 0-tól kezdődjön mivel karakterenként történik a titkosítás. Majd a végén a write-al a megadott kimenetre írunk.

```
public class Exor {
```

```
public Exor(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 olvasottBájtak = 0;

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

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

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

        }

        kimenőCsatorna.write(buffer, 0, olvasottBájtak);

    }

}
```

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Az exor törő lényegében egy brute force variáns, ami végig próbálja az összes lehetséges kódolási módszert. Ez a kódoló tudatában könnyebb, de a valóságban sokkal több változót figyelembe kell venni, így sokáig tart a törés és e miatt nem is olyan elterjedt.

Itt annak eldöntésére, hogy a feltört szöveg megfelelő e, az átlagos szóhosszt (ami a szöveg hossza osztva a spacek számával) és a magyar mondatokban gyakran előforduló szavakat használjuk.

Az exortörésben meghívjuk az exor eljárást majd az exorozott szöveget átadjuk a tiszta_lehetnek vizsgálatra, ha passzol akkor majd a brute force-os forban az if igaz lesz és kiirja a terminálra a kulcsot és a megfejtett szöveget.

Majd a main részben a while ciklusban meghívjuk a titkos szöveget, majd az ezt követő for-ban nullázzuk a maradék buffert. Ezután jöhet a kulcs megtalálására szolgáló egymásba ágyazott for ciklusok amik a törést végzik és ha megtalláják kiköpik az eredményt.

Majd párhuzamosítjuk a for ciklust a gyorsabb feltörés érdekében:

```
#pragma omp parallel for
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)
                .
                .
                .
```

Így a program fordítása:

gcc exortör.c -fopenmp -O3 -o exortör
./exortör titkos_szöveg.txt

4.5. Neurális OR, AND és EXOR kapu

R

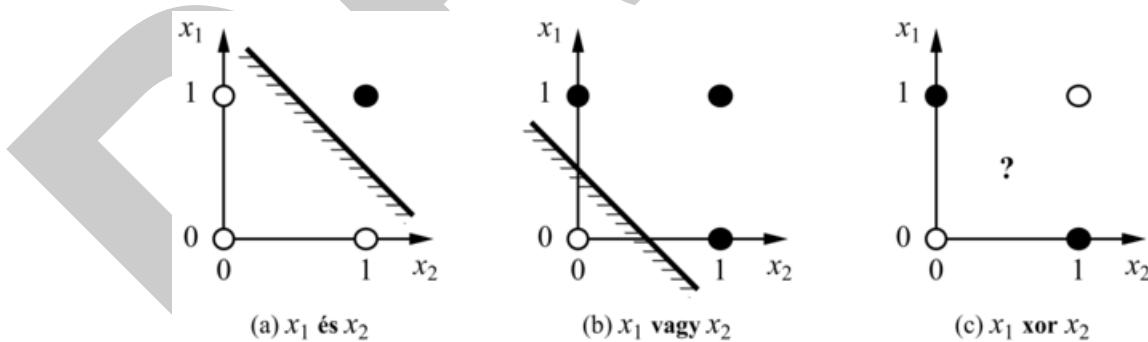
Megoldás videó: <https://youtu.be/Koyw6IH5ScQ>

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

Tanulságok, tapasztalatok, magyarázat...

Az AND és OR kapuk egyszerű perceptronként működnek. Ugyanis adatként megkapják az 1 vagy 0 bitjüket és, hogy milyen eredményt kell elérniük. Kb 100 próbálkozás után az algoritmus megtalálja a megfelelő súly értékeit és megközelítőleg jó eredményt ad, de az érdekkességek az exornál kezdődnek.

Itt ugyanis változtatás nélkül nem tud választ adni a program, nem tud "vonalat húzni" a két csoport között.



Ennek a megoldása rejtett neuronok bevezetése, amik lényegében egy új rétegnyi számolást tesz lehetővé, számuk növelése csökkenti a próbálkozások számát is.

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban tutoráltam Ranyhoczki Mariann-t.

Egy perceptron a gépi tanulás egyik legegyszerűbb megvalósítása. Az elképzélés szerint egyetlen neuront ábrázolunk, ami véges sok input alapján egy algoritmus segítségével két csoportba sorolja. Ehhez nemcsak a bemenő adatokat adjuk meg, hanem a megoldást is, így az algoritmus az adatokhoz rendelt súlyok változtatásával megpróbálja megközelíteni az elvárt megoldást. Moga az algoritmus általában a szignum függvény ami az adatok és súlyaik szorzatának összegét egy -1 és 1 közötti tört számként ábrázolja. Logikusan egy elem akkor tartozik az egyik csoprtba ha 0-nál nagyobb, a másikba ha kisebb. A program legelső próbálkozása alkalmával random súlyokat választ és az alapján, hogy a megoldása helyes vagy helytelen, csökkenti vagy növeli a súlyokat amíg a megfelelő megoldást nem kapjuk.

Jelen programunk annyiban tér el,hogy nem változtatja súlyait, hanem az első próbálkozásának eredményét adja vissza.

```
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.516501
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.500152
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.687392
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.730395
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.697868
```

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: bhax/attention_raising/CUDA/mandelpngt.cpp nevű állománya.

5.1. ábra. A Mandelbrot halmaz a komplex síkon

A Mandelbrot halmazt 1980-ban találta meg Benoit Mandelbrot a komplex számsíkon. Komplex számok

azok a számok, amelyek körében válaszolni lehet az olyan egyébként értelmezhetetlen kérdésekre, hogy melyik az a két szám, amelyet összeszorozva -9-et kapunk, mert ez a szám például a 3i komplex szám.

A Mandelbrot halmazt úgy láthatjuk meg, hogy a sík origója középpontú 4 oldalhosszúságú négyzetbe lefektetünk egy, mondjuk 800x800-as rácsot és kiszámoljuk, hogy a rács pontjai mely komplex számoknak felelnek meg. A rács minden pontját megvizsgáljuk a $z_{n+1} = z_n^2 + c$, ($0 \leq n$) képlet alapján úgy, hogy a c az éppen vizsgált rácpont. A z_0 az origó.

Azaz kiindulunk az origóból (z_0) és elugrunk a rács első pontjába a $z_1 = c$ -be, aztán a c -től függően a további z -kbe. Ha ez az utazás kivezet a 2 sugarú körből, akkor azt mondjuk, hogy az a vizsgált rácpont nem a Mandelbrot halmaz eleme.

A program elején megadjuk a kép méretét és az iterációs határt (MERET és ITER_HAT) ameddig nézni fogjuk a rácpont helyzetét, hogy a halmazban van e vagy sem. Ezután létrehozzuk az rgb képet, majd a két for ciklus végigmegy a rácson és a fenti algoritmus alapján feketére vagy fehérre színezi a rácpontot. Végül kiírjuk a kép elmentett nevét és hogy mennyi ideig tartott a számítás.

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

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](https://github.com/bhax/attention_raising/tree/Mandelbrot/3.1.2.cpp) nevű állománya.

```
// Verzio:  
#include <iostream>  
#include "png++/png.hpp"  
#include <complex>  
  
int  
main ( int argc, char *argv[] )  
{  
  
    int szelesseg = 1920;  
    int magassag = 1080;  
    int iteraciosHatar = 255;  
    double a = -1.9;  
    double b = 0.7;  
    double c = -1.3;  
    double d = 1.3;  
  
    if ( argc == 9 )  
    {  
        szelesseg = atoi ( argv[2] );  
        magassag = atoi ( argv[3] );  
        iteraciosHatar = atoi ( argv[4] );  
        a = atof ( argv[5] );
```

```
b = atof ( argv[6] );
c = atof ( argv[7] );
d = atof ( argv[8] );
}
else
{
    std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d <-
    " << std::endl;
    return -1;
}

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

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

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

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

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

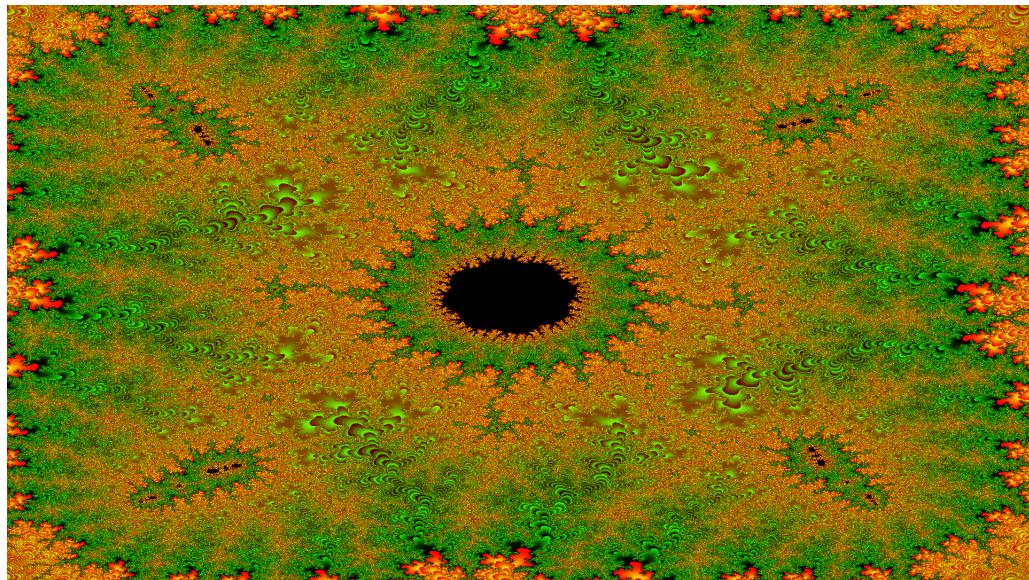
        // c = (reC, imC) a halo racspontjainak
        // megfelelo komplex szam

        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 ) );
    }
}
```



A 3.1.2.cpp nevű program a mandelpngt.c++ továbbfejlesztett változata. A főbb különbségek, hogy mostbár minden futtatáskor megadhatjuk a kép méretét, az iterációs határt és a b c d változókat amelyek a c complex szám valós és képzetes részét adják majd. Ezen felül használjuk a complex könyvtárat amivel egyszerűbb a complex számok képzése.

5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbqRzY76E>

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

A biomorfokra (a Julia halmazokat rajzoló bug-os programjával) rátaláló Clifford Pickover azt hitte természeti törvényre bukkant: https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf (lásd a 2307. oldal aljától).

A különbség a **Mandelbrot halmaz** és a Julia halmazok között az, hogy a komplex iterációban az előbbiben a c változó, utóbbiban pedig állandó.

Ezzel szemben a Julia halmazos csipetben a cc nem változik, hanem minden vizsgált z rácpontra ugyanaz.

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

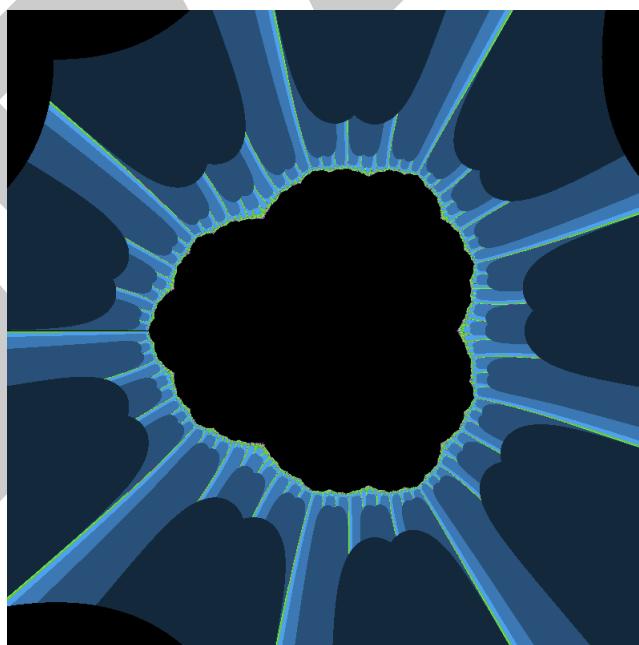
    for ( int x = 0; x < szelessseg; ++x )

        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 < iteracionsHatar; ++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 ));
```

biomorfos program abban különbözik az előzőtől, hogy most több argumentumot tudunk megadni tehát adhatunk kezdőértéket egy komplexszámnak cc-nek, amelyet majd z-hez minden hozzáadunk. Emellett még lényeges változtatás, hogy eddig csak az iterációt osztottuk maradékosan a kép színeihez, de mostmár konstansokkal szorozzuk meg a különböző színeket előállító képletet. Ez színesebb képet fog eredményezni és a több argumentum nagyobb szabadságot nyújt a felhasználónak, hogy különböző képeket alkossan. A biomorfos képek az egysejtűekre hasonlítanak, elég érdekes formákat lehet alkotni a program segítségével.



5.4. A Mandelbrot halmaz CUDA megvalósítása

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

Megoldás forrása: [bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu](https://bhax/bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu) nevű állománya.

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 programban a tartományt ugyan úgy az a, b , c és d változó határozza meg A képlet most is ugyan az mint a legelsőnél, amivel számoljuk az iterációt: $z_{(n+1)} = z_n * z_n + c$, ez a számolás a frakszal.cpp-ben van. A számításokat soronként küldjük vissza a frakablaknak, amely majd elvégzi a színezést. A változó deklarációja és inicializálása a számításokhoz a frakablak.h-ban található.A frakablak.cpp-ben definiáljuk, hogy mit csináljon a program az egérmozgására és, hogyha kijelölünk egy területet az egérrel akkor arra a területre nagyítson rá.

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_halmazzal

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

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...

A Polárgenerátor osztályban először deklarálunk egy nincsTárolt boolean típusú változót ami megmondja,hogy van e eltárolva random számunk, termászetesen alapesetben true értékű. Van egy tárolt változónk ami a nincsTárolt false értéke esetén visszaadja az eltárolt számot.

A következő() tagfüggvényben először megnézzük, hogy van e eltárolva számunk, ha nincs akkor kiszámítunk kettőt: egyiket visszaadjuk a usernek, a másikat elrakjuk a tárolt változóba és a nincstárolt értékét hamisra állítjuk; ha van eltárolt számunk akkor azt visszaadjuk és a nincsTárolt változót igazra állítjuk.

A mainben meghívjuk a PolárGenerátor osztályt g néven és forral csinálunk 10 példát a kimenetre.

Ez nagyban megegyezik ahhoz ahogy a JDK Random.javabán a Sun programozói csinálták:

```
private double nextNextGaussian;
private boolean haveNextNextGaussian = false;

public double nextGaussian() {
    if (haveNextNextGaussian) {
        haveNextNextGaussian = false;
        return nextNextGaussian;
    } else {
        double v1, v2, s;
        do {
            v1 = 2 * nextDouble() - 1;    // between -1.0 and 1.0
            v2 = 2 * nextDouble() - 1;    // between -1.0 and 1.0
```

```
s = v1 * v1 + v2 * v2;
} while (s >= 1 || s == 0);
double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
nextNextGaussian = v2 * multiplier;
haveNextNextGaussian = true;
return v1 * multiplier;
```

6.2. LZW

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

Megoldás videó:

Megoldás forrása:

A binfa algoritmus egy vektorban tárolt bitsorozatból bináris fa típusú adatszerkezetet csinál.

Létrehozunk egy binfa nevű struktúrát amiben deklarálunk egy int érték és 2 struct típusú mutatót, amik a gyökér jobb és bal oldali elemeire mutatnak.

A BINFA_PTR függvény fogja visszaadni majd a p eredményét, eg sikeres memória foglalás után. A kiir és szabadít függvényeknél extern-nel jelezzük hogy majd később deklaráljuk őket. A mainben létrehozunk egy char változót amib mindenkor minden lementjük éppen milyen elemet olvasunk be. Aztán deklaráljuk a fát és a fa mutatót ráállítjuk a gyökérre és a while ciklusban a jobb vagy bal oldalra rajuk az elemet attól függően hogy 1 vagy 0

A kiirral végigmegyünk a fán és növeljük a mélyságet minden ugrásnál majd ábrázoljuk magát a bináris fa kinézetét a standard outputon. A szabadit felszabadítja a az erőforrásokat használat után.

```
0111001010110101011101001011110
-----1(3)
-----1(2)
-----1(6)
-----1(5)
-----1(4)
-----0(5)
-----0(6)
-----0(3)
---/(1)
-----0(2)
-----0(3)
melyseg=6
greg@greg-X510UAR:~/Prog1/Gitlab/konyv/5_Welch$
```

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása:

Inorder: először a gyökér bal oldalát, majd magát a gyökeret és végül a jobb oldalt írjuk ki.

Preorder: először kiírjuk a gyökeret és csak ez után a bal és jobb oldalt.

Postorder: kiírjuk a bal és jobb oldalát a fának és végül a gyökeret.

Ez a kódban annyit jelent, hogy Inordernél a for ciklus előtt van az egyes gyerekek kiírása a nullásoké pedig utána. Preordernél minden a kettő mögötte, Postordernél pedig minden a kettő előtte van.

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

Postorder

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

Preorder

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása:

A C-s változattól abban különbözik, hogy használhatunk osztályokat. Létre is hozzuk az LZWBInFa osztályunkat majd deklarálunk egy konstruktort és egy destruktort. Majd túlterheljük az operátort a void operator-ban, amely paraméterül a char b-t kapja. Ezzel vizsgáljuk milyen elem megy be épp. Ha ez az elem 0 és a fának nincs 0 ás eleme akkor létrehozunk egyet neki. Ha van akkor ráállítjuk a fa mutatót. Ha ez az elem 1-es akkor hasonlóképpen működik. Majd jön a kiir eljárás, amely rekurzívan hívja meg magát. Argumentumként megkapja a gyökeret és azt, hogy mit kell kiirni ez az egyes gyermek és nullás gyermek lesz.

```
class LZWBInFa
{
public:

    LZWBInFa () :fa (&gyoker)
    {
    }
    ~LZWBInFa ()
    {
        szabadit (gyoker.egyesGyermek ());
        szabadit (gyoker.nullasGyermek ());
    }
}
```

Létrehozzuk a kiir eljárást, a kiiratás csak akkor tud megtörténni ha van elem a fában, itt inorder kiiratás történik. Ezután a fölösleges nem használt részeket felszabadítjuk a szabadíttal. Majd van egy protected rész ahol kiemeljük, hogy a fának van egy kitüntetett tag csomópontja a /. Ezután az osztályból kilépve a sima globális térbe létrehozunk egy usage eljárást, amelyel ha hibásan futtatnánk a programot segítséget nyújtunk a felhasználónak.

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:

A különbség mostmár az hogy a csomópontból pointer lett. Tehát a konstruktorba be keletti vinni kívülről az eddig átadott fa gyokeret. Mivel eddig a gyökér tagként szerepelt a csomópontba, de most mutató lett tehát könnyedén átadhatjuk az értékét a fának ami egy mutató. Miután a gyökérből pointert csináltunk így könnyedén elhagyhatjuk az és jeleket ugyanis alapból a memória címét fogja átadni majd nem kell érték szerinti referenciaként hivatkozni rá. Viszont, így hogy pointer lett a destruktorkban őt is fel kell szabadítani tehát bele írjuk a delete gyokeret a destruktorkba.

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása:

A mozgató konstruktornál létrehozunk egy új konstruktort. A konstruktorkban létrehozzuk a gyökér csomópontot és a mutatót erre irányítjuk. Majd a magasságot 0-ra állítjuk. Ezután jön a destruktur, amit azért itt hívunk meg mert a gyökérre mutató root mutatót null pointerre állítja aztán ha kiakarnánk törölni a semmit arra szegmentációs hibát dobna a fordító. Itt felszabadítjuk a dinamikusan lefoglalt memóriát és a gyokeret. Mivel az egy éses BT-t már ellőttük a másolásnál ezért 2 éssel jelöljük a mozgató konstruktort itt a megadott mutatókat egyenlővé tesszük az eredeti fa paramétereivel. És a gyökér mutatóját nullázzuk, mármint az eredetijét.

7. fejezet

Helló, Conway!

7.1. Hangyszimulációk

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

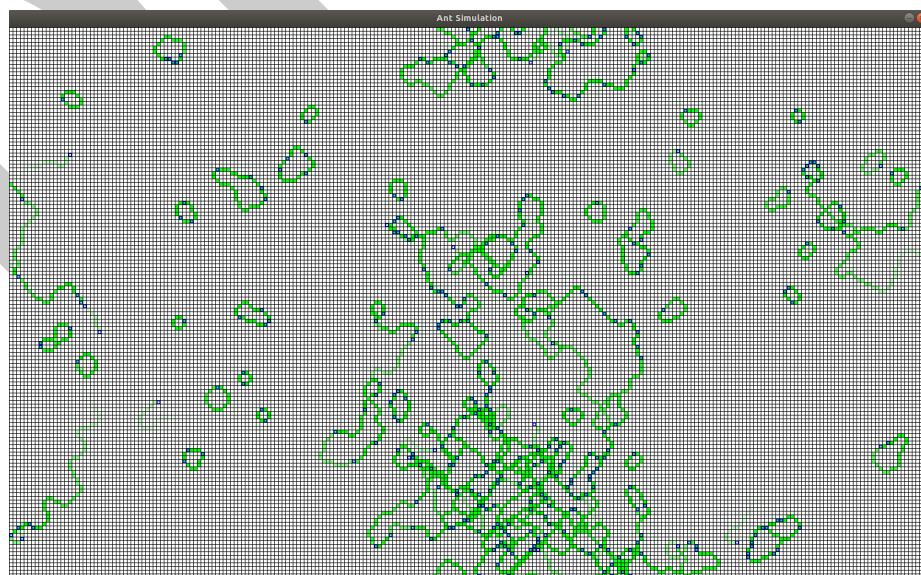
Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

Megoldás forrása:

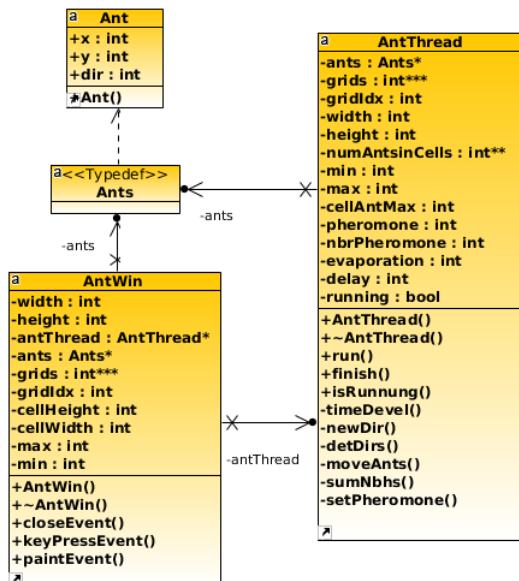
Forrás: <https://bhaxor.blog.hu/2018/10/10/myrmecologist?token=b90955bd5311e9986f0c3f99263ff386>

A hangyszimulációban egy hangy kolónia mozgását szimuláljuk. minden hangya feromon nyomot hagy, és ha feromon nyomra lép egy hangya, akkor elkezdi követni az előző hangya által hagyott nyomot, de ez egy idő után eltűnik ha nem lép rá senki. Minél erősebb a feromon annál nagyobb az esélye, hogy egy közeli hangya rálép. Először létrehozzuk a hangya osztályt, amelyben a hangyák koordinátáit és mozgásukat adjuk meg. A mozgásuk irányát maradékos osztással számoljuk.

A terminálban a program futtatásakor megadathatunk különböző flageket amelyek változtatják a tulajdon-ságait: -w a rács szélességét, -m a rács magasságát, -n a hangyák mennyiségét, -t a hangyák 2 lépés közötti sebességét, -t a párolgas értékét, -f a hanyott feromonok értékét, -a és -i a cellák maximum és minimum értékét , -c pedig azt adja meg hogy hány hangya lehet egy cellában egyszerre.



Az AntWin határozza meg az ablak tulajdonságait: hangyák kinézetét (ami az én változatomban kék négyzetekkel jelölünk), a rács méretét. Itt történik a key eventek implementálása is, P esetén a program megáll, Q vagy Esc lenyomásakor kilépünk. Ha nem adunk meg semmilyen kapcsolót futtatáskor az itt megadott értékeket használjuk.



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!

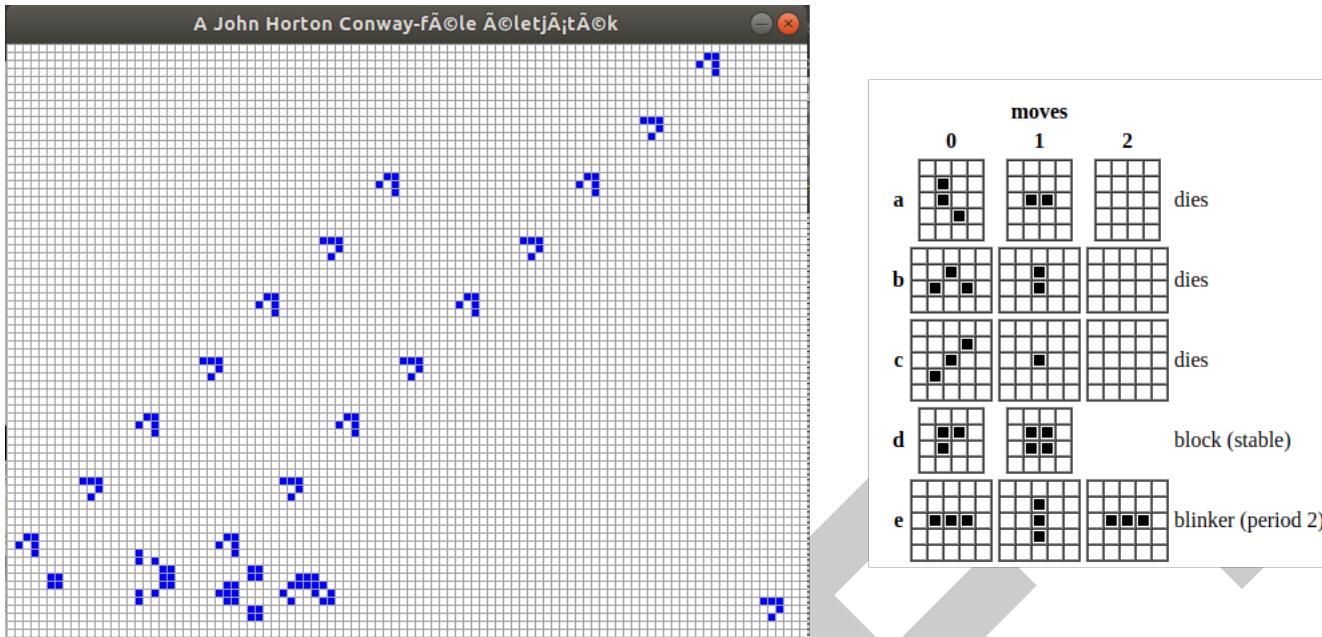
Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Forrás:<https://bit.ly/2G2objN>

Az eredeti életjátékot (The Game of Life) John Horton Conway matematikus találta ki 1970-ben. Ez lényegében egy "nulla-fős" játék hiszen csak a kezdeti állapotot adhatjuk meg a többi a játék szabályai szerint kell kiszámolni. Ezt meg lehetne tenni kockás papíron is, de az unalmas és hosszadalmas ezért ezt a program megcsinálja helyettünk.



A kísérletek száma változóban definiáljuk, hogy hányszor fusson le a kísérlet. Azaz a minták száma.

A kiserlet és a jatekos tömbök, amelyeket 1 és 3 közé eső számokkal tölt fel a sample. A műsorvezető egy vektor amelyet ugyan olyan méretűre deklarálunk mint a kísérletek száma.

Szabályok:

Egy x^*y méretű rácson játszunk, ahol a rácspontokat celláknak nevezzük. A játék kezdetén minden cella halott, mi dönjük el melyekre helyezünk sejteket, amik a játék elindítása után minden 'körben' a következő szabályok szerint élnek/halnak meg. (Egy sejt szomszédságán a körülötté lévő 8 cellát értjük)

- 1. Ha két vagy három szomszédja van, a sejt túléli a kört.
- 2. A sejt meghal, ha kettőnél kevesebb, vagy háromnál több szomszédja van.
- 3. Új sejt születik minden olyan cellában, melynek környezetében pontosan három sejt található.

Fontos megjegyezni, hogy a születések és elhalálozások egyszerre történnek, vagyis azok a sejtek amik meghalnak bele számítanak a születő sejtek meghatározásába. A szabályok szerinti sejtek felrakását és eltávolítását egy körnek vagy generációt nevezzük.

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

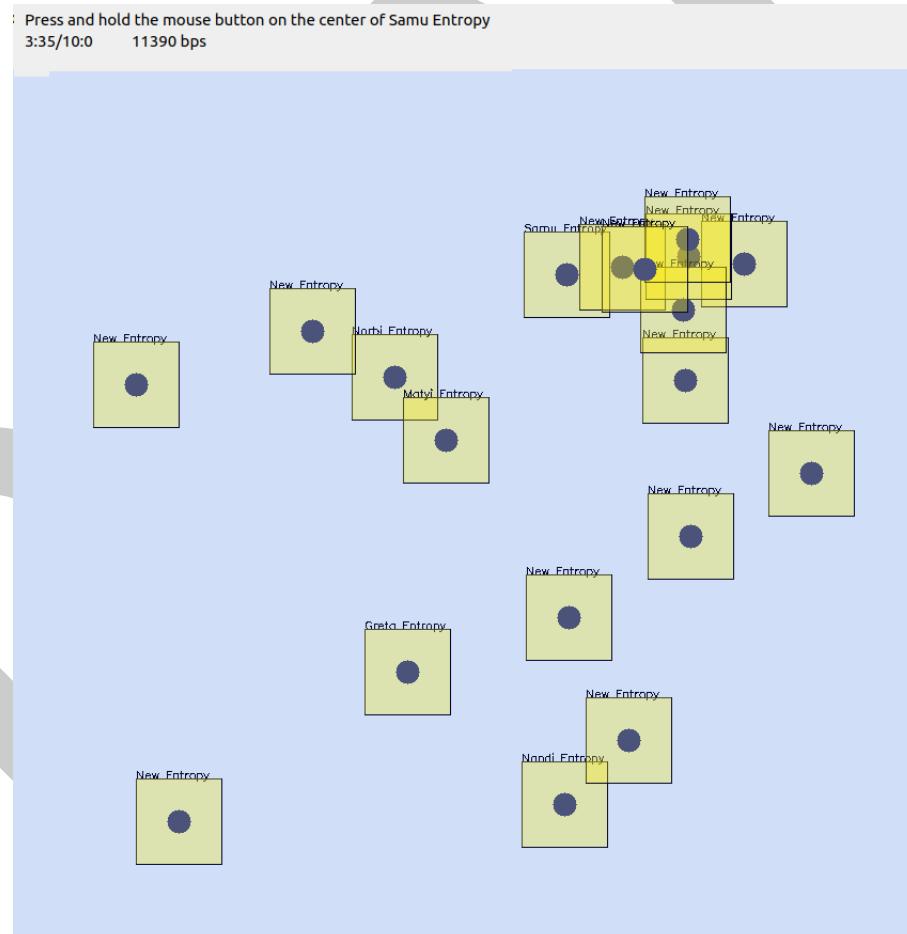
Forrás:http://real.mtak.hu/79216/1/it_2018_1_10_batfai_et_al.pdf

A BrainB benchmark program lényege a felhasználó koncentrációs képességének mérése. A teszt lényege, hogy az egér kurzort folyamatosan a 'Samu entropy' nevű nézet közepén, a kék körben kell tartani.

Samu és a többi objektum is random mozognak, a feladatot nehezíti, hogy minél tovább tartjuk rajta az egeret annál több új entropyt spawnolunk és gyorsaságuk is nő. Viszont ha elveszjük Samut csökken az entropyk száma és sebessége. A teszt 10 percig tart, amit kicsit soknak érzek mert elfárad az ember szeme és a keze is a folyamatos egérgomb lenyomástól a végére. Ha teljesítettük a tesztet a program egy külön fájlba elmenti eredményünket, amit a lost2found és found2lost változókból így számolunk:

```
int m1 = mean ( lost2found );
int m2 = mean ( found2lost );
double res = ( ( (double)m1+(double)m2 ) /2.0 ) /8.0 ) /1024.0;
textStream << "U R about " << res << " Kilobytes\n";
```

A BrainBThred konstruktorában deklaráljuk a hős osztályt, és állítjuk be a mozgásukat, amit random számolunk. Ezen felül itt írjuk meg az adatok méréséhez szükséges get_bps, get_w, lost, stb. függvényeket. A BrainWin-ben írjuk meg az eventeket amik érzékelik hogy az egér gombja le van-e nyomva vagy sem és ez alapján indítja vagy állítja meg a játékot. Ha Samun van ez egér új entropykat hozunk létre és növeljük az agilitijét, ha elveszítjük entropykat törlünk és lassítjuk. Ezen felül implementáljuk hogy a P gomb lenyomásával meglehessen állítani a játékot.



8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

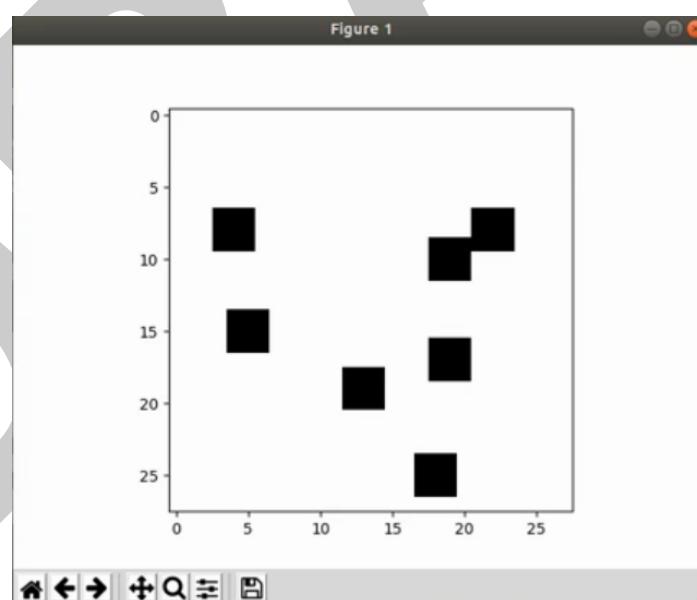
Python

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...

A tensorflowt a google készítette a gépi tanulást segíti, a tervezésben a fejlesztésben és a tanulmányozásban használják főként mivel készít adatáramlási gráfot, amelyben a node-ok a matematikai műveletek és az élek az az áramló adatok.



A tensorflow-ot import tensorflow kent hívjuk meg. A readimg függvény beolvassa a kép file-t majd de-kódolja, erre a későbbiekben lesz szükség.A program lényege, hogy a megadott képen szereplő számot felismerje.Ehhez meg kell tanítanunk a programunkat. Tehát elsőnek készítünk egy modelt.Majd ezen gyakoroltatjuk a programunkat. Majd futtatunk egy teszt kört ahol a program kiirja a becsült pontosságát. Ezután a 42 es tesztkép felismerése következik. Majd végül a beolvastott képünkön teszteljük a program működését.

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...

DRAFT

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:

Lispben két fő szintaktikai különbség van a híresebb programozási nyelvekhez képest. Az első az, hogy maga a program egymásba ágyazott S-kifejezések sorozata, a nyelv egyik nagy előnye és hátránya is egyben: egyszerű hozzá elemző programokat írni de az emberi szem hamar belekavarodik a sok zárójelbe. Második pedig hogy a nyelv prefix jelölést használ. Ez azt jelenti hogy egy például egy összeadásban először kell az összeadásjelet leírni és utána a művelet argumentumait(pl: + 2 2).

Rekurzív faktoriális

```
(defun fakt_r (n) (if(< n 1) 1 (* n (fakt_r(- n 1)))))
```

Definiáljuk a függvényünket a defun kulcsszó után fakt_r néven, ahol n-re helyettesíjük be a kívánt faktoriálist. Ha n kiseb mint 1 akkor 1-et adunk vissza eredményül, egyébként n-et megszorozzuk a rekurzívan meghívott fakt_r függvényvel az n-1 elemre.

Iteratív faktoriális

```
(defun fakt_i (n) (loop for i from 1 to n for result = 1 then (* ←
    result i) finally return result))
```

Definiáljuk a függvényünket fakt_i néven Majd egy loop for-al kezdünk, ami lényegében a lisp megfelelője az átlagos for-nak, ebben a ciklusváltozó i=1-től n-ig fut le a ciklus. minden iterációban megszorozzuk a result változót (amivel a végeredményt számoljuk) i-vel és így a vegére megkapjuk az adott faktoriálist amit returnnel visszaadunk.

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

Tanulságok, tapasztalatok, magyarázat...

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Tanulságok, tapasztalatok, magyarázat...

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

[?]

2.4 Adattípusok

Az adattípusok a programozási nyelvek egy absztrakt eszköze, ami minden valamilyen másik feladatunk részeként jelenik meg. Három tulajdonság jellemzi őket: tartományuk, műveleteik és reprezentációjuk. A tartománya az spectrum melyen belül az adattípus elemi felvehetnek értéket. Azok a műveletek tartoznak hozzá, melyeket az elemein eltudunk végezni. Reprezentációja pedig az egyes elemek tárolását mutatja meg bitekben(pl: C-ben egy chart egy bájton reprezentálunk). minden típusos nyelvben vannak beépített típusok, néhányban pedig magunk is deklarálhatjuk őket a három tulajdonságuk megadásával. Ezen felül úgy is letrehozhatunk saját típust hogy egy már meglévő tartoményt csökkentjük. Az adattípusok két nagy csoportja az egyszerű és az összetett típusok. Az egyszerű közé tartoznak az egész, valós, karakter és a logikai típus míg az összetetthez tömb és a rekord. Ezenfelül vannak speciális típusok, például a mutató aminek adatrészében egy másik elem memóriacíme van, alapvető feladata a tárhelyen lévő adat elérése.

2.5 Nevesített konstans

A nevesített konstansnak van neve, típusa és értéke. Mindig deklarálni kell és egy olyan értéket jelöl beszédesebb nevekkel amelyek a programunk alatt nem változtatnak értéket. Ezen felül ha meg kell változtatni az értéket, akkor elég egy helyen és nem minden előfordulásában átírni.

2.6 Változó

A változó egy olyan programozási eszköz melynek neve, attribútumai, címe és értéke van. A nevével jelöljük programkódunkban, attribútumokat deklarációkor kap(pl típus), címe a memóriában elfoglalt helye, értéke pedig ezen a helyen lévő információ.

4. Utasítások

Az utasítások segítségével írjuk le az algoritmusunk feladatát lépésenként. Egy részük csak a fordítóprogramnak szólnak, ezek a deklarációs utasítások, melyek valamelyen szolgáltatást kérnek vagy információt szolgáltatni a fordítónak. Másik részük pedig a végrehajtó utasítások, melyekből majd a programkód épül fel. Csoportosításuk:

- 1. Értékadó utasítás: Ezzel adunk a változóinknak értékét.

- 2. Üres utasítás: Mint nevében is szerepel az ilyen utasítások törzse üres. Ilyenkor a processzor egy üres gépi utasítást csinál.
- 3. Ugró utasítás: A program jelenlegi pontjáról egy másik meghatározott pontra dobuk a vezérlést.
- 4. Elágazató utasítások: Van kétirányú elágazó utasítás (if else) mely egy feltétel alapján két lehetséges tevékenység között választ. Ennek az összetettebb változata a többirányú elágazó utasítás(switch) ami egy feltétel alapján több lehetséges esetet próbál ráilesteni, ha sikerül azt hajtja végre, ha nem akkor van egy default válasz hajtódi végre.
- 5. Ciklusszervező utasítások: Lehetővé teszik , hogy a program valahány utasítását bármennyiszer megismételjük. Maga a ciklus magjában található a kód, ami megismétlésre kerül majd. Fontos része a ciklusoknak az ismétlődésre vonatkozó információ, ami lehet: egy feltétel igaz/hamis állapota, ezt a ciklus minden iteráció után(do..while)vagy előtt(while) ellenőrzi; lehet előírt lépésszámú ciklus is amely egy előre megadott értékszer hajtja végre a magját(for ciklus)
- 6. Hívó utasítás:
- 7. Vezérlésátadó utasítások: CONTINUE:Megnézi a ciklus ismétlődés feltételeit és vagy újrakezdi a cilust vagy befejezi azt; BREAK: A ciklust szabályosan befelyezi és kilép belőle; RETURN(érték): Befejezi a függvényt és visszadja az értéket a hívónak.
- 8. I/O utasítások
- 9. Egyéb utasítások

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

3. Fejezet Vezérlési szerkezetek

A C-nyelvben az utasításokat pontos vesszővel zárnak. Az utasítás blokkokat { }-el jelöljük.

3.2 If-else

Az if szerkezet döntést hozó utasítás. Ha(feltétel) utasítás else utasítás2 , az elsre nem mindig van szükség lehet olyan is, hogy ha történik valami akkor csináljon valamit a program,ha nem akkor ugorja át. Az else ág mindig a hozzá legközelebb lévő else nélküli ifhez fog tartozni. Ha nem így szeretnénk akkor az if hatáskörét {}-jelek közé kell tenni. Az ifnek van egy másik fajtája az else if itt több feltétel egymásba ágyazása történik. Itt a legutolsó else akkor fut le ha egyik feltétel sem teljesül. Amint egy teljesül a feltételek közül a program végrehajtja és kilép az else-if ágról.

3.3 Switch

A switchet többirányú programelágazások esetén használjuk itt valamilyen állandó értékhez rendeli az utasítást. A switchben case-eket hozunk létre, amelyek akkor futnak le ha teljesül az állandó, ezen kívül minden case-t break-el kell zárnai. Létezik egy default ág, amely akkor fut le ha egyik case feltétele se teljesül.

A break el nem csak a switchből tudunk kilépni hanem bármely ciklust képesek vagyunk vele megszakítani.

A for előírt lépésszámú ciklus, amelynek van egy kezdő értéke egy végértéke és egy lépésszáma.

A while addig fut amíg a ciklusfejben megadott feltétel hamis nem lesz.

10.3. Programozás

2.1.1 Függvényparaméterek és visszatérési érték

Ha C-ben megadunk egy függvényt paramétereik nélkül az bármennyi paraméterrel hívható, míg C++-ban azt jelenti, hogy nincs paramétere. Sőt ha visszatérési értéket nem adunk meg a C automatikusan intet fog visszaadni, a C++ pedig hibát mivel nincs alapértelmezett visszatérési értéke.

2.1.2 Main függvény

Az main függvény argumentumaként dekralálható argc a parancssorba beérkező argumentumok számát, míg az argv a paracssori argumentumokat adja meg. C++ nem muszáj returnt írni a main végére.

2.2 Függvények túlterhelése

Két függvényleg lehet ugyanaz a neve amíg argumentumlistájuk különbözik. A C++ a függvény nevet az argumentumokkal együtt tárolja, így a különböző változatoknak nem kell egyesével nevet adni.

2.3 Alapértelmezett függvényargumentumok

C++ nelvben lehetőség van a függvények argumentumainak alapértéket megadni. Ekkor ha a függvény hívásakot nem adjuk meg paraméterként akkor az alapértelmezett értéket használja.

2.4 Paramáteradás referenciátipussal

Ha azt szeretnénk C-ben, hogy egy függvény megváltoztassa az argumentum változó értékét pointerekkel kell hivatkoznunk rá, íg meg tudja változtatni az értékét. C++-ban ezt referenciátípus bevezetésével hidálják át. A C teljes pointerré alakítás helyett elég egy jelet írni a függvény deklarációjában a változónév előtt.

III. rész

Második felvonás

DRAFT

**Bátf41 Haxor Stream**

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

DRAFT

11. fejezet

Helló, Arroway!

11.1. 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

DRAFT

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, Tíhamér, *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/DEACHackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésekért és hasznos észrevételeikért.

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