DRAFT

Univerzális programozás

C0d1n6 my l1f3

Ed. BHAX, DEBRECEN,

2019. február 19, v. 0.0.4

PDF BY DBLATEX

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

DRAFT

COLLABORATORS

	TITLE : Univerzális programozás	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Pataki, Donát	2019. szeptember 18.	

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
0.2.0	2019-03-11	Első csokor kidolgozva.	Donát
0.3.0	2019-03-18	Második csokor kidolgozva.	Donát
0.4.0	2019-03-25	Harmadik csokor, frissítés, olvasónapló.	Donát
0.5.0	2019-04-01	Negyedik csokor, olvasónapló.	Donát

DRAFT

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.6.0	2019-04-08	Ötödik csokor, olvasónapló, firssítés.	Donát
0.7.0	2019-04-15	Feladatok további kidolgozása.	Donát
0.8.0	2019-04-22	Feladatok további kidolgozása.	Donát
0.9.0	2019-04-29	Kilencedik csokor, néhány firssítés	Donát
1.0.0	2019-05-10	Véglegesítés	Donát

DRAFT

Ajánlás

„If debugging is the way to remove bugs, programming must be the way to put them in.”

—Edsger Wybe Dijkstra []

DRAFT

Tartalomjegyzék

DRAFT

DRAFT

DRAFT

Ábrák jegyzéke

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 allokálni igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

DRAFT

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.

DRAFT

I. rész

Bevezetés

DRAFT

1. fejezet

Vízió

1.1. Mi a programozás?

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

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

DRAFT

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

DRAFT

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: https://github.com/DonatPataki/University/blob/master/prog1/ciklus.c

Az hogy mi a vételen ciklus, arra nem olyan nehéz rájönni. Egyszerűen egy olyan ciklus, ami a végtelenségig fut. Ennek megfelelően, ha akarunk egyet csinálni, akkor azt nagyon egyszerűen megtehetjük az alábbi kódcsipet segítségével.

```
while(1); //végtelen ciklus, ami 100%-on pörget egy magot
```

Miután megírtuk ezt a borzasztóan nehéz ciklust akár büszkék is lehetnénk magunkra, hogy van 100%-on tudunk pörgetni egy magot. Viszont a fenti kódcsipetből, akár azt is gondolhatánk, hogy nem fogja ennyire megterhelni a magot. Ez azonban egy téves következtetés, mivel a cpu nem tudja a forráskódban szereplő utasításokat elvégezni. Ezért előtte egy compiler-rel gépi kódot hozunk létre, ami igazából egyesek és nullák sorozata.

De hogy mégis szemléltetni tudjam, hogy miért lesz ennyire erőforrásigénylő a program, amiatt az egy sor miatt, ezért a -S kapcsolót használva lefordítom assemlby nyelvre, ami elég közel áll a gépi kódhoz, ami ténylegesen futattásra kerül és még valamennyire olvasható is.

```
jmp .L2
```

A lefordított programban megtalálható a fenti kódcsipet, ami azt mondja a cpunak, hogy menjen a program megadott részére és ez fog folyamatosan végrehajtódni, mégha a c forráskódból ez nem is látszik.

Ha egy olyan végtelen ciklust akarunk, ami nem terheli a processzort, akkor meg kell hívnunk a sleep function-t.

Ha pedig az összes magot le szeretnénk terhelni, akkor hasznánunk kell az omp.h nyújtotta lehetőséget. Ez nem más mint, hogy lehetőségünk van a for ciklusokat parallel módon futattni, ami a valóságban, annyi, hogy a for ciklus felosztja kisebb részekre és odaad minden magnak egyet. Az előbb leírtak alapján viszont a for ciklus is ugyanúgy terhelni fogja az éppen használt magot. Az openmp amúgy nem engedi lefordítani

DRAFT

a programot ha csak egy sima végtelen for ciklust adok neki. Ezért egy végtelen while loopba írtam egy parallel for ciklust, ami végeredményben a végtelenségig fog futni és még minden magot is terhel.

Végül a programot összeállítva kaptunk 3 fajta végtelen ciklust, amit én beraktam egy switch-be és így parancssori argumentumként el lehet dönteni, hogy melyik fajtát szeretnénk. Ezt egyszerűen úgy tehetjük meg hogy a program indításakor 0-tól 2-ig megadunk egy számot, mivel csak ezeket veszi figyelembe és enélkül nem jutna el egyik végtelen ciklusig sem.

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 vlgtelen ciklus:

```
Program T100
{

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

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{

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

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

  main(Input Q)
  {
    Lefagy2(Q)
  }

}
```

Mit for kiírni erre a `T1000(T1000)` futtatásra?

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

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

Nem lehet mivel a programok jelenleg képtelenek elemezni emberi szinten a forráskódokat. Így le kell futattniuk és meglátni mi lesz az eredménye. Nem lehet időhöz vagy lépésszámhoz kötni a ciklust, mivel mi a garanciája, hogy nem a következő másodprecben vagy lépésben végezne. Ezért ha egyszer elkapnak egy végtelen ciklust meg kell várni a végét, hogy biztosra meg tudják mondani, aminek az a következménye, hogy sosem fogják tudni eldönteni. Továbbá azzal, hogy eltudja dönteni, hogy valami a "végtelenségig" fut nem feltétlen vagyunk előrébb. Ez alatt azt értem, hogy, ha azt szeretnénk, hogy valami többször fusson le, akkor előfordúl, hogy egy végtelen ciklust használunk valamilyen kilépési feltétellel.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/valtozocsere.c

A változók felcserélését általában segédváltozó bevezetésével szokták megoldani, mivel a program futása során egy cpu mag egyszerre csak egy műveletet tud elvégezni. Így egyszerű adatszerkezetek felcserélése a következőéppen néz ki.

```
seged = a; //a értékét letároljuk
a = b; //a b-vel lesz egyenlő
b = seged; //végül b megkapja a letárolt éréket, ami a
```

Viszont, ha számokról van szó azokat exor, szorzás/osztás illetve összeadás/kivonás segítségével is fel lehet cserélni. Azonban ezzel a módszerrel az a baj, hogy nem feltétlen lehet minden esetben megvalósítani, mert például az összeadás/kivonás megvalósításához előbb össze kell adni a két számot egy változóba és ez ahhoz vezethet, hogy nem fog elférni a lefglalt memóriában, így adatvesztéssel szembesülünk. Viszont ha kis számokkal dolgozunk, akkor azt a memóriahelyet megspórolhatjuk, amit a segédváltozó foglalna le. Viszont ez eléggé minimális, mivel csak egyszer kell lefoglalni segédváltozónak helyet.

```
a += b; //segédváltozó nélküli csere
b = a-b;
a = a-b;
```

Habár nem használtunk segédváltozót az elvégzendő műveletek számát nem csökkentettük, habár a másolás lehet nagyobb erőforrásigényű mint az összeadás/kivonás. De így is ha a két szám összege nem fér bele egy int változóba, akkor nem az eredeti számokat fogjuk visszakapni.

Szorzás/osztás használatával is ugyanígy megoldható, csak ott hamarabb elérjük a lefoglalt memóriaméret határát. Exor-t használva pedig csak háromszor kell exorozni.

2.4. Labdapattogás

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

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

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/labda.c, https://github.com/DonatPataki/University/blob/master/prog1/labda_if.cpp

Maga a labdapattogtatás nem áll másból, minta konzolon belül egy karaktert pattogtatni. Ezt kellet megvalósítani először iffel majd anélkül. A konzolt fel lehet fogni úgy mint egy kétdimenziós síkot és x és y tengeleyeken mindig egyet lépünk. Amikkor elérjük a szélét akkor pedig irányvtáltunk. Azért hogy lássunk is valamit belőle rakunk bele sleep parancsot, ami kis időre "megfagyasztja" a programot.

```
    for ( ;; ) {

        getmaxyx ( ablak, my , mx );

        mvprintw ( y, x, "O" );
```

DRAFT

```
        refresh ();
        usleep ( 100000 );

        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;
}
```

Fent látató az ifes megoldás lényegi része. Az if nélkülit még tavaly oldottam meg, erre nem vagyok mostméóár olyan büszke, de működőképes csak nem olyan szép a kódja.

```
    int x_max=159, y_max = 44, x = 0, y = 0, x_, y_;
    while(1)
    {
      y_ = abs(y%y_max-y_max/2);
      for (int i = 0; i < y_; i++)
      {
        printf("\n");
      }
      x_ = abs(x%x_max-x_max/2);
      for (int i = 0; i < x_; i++)
      {
        printf(" ");
      }
      printf("*");
      for (int i = y_; i < 23; i++)
      {
        printf("\n");
      }
      x++;
      y++;
      sleep(1);

}
```

Itt az elképzelés az volt, hogy ha tudjuk előre, hogy mekkora helyen pattogtatjuk, akkor az abszolútérték fggvény tulajdonságit ki lehet használni. És működik is csak nekem valahogy nem tetszik annyira.

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: https://github.com/DonatPataki/University/blob/master/prog1/szohossz.c, https://github.com/-DonatPataki/University/blob/master/prog1/szohossz_BMIPS.cpp

Bekér egy char típusú változót és visszadja, hogy mi az a legkisebb bit amin még lehet tárolni azt a karaktert. Ezt úgy éri el, hogy a beolvasott karaktert számként kezeli és egy ciklusban minden alkalommal shifteli a biteket míg a változó értéke el nem éri a nullát. Így meg lehet kapni, hogy hány bitre volt szükség az adott karakter tárolására. De ettől még nem feltétlen azt kapjuk meg, hogy hány bit helyet foglal, hanem azt hogy mi az a legkisebb bitmennyiség amin még tárolható. Ez egyszerűen így oldható meg.

```
#include <stdio.h>

int main()
{
    char line[1];
    scanf("%1023[^\n]", line);
    int bitek = 0;
    int szam = line[0];
    printf("\nBeolvasott char értéke intben: ");
    printf("%d\n",szam);
    while (szam != 0)
    {
        szam = szam >> 1;
        bitek++;
    }
    printf("%d", bitek);
    printf(" biten tárolható\n");
    return 0;
}
```

Ez még nem a bogomipses megoldás, de simán át lehet alakítani, hogy működjön. Csak akkor most a while ciklusban fogunk shiftelni. Ezután csak a megfelelő helyre kell másolni a dolgokatés megakjuk ezt.

```
int
main (void)
{
    char line[1];
    scanf("%1023[^\n]", line);
    int bitek = 0;
    int szam = line[0];
    printf("\nBeolvasott char értéke intben: ");
    printf("%d\n",szam);

    unsigned long long int loops_per_sec = line[0];
    unsigned long long int ticks;
```

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

    while ((loops_per_sec >>= 1) != -1)
    {

  bitek++;
        if (loops_per_sec == 0)
        {
        printf("\nBeolvasott char értéke intben: ");
      printf("%d\n",szam);
    printf("%d", bitek);
    printf(" biten tárolható\n");

            return 0;
        }
    }

    printf ("failed\n");
    return -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: https://github.com/DonatPataki/University/blob/master/prog1/pagerank.c

Az alapötlet szerint egy oldal annál jobb minél kevesebb link mutat kifele belőle és minél több mutat rá. Mivel az emberek nyílván a jobb dolgokat többször osztják meg. Illetve azt is számításba veszi, hogy milyen fajta oldalak mutatnak másra. Ezért minden oldalnak kell adni egy kezdő értéket, ami minden egyes iteráció után változtatunk és azzal számoljuk tovább az oldalak viszonylagos rangját. Emiatt többször le kell futatni ugyanazt a képletet egészen addig, amíg az oldalak közötti változás elenyésző.

Ez a velóságban úgy néz, ki, hogy egy kétdimenziós tömbben letároljuk soronként a kifele mutató linkeket. Majd két tömbben tároljuk az oldalak rangját, hogy összetudjuk hasonlítani az eltérést, mivel nem szeretnénk a végtelenségig futattni. A képletben szerepel maga a pagerank értéke is, ezért van a PRv tömbben megadva egy kezdőérték. Fontos még megjegyezni, hogy a pagerank csak az oldalak egymáshoz viszonylagos értékét jelenti és nem feltétlen %-os értéket.

És a képlet, amivel kiszámoljuk így néz ki c-ben.

```
PR[i] += (L[i][j]*PRv[j]); //PR lesza számított pagerank, L a kimenő ↩
    linkek, PRv pedig az a pagerank amivel közben számol
```

Ezt elvégezzük minden oldalra és megnézzük mennyi az eltérés. Ha elég kicsi, akkor megállhatunk mivel elég pontos eredményt kaptunk. Ha nagy akkor PRv értékeit firssitjük és kezdjük előlről.

```
for (int i = 0; i < n; i++)
    {
        osszeg += (PRv[i]-PR[i])*(PRv[i]-PR[i]);
    }
return sqrt(osszeg);
```

A fenti kódcsipet talán kicsit értelmetlennek látszik, de azért van összeszorozva magával majd gyökvonás, hogy ne kapjunk negatív értéket. Ezt lehetne helyetesíteni azzal, hogy egyszerűen a kivonás után az abszolútértékét kérjük. Valamint a -lm kapcsolót kell használni fordítás során különben a gcc nem fogja megtaláltni vagy pedig c++ fordítót kell használni.

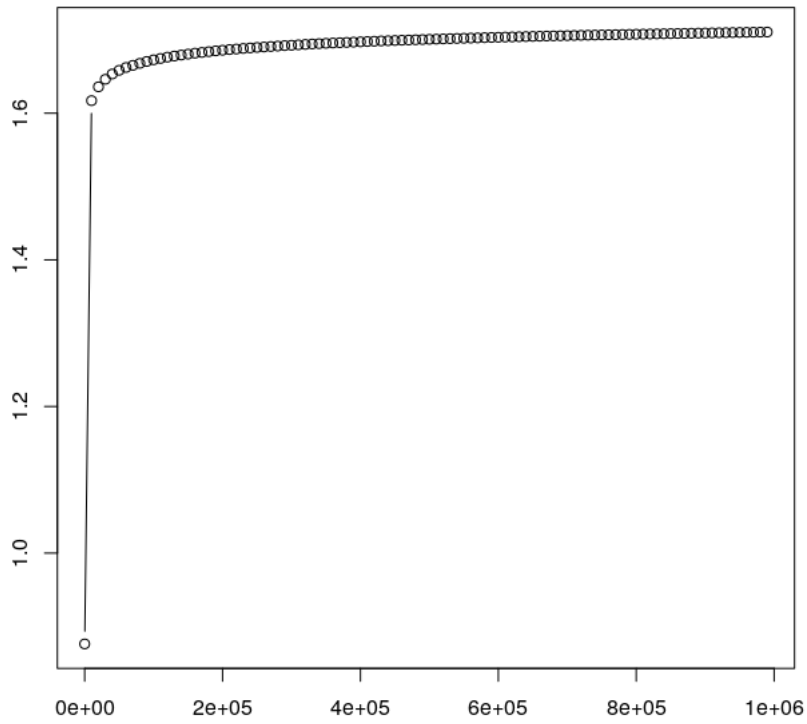
2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/stp.r

A Brun tétel kimondja, hogy az ikerprímek reciprokának összege egy konstanshoz konvergál. Lásd lenti ábra.



2.1. ábra. konstans közelítése

Az R szimuláció ennek megfelelően először megkeresni a prímeket, majd azon prímeket amiknek kettő a különbsége valamely prímmel, azaz egy ikerprímet.

```
primes = primes(x) #prímek leszűrése
diff = primes[2:length(primes)]-primes[1:length(primes)-1] #különbségek ↩
   megnézése
idx = which(diff==2) #ahol 2 az eltérés lementjük
```

Az így kapott kettővel kisebb prímet és annak kettővel nagyobb szintén prím számnak azaz az ikerprím tagjainak reciprokát vesszük és összeadjuk.

```
t1primes = primes[idx] #ikerprím első tagja
t2primes = primes[idx]+2 #ikerprí második tagja
rt1plust2 = 1/t1primes+1/t2primes #ikerprímek reciprokának összege
```

Ezután a függvény visszatéríti az összeget majd a program végén összefüzzük a megfelelő adatokat és kiíratjuk egy koordináta rendszerbe.

R kódot parancssorból r -f [fájl] paranccsal lehet futattni.

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://github.com/DonatPataki/University/blob/master/prog1/mh.r

Tutorom: Ratku Dániel

A paradoxon alapja, hogy az első választásnál 1/3 az esély, hogy jól választunk majd ajtó kinyílik bla bla bla és felkínálják egy második választás lehetőségét, ahol már 2/3 esélye annak, hogy a kívánt nyeremény van az ajtó mögött. Az R szimuláció pedig ezt próbálja szimulálni egy viszonylag nagy, véletlen generált választások sorozatával, ahol egy részében változtattak a második körben és ahol nem. Aminek a végeredménye az, hogy érdemes váltani az elején leírtak alapján.

A Monty Hall probléma amúgy egy veridical paradox, ami annyit tesz, hogy a megoldást be lehet bizonyítani, csak meglepő módon a megoldás nem az lesz, amit az ember gondolna.

Ez program szintjén úgy néz ki, hogy megadjuk, hány próbálkozást szeretnénk. Persze minél nagyobb annál pontosabb eredményt kapunk. Létrehozunk egy "játékos"-t aki elvégzi a doöntéseket és egy "műsorvezető"-t, aki megpróbálja elbizonytalaníteni játékosunkat.

Megnézzük elég sokszor úgy, hogy ha nem vált hányszor nyer és ha vált úgy hányszor.

```
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)]
}
```

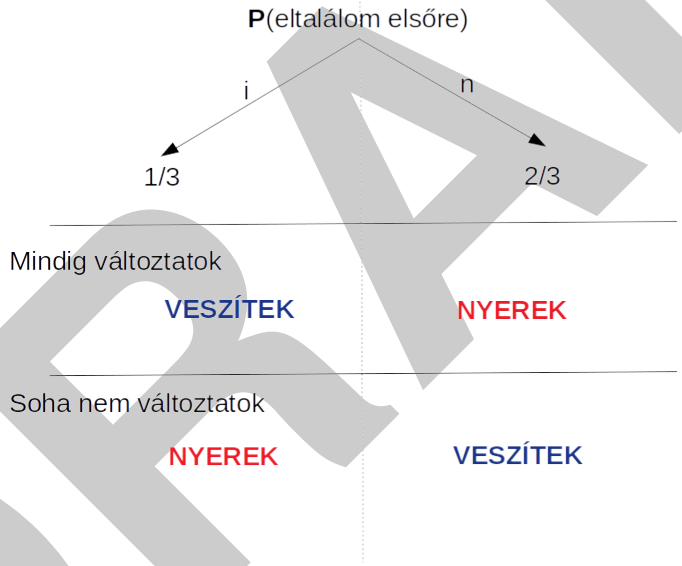
DRAFT

```
for (i in 1:kiserletek_szama) {

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

Persze a fenti kódcsipet között még kiírjuk az eredményt, igazából az a lényegi része a programnak.

Végül megkapjuk, hogy ha elsőre eltatláljuk és nyerni akarunk, akkor nem kell változtatnuk. Viszont annak az esélye, hogy ez bekövetkezzen 1/3. Azaz jobban járunk, hogy ha mindig változtatunk, mivel 2/3 esélye van annak, hogy nem találjuk el elsőre a kívánt ajtót.



2.2. ábra. forrás: https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcsan

DRAFT

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/decimal_unary.c

Az unáris számrendszer nem más mint vonalak egymás után, amiből annyi darab van, amekkora az ábrázolandó szám nagysága, ami csak természetes szám lehet. Ezt én a következőképpen oldottam meg.

```
    while(i != decimal)
    {
        std::cout << "1";
  i++;
}
```

Persze előtte még beolvasunk egy int típusú változót, hogy tudjuk, mihez sonlítani. De persze ha negatív számot adunk, akkor a program hibás eredményt ad.

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:

Első:

S (S $\rightarrow$ aXbc)

aXbc (Xb $\rightarrow$ bX)

abXc (Xc $\rightarrow$ Ybcc)

abYbcc (bY $\rightarrow$ Yb)

DRAFT

aYbbcc (aY → aa)

aabbcc

Második:

S (S → aXbc)

aXbc (Xb → bX)

abXc (Xc → Ybcc)

abYbcc (bY → Yb)

aYbbcc (aY -> aaX)

aaXbbcc (Xb → bX)

aabXbcc (Xb → bX)

aabbXcc (Xc → Ybcc)

aabbYbccc (bY → Yb)

aabYbbccc (bY → Yb)

aaYbbbccc (aY → aa)

aaabbbccc

A nyelvtanoknak amúgy haladóbb szinten jönnek be a programozásba, de én még egyszerű halandó vagyok hozzá és még nem is tanították, így bátorkodtam nem írni róla. De ha ez probléma jobban utánajárhatok a dolgoknak.

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/hivatkozas.c

Elég meglepő volt számomra, hogy c89 és c99 között elég sokminden változott. Néhány megszorítást elengedett a c99 például a for ciklusoknál.

```
int x; /* C89 féle for ciklus */
for(x = 0; x < 10; x++)

for(int i = 0; i < 10; i++) /* C99 féle for ciklus */
```

Néhány dolgot nem enged explicit módon. Megjelent a c++ stílusú egysoros komment stb.

```
//C++ stílusú komment, amivel a c89 compiler nem bír el
```

Azonkívűl hogy megtudtam, hogy a -std kapcsolóval lehet megadni, hogy melyik c compilert használja csak a change logot tudnám felsorolni, aminek nincs sok értelme, mert azt bárki meg tudja tenni.

DRAFT

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/realnumber.c

A lexer egy lexikális elemző, aminek pontos működését nem fontos tudni használatához. Ezért nem is próbáltam teljesen megérteni. A lényege annyi, hogy tokeneket lehet megadni, hogy mi alapján keressen és, hogy mit csináljon velük. Lehetséges karakterek felcserélése, összegzése stb. Mi most meg akarjuk számolni a valós számokat.

```
%{
#include <stdio.h>
int realnumbers = 0; // nullázuk a változót amive osszeszámoljuk
%}
digit [0-9] // megmondjuk neki milyenek a számok, azaz ezeket keresse
%%
{digit}*(\.{digit}+)? {++realnumbers;
    printf("[realnum=%s %f]", yytext, atof(yytext));} // kiírjuk a számokat ↩
        + növeljük a változó értékét, amiben ároljuk összesen hány db van
%%
int
main ()
{
 yylex ();
 printf("The number of real numbers is %d\n", realnumbers); // kiírjuk hány ↩
     természetes számot talált
 return 0;
}
```

Mivel itt csak számokról van szó elég ugye 0-tól 9-ig névni. Ez egy .l kiterjesztésű fájl lesz, amiből még generálni kell egy baromi hosszú c forráskódot. A fordítást a lex paranncsal tudjuk elvégezni, aminek nyílván telepítve kell lennie

A c forráskód lefodításához használni kell a -lfl és a program futattása kilistáza a számokat.

Futattás után pedig gyönörködhetünk, hogy az alapértelmezett bemenetről beolvasott szöveget elemzi és visszadja a valós számokat.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/l337d1c7.c

Lényegében ugyanaz a feladat mint az előző csak nem számokat keresünk, hanem a l33t-nek megfelelő karakterket cseréljük ki. Tehát nem 0-9-ig adjuk meg hanem a karakterket és azok "l33t párjait".

Ez program szinten úgy néz ki, hogy felsoroljuk a kívánt karakterket és azok leet párjait egy struktúrában. Ami ne kötelező, csak egyszerűbb így foglalkozni vele, mert így egy helyen vannak.

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

  {'a', {"4", "4", "@", "/-\\"}},
  {'b', {"b", "8", "|3", "|}"}},
```

A fenti mintára felsoroljuk a kívánt karakterket.

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

      if(l337d1c7[i].c == tolower(*yytext)) // kisbetűssé alakítjuk, mivel ↩
         csak a kisbetűket soroltuk fel
      {

        int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0)); // adunk hozzá egy ↩
           kis véletlent

          if(r<91) // random l33t karakterr cseréljük ki
          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) // ha nem talált, akkor az eredeti szöveget adjuk vissza
       printf("%c", *yytext);
```

Ez utóbbi kódcsipet végzi el a kicserélést. Persze megadhatuk volna a sktruktúrán belűl a nagybetűs karaktereket is, de azzal csak több helyet foglalnánk a memóriában. Feltéve hogy a l33t nagy és kisbetű ugynaz lenne. A fordítása/futattása az előző feladatéval megyegyezik.

Érdemes megjegyezni, hogy több fajta l33t stílus van és ezek közül random módon választ, szóval ha

ugyanazt a szöveget többször kapja meg nem mindig ugyazt a l33t szöveget adja vissza. Ezen kívűl a l33t-nek annyi haszna volt, hogy mivel egyes karaktereket felcseréltek megváltozott a szó karaktersorozata és amikor az eredeti szót ki akarták listázni a l33t változatát nem találta meg.

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

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

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

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

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

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

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

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

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

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

Megoldás forrása:

Megoldás videó:

Ha az átiárnyitott sigint sig_ign-re nem sig-ign akkor legyen átirányitva a jelkezelore.

for ciklusok kicsit hiányosak így nem fognak lefordúlni. Feltehetően azért mert még c99 előtt voltak írva és csak a for-os sort másolták át. Midenesetben ezeket lehet róluk elmondani:

Pre increment-es for ciklus 0-tól 4-ig.

Post increment-es for ciklus 0-tól 4-ig.

0-tól 4-ig, tömb iedik eleméhez hozzáír i+1-et. Vagy legalábbis ezt akarná csinálni.

For ciklus n-ig és a két mutatónak ugyanaz em az értéke.

Kiír két decimális számot, amit a függvények adnak vissza.

Két decimális számot ír ki, a értékét és azt az érétket amin a függvény dolgozott.

Két decimális számot ír ki, az első referencia.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

$(\forall x \exists y ((x<y)\wedge(y \text{ prím})\wedge(SSy \text{ prím})) ↩
   )$

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

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

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

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

Minden x, létezik y, x kisebb mint y vagy y prím

Minden x, létezik y, x kisebb mint y vagy y prím vagy SSy prím (Fogalmam sincs mi az az ssy mivel a forrásban nem írja és a google elég egyértelműen nem a megfelelő válazt adta)

Létezik y, minden x ahol x prím és y nagyobb

Létzeik y minden x y kisebb mint x és ebből nem következik hogy x prím

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciája

DRAFT

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

Mit vezetnek be a programba a következő nevek?

- ```
  int a;
  ```
- ```
  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: https://github.com/DonatPataki/University/blob/master/prog1/deklaracio.c

egész

mutató ami megkapta a memóriacímét

referencia a-ra

5 tagú egész tömb

5 tagú mutató tömb

egész típusú függvénymutató

függvény 2 int paraméterrel ami visszaad egy mutatót ami mutat egy függvényre ami intet ad vissza

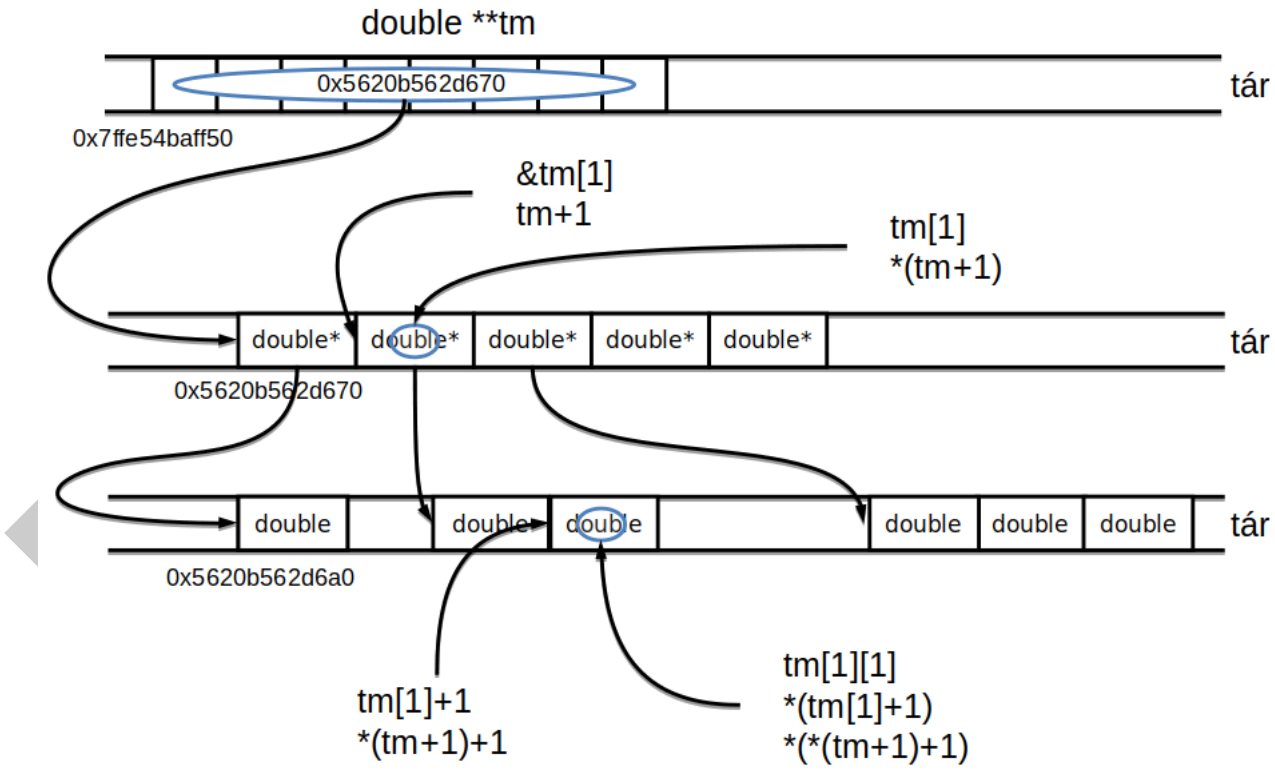
4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/tm.c



4.1. ábra. A double ** háromszögmátrix a memóriában

Egy alsó háromszög mátrix úgy néz ki, hogy az első sorban van egy használható érétke, majd minden azt követő sorban eggyel több, így egy háromszöget hoz létre, mivel a mátrix főátlój alatt helyezkednek eel

értékek. Ennek megfelelően először érdemes tudni, hogy hány soros lesz a mátrix és ezt rögötön az elején meg is adjuk.

Ezután a malloc függvényt meghíva lefoglaljuk a megfelelő memóriamértet és megkapjuk a rájuk mutató mutatót, amit a double **-al tudunk használni mivel ez egy mutató egy mutatóra. Ezután for ciklus segítségével kiírjuk a megfelelő sorokba a megfelelő mennyiségű változót, hogy egy alsó mátrixot kapjunk.

```
#include <stdio.h>
#include <stdlib.h>
int
main ()
{
    int nr = 5; //number of rows
    double **tm; //főszereplő

    printf("%p\n", &tm); //kiírjuk

    if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL) // ↩
       malloc-al foglalunk de ha null érték lenne, nr miatt azaz 0 sorost  ↩
       akarnánk létrehozni akkor hibás
    {
        return -1;
    }

    printf("%p\n", tm); //kiírjuk

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

    }

    printf("%p\n", tm[0]);    //ismét

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

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

    tm[3][0] = 42.0;
```

```
    (*(tm + 3))[1] = 43.0;  // 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");
    }

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

    free (tm);

    return 0;
}
```

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/titkosito.c

Ez a titosítás az exor logikai műveleten alapszik, ami lényegében nem csinál mást mint a kapott kulcs alapján és a beolvasott szöveg karaktereit bitenként összefésüli ott, ahol mindkét esetben a karakter bitsorozatában 1-es szerepel és az így kapott új bitsorozatnak megfelelő karakterek valamilyen olvashatatlan szöveget kell visszaadniuk.

Használata fordítás után ./titkosito [kulcs] <[bemenet] >[kimenet].

A kulcs maximális hossza 100 karakter lehet és az exor művelet 256-os karakterhosszokra van bontva.

```
#define MAX_KULCS 100 //program elején ezeket definiáltuk
#define BUFFER_MERET 256


while ((olvasott_bajtok = read (0, (void *) buffer, BUFFER_MERET))) // ↩
   titkosítás ciklusa
```

Maga a titkosítás pedig itt található. A buffer tartalmát a for ciklussal változtatuk, míg a kulcs méretet kicsit megfontotltan mert csak azt tudjuk, hogy 100-nál kisebb. Ezért vesszük az eredeti értékét és hozzáadunk egyet. És ennek vesszük a modulóját a kulcs hosszával. Így akármilyen hosszú kulccsal működni fog.

```
buffer[i] = buffer[i] ^ kulcs[kulcs_index]; //xor művelettel  ↩
   olvashatatlanná tesszük
kulcs_index = (kulcs_index + 1) % kulcs_meret; //léptetjük az indexet
```

Ezután a while ciklusban használjuk a write rendszerhívást és kiírjuk a buffert, majd ha tud még olvasni akkor folytatódik a ciklus.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/javatitkosito.java

Az előző c verzió java megvalósítása, bár van pár lényegi eltérés. Ugyanúgy exoron alapszik.

A buffer ugyanúgy 256 bájt, azaz 256-os karakterszakaszonként fog végrehajtódni, ugyanúgy mint az előbb, viszont itt nincs megadva a kulcsnak limit.

```
byte [] buffer = new byte[256];
```

Valamint itt a titkosító algoritmus egy külön osztályban található, amit a main function-ből hívunk meg egy try catch között, ha esetleg valami rosszúl sülne el. Ami semmi extra csak dobunk egy hibát ha rosszúl adnánk meg a paramétert, ami az IOException

```
try {

  new ExorTitkosító(args[0], System.in, System.out);

} catch(java.io.IOException e) {

  e.printStackTrace();
}
```

Olvasni a standard inputról olvasunk és oda is írunk. Maga a ciklus ami pedig a titkosítást, olvaást és írést végzi ugyanaz.

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

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

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

            }

            kimenőCsatorna.write(buffer, 0, olvasottBájtok);

}
```

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/exortoro.c

Az exor törő müködése arra alapszik, hogy ha a már titkosított szöveget ugyanazzal a kulccsal mégegyszer exorozuk, akkor visszakapjuk az eredeti szöveget. Ez mind szép és jó csak annyi a baj, hogy nem tudjuk mi a kulcs és ezért minden lehetséges variációt létre kell hozni. Erre a célra egymásba ágyazot for ciklusokat használunk. Ebben a példában csak a-z-ig nézi a karaktereket ezért ha mást is szeretnénk hogy használjon a kulcsok előállítása alatt érdemes egy vektorba összeszedni azokat a karakterket és a for ciklusokkal csak a vektor elemeit változtani.

```
  for (int ii = 'a'; ii <= 'z'; ++ii) //a char típus között konverzió lesz, ↩
      bár így az ékezetes karakterek kimaradnak
    for (int ji = 'a'; ji <= 'z'; ++ji)
      for (int ki = 'a'; ki <= 'z'; ++ki)
    for (int li = 'a'; li <= 'z'; ++li)
      for (int mi = 'a'; mi <= 'z'; ++mi)
          {
            kulcs[0] = ii;
            kulcs[1] = ji;
            kulcs[2] = ki;
            kulcs[3] = li;
            kulcs[4] = mi;

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

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

Most hogy minden lehetséges kulcsot előállítottunk egy "biztos" a megfelelő kulcs (ami csak 5 karakter hosszú és csak a-z-ig tartalmaz karakterket) így valahogy el kell tudnunk dönteni, hogy melyik az. Ezt úgy oldjuk meg, hogy előállítás közben mindig exorozunk és ha az áltag szóhossz 6 és 9 között található és tartalmaz hogy, nem, az, ha szavakat, akkor az eredeti szöveget kaptuk vissza.

```
double szohossz = atlagos_szohossz (titkos, titkos_meret);

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

Amúgy itt egy hivási lánc is megtekinthető, de gondolom ezt nem nehéz köveetni úgyhogy be se rakom.

Használata fordítás után ./exortoro <[bemenet].

4.5. Neurális OR, AND és EXOR kapu

R

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

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/nn.r

Neurális háló ahol and, or és exor kapuk vannak implementálva. Az and és or működése magától értetődő. Az előbbinél ha mindkettő 1 akkor lesz az eredményük 1. Az utóbbinál pedig ha az egyik 1 akkor lesz az eredményük 1. Az exor pedig nem jelent mást minthogy a két érték nem egyezik meg, tehát 0 és 1 esetén lesz az eredmény 1 és ez fordított esetben is igaz.

Az összes kaput az alábbi mintára fel lehet építeni.

```
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) # a fentni értékeket ebbe belerakjuk

nn.or <- neuralnet(OR~a1+a2, # formula
       or.data, # adatok amin dolgozik
       hidden=0, # hány rejtett réteg legyen
       linear.output=FALSE, # akarjuk-e használni act.fct-t itt most igen
       stepmax = 1e+07, # max lépésszám
       threshold = 0.000001) # treshold

plot(nn.or) # nurális háló plotolása

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

Itt megtalálható amúgy egy angol nyelvű dokumentáció, ami alapján készült is a fenti kommentelgetés, de valahogy magyarúl furán hangzottak ezért csak a lényeget írtam ki kommentben.

A neurális hálóról még érdemes annyit megjegyezni, hogy 3 részre lehet osztani. Van az input layer, ahol megkapja az adatokat. Ezt követi egy vagy több hidden layer, ahol a varázslat történik. Majd végül jön az output layer, ahol ha minden jól működik a megfelelő érétket kapjuk vissza.

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/ql.hpp

A perceptron nem más mint egy algoritmus model, ami az emberi agy működését próbálja utánozni. Hasonló a neurális hálóhoz azonban van pár különbség. Ugyanúgy input után elkezd varázsolni és jobb esetben megfelelő mintavétel után helyes eredményt ad vissza. Azonban a közbelső értékeknek van súlya amit még adott konstansokkal is ki lehet egészíteni. Az így kapott súlyokat összeadja és ha ez elér egy bizonyos szintet, akkor a program adott része aktiválódik. És egy a lineáris folyamat ismétlődik amíg el nem jut a válaszig. Valamint ez egy egyréteges megvalósítás.

DRAFT

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/mandelpngt.cpp

A mandelbrot halmazt úgy tudjuk megkpani, hogy a komplex számsíkon azon komplex számokat akarjuk ábrázolni, amely komplex számok hossza egy adott szám. Így ennek megfelelően előbb létre kell hozni egy adott nagyságú koordináta rendszert (vagy akármit amivel lehet azt reprezentálni) és annak origójából azon pontokat amelyekre az adott képlet alapján teljesül a kritérium azokat kiszinezünk a és a többit pedig nem változtatjuk.

Az alábbi kódcsipetben található a maga a forrás lényege.

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


    float dx = (b - a) / szelesseg; //ezzel keressük meg az "x" tengely ↩
       közepét
    float dy = (d - c) / magassag; //ez pedig az "y" tengely közepe
    float reC, imC, reZ, imZ, ujreZ, ujimZ;

    int iteracio = 0;

    for (int j = 0; j < magassag; ++j) // sor koordinátái
    {

        for (int k = 0; k < szelesseg; ++k) // oszlop koordinátái
        {
            reC = a + k * dx;
            imC = d - j * dy;

            reZ = 0;
            imZ = 0;
            iteracio = 0;
```

DRAFT

```
            while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)  ↩
               //adott képletes rész
            {

                ujreZ = reZ * reZ - imZ * imZ + reC;
                ujimZ = 2 * reZ * imZ + imC;
                reZ = ujreZ;
                imZ = ujimZ;

                ++iteracio;

            }

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

Ezután lementjük a pixelek mit kaptak, azaz annak megfelelő színűk lesz és majd kirajzolódik szép 600x600-as képben a mandelbrot halmaz.

Ezen kívűl még annyit tudok hozzáfűzni, hogy elég sok forrás/videó található ebből a témából, amit nem értek mivel annyira nem különleges eset számomra a mandelbrot halmaz. Mivel nem történik más mint hogy a komplex számsíkon ábrázoljuk azokat a komplex számok részhalmazát amire teljesűl a képletben megadott kritérium.

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/mandelcomp.cpp

Ugyanaz mint az előző feladat annyi eltéréssel, hogy használjuk a complex header alapján definiált típust. Egyébként a complex headerrel meglepő módon komplex számokat lehet reprezentálni float, double és long double formában.

Itt most simán a main függvénybe írtuk meg a mandelbrotos halmazt, ahol van valamennyni hibakezelés, ha rosszúl kapná mega paramétereket. Csak nem értem minek adtunk egyes elemeknek értékeket, ha pontosan 10 sorral lentebb úgyis azt adjuk hozzá, amit mi adtunk parancssori argumentuként. De mindegy. Ha esetleg nem jönne össze az a 9 argumentum akkor pedig kiírjuk, hogy hogyan is kell elindítani a programot.

```
int szelesseg = 1920; //ettől
int magassag = 1080;
int iteraciosHatar = 255;
double a = -1.9;
double b = 0.7;
double c = -1.3;
double d = 1.3; //eddig nem sok értelme van mert ezeket az értékeket  ↩
   úgyis felűlíruk

if ( argc == 9 ) //ha elegendő az argumentum
```

DRAFT

```
    {
      szelesseg = atoi ( argv[2] ); //akkor felülírjuk amit a felhasználó ↩
         adott
      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 //ha nem akkor szólunk hogy hogyan kéne elindítani
    {
      std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d ↩
         " << std::endl;
      return -1;
}
```

Most hogy ezen túl vagyunk akkor igazából elég hasonló a program.

```
  for ( int j = 0; j < magassag; ++j ) //sor oszlop iterációi
    {

      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 ) //ez ↩
             kicsit rövidebb lett
            {
              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;
}
```

5.3. Biomorfok

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

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/biomorf.cpp

Az a kevés információ alapján, amit találni lehet róla (feltéve hogy megfelelően keresetem rá) ez nem más mint egy nem megfelelően működő mandelbrot halmaz. Mivel van benne egy konstans tag. Feltalálójának amúgy az volt az alapvető elképzelése, hogy vizualizáció során gyorsabb fejlődést lehet elérni mint a nyers adatok bámulásával ezért például DNS részeket próbált vizualizálni. Gondolom innen jött később a biomorfok név.

Ennek az eleje eléggé hasonlít az előző meegoldáshoz. Csak most több értékkel fogunk dolgozni.

```
    int szelesseg = 1920; //adunk kezdőértéket de nem látom értelmét
    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] ); //beolvasott értékeket rendejük hozzá
        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 //ha nem jött össze a 12 argumentum akkor szólunk hogy hogyan kéne
    {
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c ↩
            d reC imC R" << std::endl;
        return -1;
}
```

Ezután szintén elkezdjük számolni.

```
    for ( int y = 0; y < magassag; ++y ) //sor
    {

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

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

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

                z_n = std::pow(z_n, 3) + cc;
                if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
                {
                    iteracio = i;
                    break;
                }
            }

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

Persze még adtunk hozzá egy kis apróságt is, ami minden iteráció után kiírja hogy hány %-nál tartunk ezután mentjük a képet és még azt is kiírjuk, hogy végzett.

```
      int szazalek = ( double ) y / ( double ) magassag * 100.0; //%  ↩
         számítás
        std::cout << "\r" << szazalek << "%" << std::flush; //kiírjuk hol  ↩
           járunk
    }

    kep.write ( argv[1] ); //mentjük
    std::cout << "\r" << argv[1] << " mentve." << std::endl; //jelezük hogy ↩
        vége

}
```

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/mandel.cu

Szokásos mandelbrot halmaz azzal a különbséggel, hogy most az nvidia által fejlesztett cuda magokat használjuk. Ezek segítségével a feladatokat lehet párhuzamosítani, viszont cuda csak az nvidia grafikus kártyákban található. Használható c, c++ és fortran nyelveken. Leegyszerűsítve a cuda magok olyanok mint sok kis teljesítményű processzor.

Ez a kód egyértelműen az elsőből lett átdolgozva és így már értelmet nyert az, hogy miért nézzük mennyi ideig tart lefutattni. Mivel ez egy jól párhuzamosítható folyamat így egyértelműen kijön majd a lineáris és párhuzamos folyamatvégzés időbeli különbsége.

Itt is külön eljárásban van a mandelbrotos halmaz létrehozása.

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

  int *device_kepadat;
  cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int)); // ↩
     gpu-n foglalunk helyet

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

  dim3 grid (MERET / 10, MERET / 10); //dim3 cuda int vector típús, aminek  ↩
     nem muszáj mindhárom elemét megadni
  dim3 tgrid (10, 10);
  mandelkernel <<< grid, tgrid >>> (device_kepadat); //szétosztjuk a  ↩
     feladatot

  cudaMemcpy (kepadat, device_kepadat,
        MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost); //kivesszük  ↩
           a gpu memóriájából
  cudaFree (device_kepadat); //szabadítunk

}
```

Ezután elkezdünk számolni.

```
__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); //meghívja a függvényt ami  ↩
     kiszámolja

}
```

A lenti kódrész majdnem egy az egyben megegyezik az első feladatban látottal, csak most ez nem lineáris módon fut, ezért csak a két for ciklustól kellet megszabadúlni.

```

__device__ int
mandel (int k, int j)
{
```

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


  float dx = (b - a) / szelesseg;
  float dy = (d - c) / magassag;
  float reC, imC, reZ, imZ, ujreZ, ujimZ;

  int iteracio = 0;

  reC = a + k * dx;
  imC = d - j * dy;
  reZ = 0.0;
  imZ = 0.0;
  iteracio = 0;
  while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
    {

      ujreZ = reZ * reZ - imZ * imZ + reC;
      ujimZ = 2 * reZ * imZ + imC;
      reZ = ujreZ;
      imZ = ujimZ;

      ++iteracio;

    }
  return iteracio;
}
```

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 iteréció bejárta z_n komplex számokat!

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/tree/master/prog1/Qt_mandelbrot

Még mindig mandelbrot most egy egyszerű kép helyett azonban lehet nagyítani és így a végtelenségig rajzolva a mandelbrot halmazt. Elég sok videó van erről, amik úgy néznek ki mint egy megkérdőjelezhető tudati állapotban lévő ember álma. De nyílván az én hozzáállásom volt rossz, mert azt hittem azért van ennyi mandelbrotos feladat mert ez annyira fontos és véletlen sem a változásokat egyszerű így észrevenni.

De inkább nézzük a kódot.

```
    QApplication a(argc, argv);
    // További adatokat olvashatsz le innen:
    // http://www.tankonyvtar.hu/informatika/javat-tanitok-2-3-080904
    FrakAblak w1,
    w2(-.08292191725019529, -.082921917244591272,
```

```
        -.9662079988595939, -.9662079988551173, 600, 3000),
    w3(-.08292191724880625, -.0829219172470933,
        -.9662079988581493, -.9662079988563615, 600, 4000),
    w4(.14388310361318304, .14388310362702217,
        .6523089200729396, .6523089200854384, 600, 38656);
    w1.show();
    w2.show();
    w3.show();
    w4.show();

return a.exec();
```

Lényegében csak ablakokat kérünk. Egy is elég lenne, nyílván azért van így mert vagy valami története vagy nem végleges verzió volt a forrás.

A lenti kódban nem történik sok izgalmas dolog, csak előkészítjük a terepet.

```
FrakAblak::FrakAblak(double a, double b, double c, double d,
                     int szelesseg, int iteraciosHatar, QWidget *parent)
                         : QMainWindow(parent)
{
    setWindowTitle("Mandelbrot halmaz");

    int magassag = (int)(szelesseg * ((d-c)/(b-a)));

    setFixedSize(QSize(szelesseg, magassag));
    fraktal= new QImage(szelesseg, magassag, QImage::Format_RGB32);

    mandelbrot = new FrakSzal(a, b, c, d, szelesseg, magassag,  ↩
       iteraciosHatar, this);
    mandelbrot->start(); //ez utóbbi két sor elég lényeges

}
```

És ez teszi lehetővé a nagyítást

```
void FrakAblak::paintEvent(QPaintEvent*) {
    QPainter qpainter(this);
    qpainter.drawImage(0, 0, *fraktal);
    qpainter.end();
}
```

A frakszal.cpp-n belűl ott van maga a számolás, de a kódot nem raknám be teljesen, mert már ennyi mandelbrotos feladat után rosszúl leszek. Úgyhogy csak a lénnyeget említeném benne.

```
FrakSzal::FrakSzal(double a, double b, double c, double d,
                   int szelesseg, int magassag, int iteraciosHatar,  ↩
                      FrakAblak *frakAblak)
{
    this->a = a;
    this->b = b;
    this->c = c;
```

```
    this->d = d;
    this->szelesseg = szelesseg;
    this->iteraciosHatar = iteraciosHatar;
    this->frakAblak = frakAblak;
    this->magassag = magassag;

    egySor = new int[szelesseg];
}
```

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/mandelbrot.java

Végre az utolsó mandelbrotos feladat. Szóval minden mellébeszélés nélkül essünk rajta túl minél hamarabb.

A main függvényben nem olyan nehéz elveszni, mert rögtön megyünk is tovább adott paraméterekkel.

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

Ezután az elábbi programrész hívódik meg

```
    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) { //ez ↩
               figyeli az egérgomb lenyomást

              x = m.getX();
                y = m.getY();
                mx = 0;
                my = 0;
                repaint();
            }

            public void mouseReleased(java.awt.event.MouseEvent m) { // ↩
               újraszámol
                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, //újraszámolt értékekkel új példányt hoz létre
                        MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
                        MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
                        MandelbrotHalmazNagyító.this.d-y*dy,
                        600,
                        MandelbrotHalmazNagyító.this.iterációsHatár);
            }
        });

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

Továbbá még van egy rész, ahol a pillanatképek készülnek png formában, de azt nem tartom olyan izgalmasnak. Ennyi mandelbrotos feladat után nem igazán tudok újat hozzáfűzni, esetleg annyit, hogy maga a java elég olvasható különösebb tudás nélkül.

DRAFT

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/polar.java, https://github.com/-DonatPataki/University/blob/master/prog1/polar.cpp

A program lényege az lenne, hogy a polár transzformációs algoritmussal számol, de mivel a feladatban leírtak szerint ez teljesen lényegtelen, ezért ezt nem kell elmagyarázni. Így a programhoz még annyit tudok hozzáfűzni, hogy tízszer fogja kiszámolni és annak megfelelően, hogy van-e tárolt tag visszatérítésnél azt is figyelembe veszi.

A java valósítássasl kezdve. Létrehoz egy példányt a PolarGenerator osztályból és tízszer meghívja a kovetkezo nevű függvényét és kiírja a standard outputra az eredményét.

```
  public static void main(String[] args) (
    PolarGenerator g = new PolarGenerator();
    for (int i = 0; i < 10; ++i) (
      System.out.println(g.kovetkezo());
      )
)
```

A program lelke ebben a példában maga a kovetkezo függvény, de mivel a matek része felesleges (legalábbis ez lett írva) így a forrást csak itt hagyom.

```
  public double kovetkezo() (
    if (nincsTarolt) (
      double u1, u2, v1, v2, w;
      do (
        u1 = Math.random();
        u2 = Math.random();
        v1 = 2 * u1 - 1;
```

DRAFT

```
        v2 = 2 * u2 - 1;
        w = v1 * v1 + v2 * v2;
        ) while (w > 1);
      double r = Math.sqrt((-2 * Math.log(w)) / w);
      tarolt = r * v2;
      nincsTarolt = !nincsTarolt; //ez egy szép egyszerű megoldás

      return r * v1;
      ) else (
      nincsTarolt = !nincsTarolt; // ami csak boolal működik
      retrun tarolt;
      )
)
```

A c++ forrást nem mutatom be mert teljesen ugyanaz és most tényleg.

A feladat lényege amúgy az lett volna, hogy a sun programozók (akiket nyílván csak én nem ismerek) is így oldották meg. Feltéve hogy ők minden bizonnyal nagyon tapasztalt programozók és ezzel a megoldással álltak ők is elő, így nyílván nekünk is lazán fog menni a programozás.

6.2. LZW

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/z3a7.cpp

A binfa egy olyan adatszerkezet, aminek csomópontjainak legfeljebb két gyereke lehet. Ezeket jobb és bal gyermekként szokás hívni. Valamint a gyerekek is csomópontként működnek. Azaz a gyerekeknek is lehetnek további gyerekei, így elég komplex fákat lehet kapni végeredményül. Ezért a binfákat érdemes rekurziv módon bejárni. Továbbá attól függően, hogy milyen a gyerekek elolszlása lehet egy binfa kiegyensúlyozott, tökéletesn kiegyensúsolyzott stb. Ezek a tulajdonságog többnyire csak a jobb és bal oldal viszonyát nézik egymáshoz. A program nem csinál mást mint a megfelelő bemenet megkapása után egy fájlba felépít egy binfát.

```
    if (argc != 4) //megnézi van-e elég argumentum
    {
        usage ();
        return -1;
    }

    char *inFile = *++argv; //lementi a bemenet nevét

    if (*((*++argv) + 1) != 'o') //mi a kapcsoló
    {
        usage ();
        return -2;
    }
```

```
    std::fstream beFile (inFile, std::ios_base::in); //fájl beolvasása

    if (!beFile) //ha nincs szól hogy gáz van
    {
        std::cout << inFile << " nem letezik..." << std::endl;
        usage ();
        return -3;
    }

    std::fstream kiFile (*++argv, std::ios_base::out); //létrehoza a ↩
       kimenetet

    unsigned char b;    // ide olvassik majd a bejövő fájl bájtjait
    LZWBinFa binFa;   // s nyomjuk majd be az LZW fa objektumunkba

    while (beFile.read ((char *) &b, sizeof (unsigned char)))
        if (b == 0x0a)
            break;

    bool kommentben = false;

    while (beFile.read ((char *) &b, sizeof (unsigned char))) //maga az ↩
       építést végző loop
    {

        if (b == 0x3e) //ezek itt most lényegtelenek
        {     // > karakter
            kommentben = true;
            continue;
        }

        if (b == 0x0a)
        {     // újsor
            kommentben = false;
            continue;
        }

        if (kommentben)
            continue;

        if (b == 0x4e)     // N betű
            continue;

        for (int i = 0; i < 8; ++i) //a char 8 bite alapján eldönti milyet ↩
           rak a fába
        {
            if (b & 0x80)
                binFa << '1';
            else
                binFa << '0';
```

```
            b <<= 1;
        }

}
```

A << operátor túl van terhelve ezért tudjuk annak használatával belepakolni a fába. Ez így van implementálva

```
    void operator<< (char b)
    {
        if (b == '0')
        {
            if (!fa->nullasGyermek ())
            {
                Csomopont *uj = new Csomopont ('0');
                fa->ujNullasGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->nullasGyermek ();
            }
        }
        else
        {
            if (!fa->egyesGyermek ())
            {
                Csomopont *uj = new Csomopont ('1');
                fa->ujEgyesGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->egyesGyermek ();
            }
        }
}
```

A fenti kódcsipetből jól látszik, hogy igazából úgy működik, hogy ha nincs olyan gyerek amit keres akkor csinnál egyet, ha pedig van akkor rálép.

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/tree/master/prog1/bejaras

A binfáknak három féle bejárási módja van. Ezek az inorder, preorder és postorder nevet viselik. A preorder fabejárás során a gyökérből indulunk ki és előbb azok bal majd jobb oldali gyerekeit iratjuk ki. Inorder és

postorder során az utolsó bal oldali gyerekből indulunk ki és ha inorder kiiratás szeretnénk akkor azután a gyökeret, majd a jobb oldali gyereket írjuk ki. Ha pedig ezt a sorrendet felcseréljük, azaz a bal gyerek után a jobbat majd a gyökeret írjuk ki, akkor postorder bejárást kapunk. De igazából a legegyszerűbb úgy megjegyezni, hogy a gyökérrel mikor foglalkozunk a két gyerekhez képest. Ezeket a bejárásokat célszerű rekurzívan kezelni, mivel ezek a bejárások ugyanazt csinálják és így elég egyszer megírni és csak újra meghívni a binfa egy részfájára.

Szóval a háromféle bejárás között csak annyi lesz a különbség, hogy milyen sorrendben kérjük le a 3 komponenst

```
void LZWBinaryTree::PreOrder(BinaryTreeNode<char> *node) {
  if (node == nullptr) {
    return;
  }

  traverse_function(node, traverse_depth);

  ++traverse_depth;
  PreOrder(node->get_right_child());
  PreOrder(node->get_left_child());
  --traverse_depth;
}

void LZWBinaryTree::InOrder(BinaryTreeNode<char> *node) {
  if (node == nullptr) {
    return;
  }

  ++traverse_depth;
  InOrder(node->get_right_child());

  traverse_function(node, traverse_depth);

  InOrder(node->get_left_child());
  --traverse_depth;
}

void LZWBinaryTree::PostOrder(BinaryTreeNode<char> *node) {
  if (node == nullptr) {
    return;
  }

  ++traverse_depth;
  PostOrder(node->get_right_child());
  PostOrder(node->get_left_child());
  --traverse_depth;

  traverse_function(node, traverse_depth);
}
```

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/z3a7.cpp

Mivel ez már alapból c++ kód volt, ezért az elején el van magyarázva az építős része. Most pedig megnézzük, hogy hogyan van beágyazva a gyökérbe.

```
class LZWBinFa
{
public:
    LZWBinFa ():fa (&gyoker)
    {
    }
    ~LZWBinFa ()
    {
        szabadit (gyoker.egyesGyermek ());
        szabadit (gyoker.nullasGyermek ());
    }



    friend std::ostream & operator<< (std::ostream & os, LZWBinFa & bf)
    {
        bf.kiir (os);
        return os;
    }
    void kiir (std::ostream & os)
    {
        melyseg = 0;
        kiir (&gyoker, os);
    }

private: //ez itt a lényeg
    class Csomopont
    {
    public:
        Csomopont (char b = '/'):betu (b), balNulla (0), jobbEgy (0)
        {
        };
        ~Csomopont ()
        {
        };
        Csomopont *nullasGyermek () const
        {
            return balNulla;
        }
```

```
        Csomopont *egyesGyermek () const
        {
            return jobbEgy;
        }
        void ujNullasGyermek (Csomopont * gy)
        {
            balNulla = gy;
        }
        void ujEgyesGyermek (Csomopont * gy)
        {
            jobbEgy = gy;
        }
        char getBetu () const
        {
            return betu;
        }

    private:
        char betu;
        Csomopont *balNulla;
        Csomopont *jobbEgy;
        Csomopont (const Csomopont &); //másoló konstruktor és egyebek de ezek nem fognak működni mert nincsenek implemetálva + private részben vannak
        Csomopont & operator= (const Csomopont &);
    };

    Csomopont *fa;
    LZWBinFa (const LZWBinFa &);
    LZWBinFa & operator= (const LZWBinFa &);
```

Így hogy a gyökér és a gyerek egy classon belűl van teljesen normálisan el lehet érni a gyereket és a gyökeret is. Nem kell olyan csúnya megoldásokhoz folyamodni majd, mint mutatók használata csak ezért az egyszerű feladatért.

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: https://github.com/DonatPataki/University/blob/master/prog1/bintree.cpp

A node és a tree osztályok teljesen el vannak választva egymástól, így a fa működését úgy tudjuk, hogy megvalósítani, hogy a gyökér tagod mutatóként továbbadjuk a csomópontnak és így épitjük fel a fát minden egyes új csomóponttal, aminek az a nehézsége vagy különbsége, hogy ha a gyökeret akarjuk elérni, azt csak a mutatón keresztűl tudjuk megtenni.

Később valószínűleg nem ezt a kódot fogom használni, csak tetszett, hogy milyen élesen látszik, hogy két külön osztályról van szó.

```
class Node{
    public:
    char content;
    Node *child0, *child1;
    Node(char c):content(c), child0(0), child1(0){}
    Node(const Node& node):content(node.content){
        if(node.child0!=0) child0=new Node(*(node.child0)); else this-> ↩
           child0=0;
        if(node.child1!=0) child1=new Node(*(node.child1)); else this-> ↩
           child1=0;}
    void makeChild(char type){
        if(type=='0') child0=new Node(type);
        else child1=new Node(type);}
};

class Tree{
    public:
    Node *root, *actual; //ezeket használva tudjuk majd bejárni a fát
    int maxdepth, avg_c, avg_sum;
    double avg, deviation, deviation_sum;
    Tree (){
        root = new Node('/');
        actual = root;
        maxdepth =  avg_c = avg_sum = 0;
        avg = deviation = deviation_sum = 0.0;}
    Tree(const Tree& t): maxdepth(t.maxdepth), avg_c(t.avg_c), avg_sum(t. ↩
       avg_sum), avg(t.avg), deviation(t.deviation), deviation_sum(t. ↩
       deviation_sum) {
        root = new Node(*(t.root));
        actual = root;}
    Tree(Tree&& t): maxdepth(t.maxdepth), avg_c(t.avg_c), avg_sum(t.avg_sum ↩
       ), avg(t.avg), deviation(t.deviation), deviation_sum(t.deviation_sum ↩
       ) {
        actual = root = t.root;
        t.root = 0;}
    Tree& operator= (Tree&& t){
        avg_c=(t.avg_c); avg_sum=(t.avg_sum); avg=(t.avg); deviation=(t. ↩
           deviation); deviation_sum=(t.deviation_sum);
        freeNodesFrom(root);
        root = t.root;
        actual = t.actual;
        t.root=0;}
    Tree& operator= (const Tree& t) {
        avg_c=(t.avg_c); avg_sum=(t.avg_sum); avg=(t.avg); deviation=(t. ↩
           deviation); deviation_sum=(t.deviation_sum);
        freeNodesFrom(root);
        root = new Node(*(t.root));
        actual = t.actual;}
    ~Tree(){freeNodesFrom(root);}
```

```
    void freeNodesFrom(Node *node){
      if(node!=0){
        freeNodesFrom(node->child0);
        freeNodesFrom(node->child1);
        delete node;}}
}
};
```

A fenti kódból ki lett törölve pár rész, hogy ne legyen zavaró annyira a sűrű blokk, de jól látszik hogy teljesen el vannak szeparálva és pointerek segítségével meg lehet oldani gond nélkűl így is.

Egyelőre marad ez a kód, de majd ahogy lesz időm lecserélem egy jobban olvashatóra.

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/tree/master/prog1/mozgato

A mozgató konstruktor lehetővé teszi, hogy egy object érétkeit egy másikba helyezük át. Azaz jelen esetben ha binfás példára nézzük, akkor a mozgató konstruktor segítségével lehetséges, hogy egy már kész binfát egy másikba tegyünk, azonban így az amiből mozgattuk az értékeket üresen marad, mivel nem történik más mint, hogy a az értékeket más helyre pakoljuk. Előnye a másolással szenben azonban az, hogy a mozgatás kevesebb erőforrást vesz igénybe mint egy teljes másolatot készíteni az egészről. A c++11 óta van alapértelmezett mozgató konstruktor is, ami hasonlóan a másolóhoz nem mindig működik tökéletesen, ezért lehetőség van annak működését deklarálni is. A mozgató értékadás hasonló a mozgató konsteruktorhoz, azzal a különbséggel, hogy az utóbbi akkor kerűl meghívásra, ha deklarációkor adjuk meg az értéket, míg a mozgató értékadás akkor, ha már egy object létrejött és annak értékét később akarjuk egy másikból áthelyezni.

```
BT::BT(BT&& moved) :
  root(moved.root),
  currentNode(moved.root),
  treeHeight(moved.treeHeight) //mozgató konstruktor
{
  moved.root = nullptr;

BT& BT::operator= (BT&& moved) //mozgató értékadás
{
  if (this == &moved)
    return *this;

  delete root;

  root = moved.root;
  currentNode = moved.root;
  treeHeight = moved.treeHeight;
```

DRAFT

```
  moved.root = nullptr;

  return *this;
}
```

DRAFT

7. fejezet

Helló, Conway!

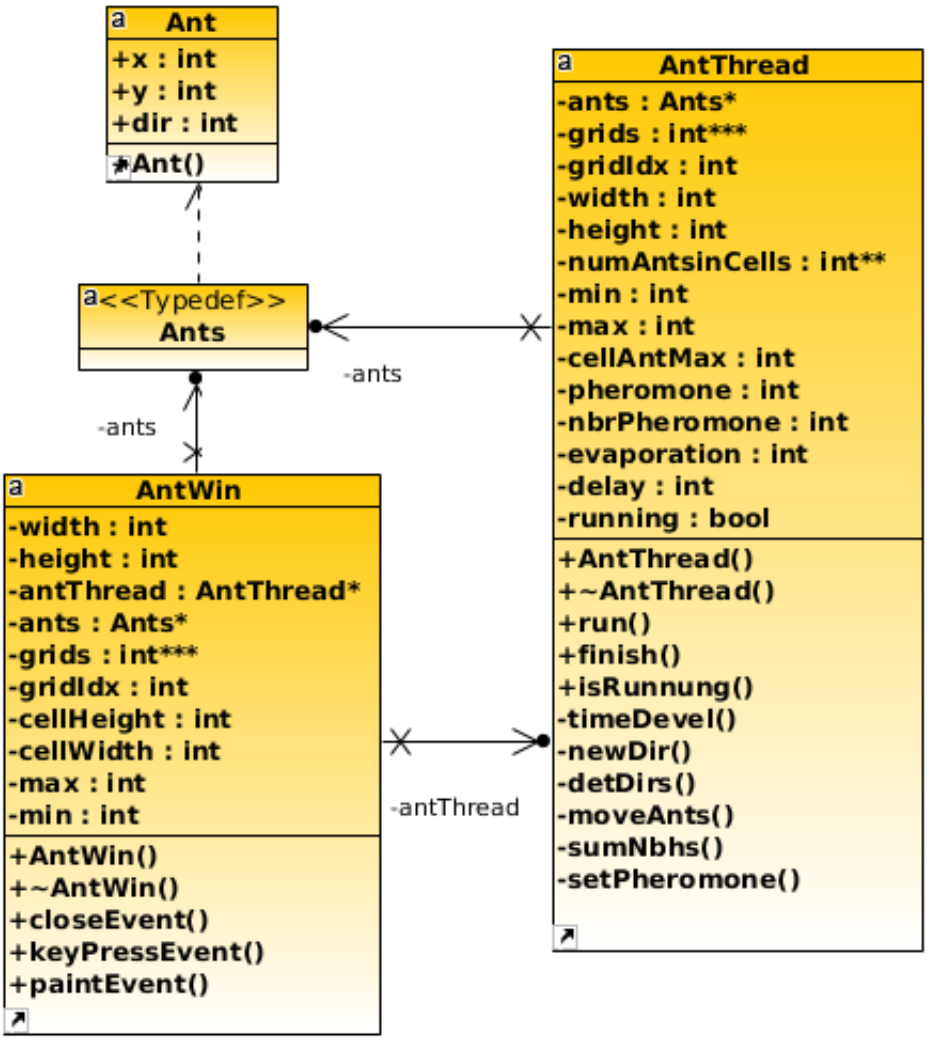
7.1. Hangyaszimulációk

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

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

Megoldás forrása: https://github.com/DonatPataki/University/tree/master/prog1/hangyaszimulacio

DRAFT



7.1. ábra. Forrás: https://bhaxor.blog.hu/2018/10/10/myrmecologist

Fordításához szükséges Qt telepítése. A program nem csinál mást mint a hangyák viselkedését próbálja szimulálni. Ezt úgy érhetjük el, hogy a képernyőt cellákra bontjuk és cellánként megkeresik a hangyák azt, akinek a "legerősebb a szaga" és abba az irányba indulnak el. A main.cpp-ben található egy ajánlott futattás kapcsolókkal együtt. Az alábbi kapcsolókkal a következőket tudjuk befolyásolni w a cella szélessége, m a cellamagasság, n a hangyák száma, t az idő, p a párolgás mérétke, f a feromon értékének nagysága, s a szomszédos cellákban milyen erős szagot hagy, d a cellánkénti alapérték, a a maximális érték ami egy cellában lehet, i a minimális és a c kapcsolóval azt adhatjuk meg, hogy hány hangya fér bele egy cellába.

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

    grid = grids[gridIdx];

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

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

            double rel = 255.0/max;

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

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

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



            qpainter.setPen (
                QPen (
                    QColor (0,0,0 ),   //visszaállítja fehér színre
                    1 )
            );

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

        }
    }

    for ( auto h: *ants) {
        qpainter.setPen ( QPen ( Qt::black, 1 ) ); //feketére állít majd ↩
           hangyákat rajzol

        qpainter.drawRect ( h.x*cellWidth+1, h.y*cellHeight+1,
                            cellWidth-2, cellHeight-2 );

    }

    qpainter.end();
}
```

A fenti kódrész tartalmazza a rajzolásokat és színváltásokat.

Az antthread.cpp tartalmazza a hangyák viselkedését. Vagyis itt van a kóddrész ami változtatja például a hangyák szag erősségének értéket, magának a hangyáknak számát a cellákban, az időt.

Az ant.h-ban megtalálható magának a tulajdonságai, amit én első ránézésre nem tudom melyik pontosan melyik mivel ezek nem beszédes nevek. De van egy érték, amit a konstruktor hoz létre és az random.

7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/life.java

John Horton életjátékának java verziója, ami az általa megadott egyszerű szabályokra épül.

A program úgy néz ki, hogy a main függvényen belűl létrehozunk egy példányát a game_of_life osztálynak, ami maga a játékot tartallmaza. Itt az is megfigyelhető, hogy a játékot valószíűleg a konsolban tervezték el, mert minden iterációban elküldjük 200 ms-re pihenni. Illetve az is esélyes ebből, hogy a class előre megadott módokon módosít mindenen, és egy játékostól vár majd inputot.

```
    public static void main(String args[])
    {
        game_of_life gol = new game_of_life();
        while(gol.ra.running)
        {
            if(!gol.ra.edit_mode)gol.update();
            try{Thread.sleep(200);}
            catch(Exception ex)
            {

            }
            gol.ra.update(gol.ra.getGraphics());
        }
        gol.dispose();
}
```

Ezután a program lefutatja saját konstruktorát, ami eddig engem igazolt a tippemmel. Itt nem látszik semmi extra vagy nagyon különleges és semmi sem tűnik randomnak.

```
    public game_of_life() {
        super("Game of Life");
        this.setSize(1005, 1030);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setVisible(true);
        this.setResizable(false);
        ra = new RenderArea();
        ra.setFocusable(true);
        ra.grabFocus();
        add(ra);
        int[][] siklokilovo = {{6,0},{6,1},
```

```
            {7,0},{7,1},
            {3,13},
            {4,12},{4,14},
            {5,11},{5,15},{5,16},{5,25},
            {6,11},{6,15},{6,16},{6,22},{6,23},{6,24},{6,25},
            {7,11},{7,15},{7,16},{7,21},{7,22},{7,23},{7,24},
            {8,12},{8,14},{8,21},{8,24},{8,34},{8,35},
            {9,13},{9,21},{9,22},{9,23},{9,24},{9,34},{9,35},
            {10,22},{10,23},{10,24},{10,25},
            {11,25}};
        int min_o = 5;
        int min_s = 85;
        for (int i = 0; i < siklokilovo.length; ++i)
        {
          ra.entities.get(min_o + siklokilovo[i][1]).set(min_s+ siklokilovo ↩
             [i][0],!ra.entities.get(min_o + siklokilovo[i][1]).get((min_s+ ↩
              siklokilovo[i][0])));
                this.update(this.getGraphics());
        }


        ra.edit_mode = false;
        ra.running = true;
}
```

Ezen kívűl egy pár dolog megtalálható benne. Például egy update eljárás, ami az enitiken végez módosításokat, annak megvelelően, hogy milyen kondició teljesűl. A forráskódját ennek most nem raknám be ide, mert nem akarom, hogy az egész könyv csak kód legyen és olyan sokat nem mond elsőre maga a kód.

Ezen kívűl van még néhány keyevent figyelő és egy programrész, ami a rajzolást végzi.

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/tree/master/prog1/eletjatek

John Horton Conway brit matematikus állt elő ezzel az ötlettel 1970-ben. A játék nem igényel inputot egy játékostól sem, mivel önmagát konfigurálja és megpróbálja a legjobb lövést végrehajtani. Van néhány szabálya a játéknak mint például akármelyik élő sejt meghal kevesebb mint két élő szomszéddal stb. Alapvetően három kategóriája van a játékban szereplő dolgoknak. Still lifes, amik nem csinálnak semmit se. Oscillators, amik visszatérnek eredeti alakjukhoz, adott lépésszám után. Valamint Spaceships, amik mozogni tudnak. Kb ez a lényegi része a Conway féle életjátéknak és a program ezen alapszik.

Létrehozzuk egy ablakot Qt-val.

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

DRAFT

```
    SejtAblak w(100, 75);
    w.show();

    return a.exec();
}
```

A sejtablak.cpp-ben már található izgalmasabb dolog, mint például maga a rajzolást végző egység. De itt vannak megadva a siklókilövő koordinátái + a siklóé is bár ahogy látom azok a koordináták x és y-hoz képest lesznek viszonyítva. Valamint még itt található egy update is.

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

    // Az aktuális
    bool **racs = racsok[racsIndex];
    // racsot rajzoljuk ki:
    for(int i=0; i<magassag; ++i) { // végig lépked a sorokon
        for(int j=0; j<szelesseg; ++j) { // s az oszlopok
            // Sejt cella kirajzolása
            if(racs[i][j] == ELO)
                qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                                  cellaSzelesseg, cellaMagassag, Qt::black) ↩
                                     ;
            else
                qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                                  cellaSzelesseg, cellaMagassag, Qt::white) ↩
                                     ;
            qpainter.setPen(QPen(Qt::gray, 1));

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

    qpainter.end();
}
```

A sejtszal.cpp-n belűl található a funkció, ami megadja hogy hány szomszédos élő egység van. Ez azért fontos mert van egy olyan szabály, ami megadja, hogy mennyi szomszédos egység után számít valami halottnak.

```
int SejtSzal::szomszedokSzama(bool **racs,
                              int sor, int oszlop, bool allapot) {
    int allapotuSzomszed = 0;
    // A nyolcszomszédok végigzongorázása:
    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)
            // A vizsgált sejtet magát kihagyva:
            if(!((i==0) && (j==0))) {
        // A sejttérbõl szélének szomszédai
```

```
        // a szembe oldalakon ("periódikus határfeltétel")
        int o = oszlop + j;
        if(o < 0)
            o = szelesseg-1;
        else if(o >= szelesseg)
            o = 0;

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

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

    return allapotuSzomszed;
}
```

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/tree/master/prog1/BrainB

Mivel manapság mindenki e-sportoló akar lenni, ez a teszt fázisban lévő program megpróbálja megbecsülni az emberek szellemi teljesítményét vagyis inkább, hogy mennyire képes több zavaró tényező között a saját karakterét követni, így elérve azt, hogy az adott személy alkalmas-e e-sportolónak vagy sem. Ezt úgy éri el, hogy több mozgó karakter közül olyan statisztikákat figyel, mint hogy hányszor veszíti szem elől a saját karakterét, mennyi ideig tart megtalálni stb. Majd kap egy összesített pontszámot, ami vagy jó vagy nem. Viszont ennek a pontnak még nem látszik túl sok értelme, mivel nincs bizonyítva, hogy ez egy működő koncepció vagy sem. Legalábbis én ezt az iinformációt kaptam.

Mivel ez egy kutatás része ezért van hozzá dokumentáció, aminek köszönhetően sokkal több mindent meglehet tudni ezzel kapcsolatban írni. De én csak egy egyszerű halandó vagyok, aki megpróbállja értelmezni a kódot és a dokumentáció olvasása nélkül.

Első ránézésre is vannak elég egyszerúen kivehető részek, hogy mi mit csinál a kódban. Ilyen például a hősök mozgása, ami teljesen random.

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

        dispShift = heroRectSize+heroRectSize/2;

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

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

DRAFT

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

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

        heroes.push_back ( me );
        heroes.push_back ( other1 );
        heroes.push_back ( other2 );
        heroes.push_back ( other4 );
        heroes.push_back ( other5 );

}
```

Van még pár könnyen kivehtő dolog, mint a billentyű/egéresemények, a statiszika része stb. De itt már érződik, hogy ez több időt igényel a megértéshez.

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

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

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/szoftmax.py

A MNIST egy tenserflow alapú könyvtár, ezért futatásához előbb telepíteni kell a tenserflowt. Ennek az lenne a lényege, hogy egy nagy adatbázis segítségével, ami biztosítva van ezáltal a tenserflow könyvtárait használva, különböző AI-okat lehet rajtauk edzeni. Viszont ez egy elléggé erőforrás igénylő feladat, szóval érdemes gpu gyorsítást használni. Ez a program edzését befejezően képes számokat felismerni a MNIST adatbázisból és remélhetőleg azon kívűl is. Érdemes azonban megjegyezni, hogy ezek az AI-ok csak bizonyos pontossággal működnek, így előfordúlhat viszonylag kis eséllyel, hogy téves megoldást ad. Minél tovább vannak edzve és minél nagyobb mintán elvileg annál pontosabbnak kell lenniük.

Először beimportálunk egy rakat dolgot, többek között az adatokat is. Azután elég sok minden fogad minket, ami talá nem néz ki olyan túl jól. Ez a rész végzi a hálózat tanítását, ahol 10 százalékonként jelzi, hogy hogy haladunk.

```
  tf.initialize_all_variables().run(session=sess)
  print("-- A halozat tanitasa")
  for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
    if i % 100 == 0:
      print(i/10, "%")
print("------------------------------------------------------------")
```

Ezután teszteljük, hogy mennyire lett pontos maga az AI. Ami innen érdekesebb lehet, hogy készít egy kisebb statiszkikát.

```
  matplotlib.pyplot.imshow(image.reshape(28,28), cmap=matplotlib.pyplot.cm. ↩
     binary)
  matplotlib.pyplot.savefig("8.png")
  matplotlib.pyplot.show()
```

DRAFT

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

Végül pedig ha ezt a python programot futattuk és nem másikból hívtuk akkor még lefut a lenti kód is.

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

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/deep_mnist.py

Hasonlóan az előző feladathoz, ez is igényli a tenserflow telepítését. És hasonlóan ez előző feladathoz, ez is számokat ismer fel, azzal a különbséggel, hogy most több réteget kapott az AI, így lehetővé teszi a még komplexebb programok létrehozását, ahol több réteg döntései által kapjuk meg a végeredményt. És itt egy kis vizualizáció mert most lusta vagyok magyarázni http://scs.ryerson.ca/~aharley/vis/conv/

Egy többrétegű neurális hálót használva 90% környékéről 99%-ra lehet emelni a pontosságát. A program elején ugyanúgy beimportéljuk a megfelelő dolgokat. Létrehozunk pár függvényt. És utána elkezdhetjük építeni.

```
# Input layer
x  = tf.placeholder(tf.float32, [None, 784], name='x')
y_ = tf.placeholder(tf.float32, [None, 10],  name='y_')
x_image = tf.reshape(x, [-1, 28, 28, 1])

# Convolutional layer 1
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])

h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

# Convolutional layer 2
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

# Fully connected layer 1
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
```

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# Dropout
keep_prob  = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# Fully connected layer 2 (Output layer)
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2, name='y')

# Evaluation functions
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),  ↩
   reduction_indices=[1]))

correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name=' ↩
   accuracy')

# Training algorithm
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

# Training steps
with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())

  max_steps = 1000
  for step in range(max_steps):
    batch_xs, batch_ys = mnist.train.next_batch(50)
    if (step % 100) == 0:
      print(step, sess.run(accuracy, feed_dict={x: mnist.test.images, y_:  ↩
         mnist.test.labels, keep_prob: 1.0}))
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys, keep_prob:  ↩
       0.5})
print(max_steps, sess.run(accuracy, feed_dict={x: mnist.test.images, y_:  ↩
   mnist.test.labels, keep_prob: 1.0}))
```

8.3. Minecraft-MALMÖ

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

Megoldás forrása: https://github.com/Microsoft/malmo

MALMÖ egy open source projekt, amit Microsoft kezdeményezett egy pár évvel ezelőtt. A célja az, hogy

kicsit máshogyan közelítse meg az AI kutatásokat, amit új problémákkal és egyedi környezettel próbál megvalósítani. Ezt úgy éri el, hogy a minecraft környezetére építetettek rá egy AI-t, amivel lehet kisérletezni. A projekt több programázási nyelvet is támogat ezek python, c++, c#, java és atari, azaz nincs probléma azzal, hogy platformfüggő lenne. Továbbá minden nyelvhez található minték, amiket futattni/módosítani lehet.

Kicsit elszámoltam magam és végül nem sikerült magamtól írni. De hogy mégis legyen itt valami ezért megnézunk egy minta AI-t.

```
// Malmo:
#include <AgentHost.h>
#include <ClientPool.h>
using namespace malmo;

// STL:
#include <cstdlib>
#include <exception>
#include <iostream>
using namespace std;

int main(int argc, const char **argv)
{
    AgentHost agent_host;

    try //ha esetleg valami hiba lene akkor az it el lesz kapva
    {
        agent_host.parseArgs(argc, argv);
    }
    catch( const exception& e )
    {
        cout << "ERROR: " << e.what() << endl;
        cout << agent_host.getUsage() << endl;
        return EXIT_SUCCESS;
    }
    if( agent_host.receivedArgument("help") ) //lelehet kérni a használatot
    {
        cout << agent_host.getUsage() << endl;
        return EXIT_SUCCESS;
    }

    MissionSpec my_mission; //missio specifikációi
    my_mission.timeLimitInSeconds(10);
    my_mission.requestVideo( 320, 240 );
    my_mission.rewardForReachingPosition(19.5f,0.0f,19.5f,100.0f,1.1f);

    MissionRecordSpec my_mission_record("./saved_data.tgz"); //hova menti ↩
       hogyan stb
    my_mission_record.recordCommands();
    my_mission_record.recordMP4(20, 400000);
    my_mission_record.recordRewards();
    my_mission_record.recordObservations();
```

```
    int attempts = 0; //kísérletek száma
    bool connected = false;
    do {
        try { //megpróbálunk csatlakozni
            agent_host.startMission(my_mission, my_mission_record);
            connected = true;
        }
        catch (exception& e) {
            cout << "Error starting mission: " << e.what() << endl;
            attempts += 1;
            if (attempts >= 3)
                return EXIT_FAILURE;    // ha már háromszor nem jött össze ↩
                   akkor nem is fog
            else
                boost::this_thread::sleep(boost::posix_time::milliseconds ↩
                   (1000));   // ha még nem akkor újrapróbáljuk
        }
    } while (!connected);

    WorldState world_state;

    cout << "Waiting for the mission to start" << flush;
    do {
        cout << "." << flush;
        boost::this_thread::sleep(boost::posix_time::milliseconds(100));
        world_state = agent_host.getWorldState();
        for( boost::shared_ptr<TimestampedString> error : world_state. ↩
           errors )
            cout << "Error: " << error->text << endl;
    } while (!world_state.has_mission_begun);
    cout << endl;

    // ez a lényegi rész
    do {
        agent_host.sendCommand("move 1");
        {
            ostringstream oss;
            oss << "turn " << rand() / float(RAND_MAX);
            agent_host.sendCommand(oss.str());
        }
        boost::this_thread::sleep(boost::posix_time::milliseconds(500));
        world_state = agent_host.getWorldState();
        cout << "video,observations,rewards received: "
             << world_state.number_of_video_frames_since_last_state << ","
             << world_state.number_of_observations_since_last_state << ","
             << world_state.number_of_rewards_since_last_state << endl;
        for( boost::shared_ptr<TimestampedReward> reward : world_state. ↩
           rewards )
            cout << "Summed reward: " << reward->getValue() << endl;
        for( boost::shared_ptr<TimestampedString> error : world_state. ↩
```

```
            errors )
             cout << "Error: " << error->text << endl;
    } while (world_state.is_mission_running);

    cout << "Mission has stopped." << endl;

    return EXIT_SUCCESS;
}
```

Igazából itt nem történt semmi extra, csak memondtuk az agentnek hogy fusson és random fokokban forduljon el. A célja pedig 10 sec alatt elérni a megadott pontra.

DRAFT

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/faktorialis.lisp

Olyan sok magyarázatot szerintem nem igényel, de itt van pár info róla. Két függvény található benne.

Az első rekurzívan hívja magát, addig amíg nullát nem kap. Persze ha minusz értéket kap maga a függvény, akkor azzal szépen el lehet rendezni.

```
(defun factorial (n) ; rekurzív
  (if (= n 0)
      1
(* n (factorial (- n 1))) ) )
```

A második függvény pedig addig iterál, ameddig el nem éri az adott számot és közben szoroz.

```
(defun fact (n) ; iteratív
    (do
       ((i 1 (+ 1 i))
        (prod 1 (* i prod)))
((equal i (+ n 1)) prod)))
```

Fontos megjegyezni, hogy csak ezt a két függvényt tartalmazza a forrás és sehol nincsenek meghíva, így futattás során nem ír ki értéket. Ha azt szeretnénk, hogy írja is ki egy szám faktoriálisát, akkor azt a következőképpen tehetjük meg.

```
(factorial i) ; i egy természetes szám
```

A lisp amúgy egy elég régi magas szintű programozási nyelv, amit az 50-es évek végén fejlesztettek ki, de persze azóta sokat változott. Alapvetően matematikai műveletek végeztek vele akkoriban, de később elég fontos szerepe volt AI kutatásokban.

DRAFT

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://github.com/DonatPataki/University/blob/master/prog1/bhax_chrome3.scm

Ha egyszerűen akarnám elmagyarázni, akkor azt mondanám, hogy Script-Fu segítségével scripteket lehet írni gimphez, a scheme programozási nyelv segítségével, ami hasonló a lisphez. Azaz automatizálni lehet akár egyszerűbb vagy komplikáltabb feladatokat is.

A króm effekt lényege, hogy a program kap egy szöveget és azt úgy, alakítja át, hogy azt az illuziót keltse, hogy a szövegnek fémes tulajdonságai vannak. Ez persze megkövetel pár előkészületet mint például maga a színgörbe amit használni szeretnénk, valamit az hogy a szövegből megkapjuk a számonkra fontos pixeleket.

Ezek után elkezdhetjük a króm effekt létrehozását, amit egyszerű lépésekre lehet bontani és szerencsékre van is dokumentáció is róla, amit tanár úr is használt. http://penguinpetes.com/b2evo/index.php?p=351

Az első lépés alapján először létrehozunk egy fekete háteren, a már általunk beírt szöveget fehér betűszínnel. Ezt a scriptben úgy tudjuk megoldani, hogy először létrohozunk egy fekete réteget és mégegyet, ahol maga a szöveg van fehérrel. Majd végül a két réteget egyesítjük.

```
(gimp-image-insert-layer image layer 0 0)
(gimp-context-set-foreground '(0 0 0))
(gimp-drawable-fill layer  FILL-FOREGROUND )
(gimp-context-set-foreground '(255 255 255)) ; feketére layer  ↩
   létrehozásának vége

(set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS)))
(gimp-image-insert-layer image textfs 0 0)
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (- (/  ↩
   height 2) (/ text-height 2))) ; szöveg középre igazításával vége a  ↩
   szöveges rétegnek is

(set! layer (car(gimp-image-merge-down image textfs CLIP-TO-BOTTOM-LAYER))) ↩
    ; végül egyesítjük a két régteget
```

Ezután második lépésben elmossuk az egészet, hogy az éles széleket, kissé elsimitsuk.

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 15 TRUE TRUE)
```

A második lépés miatt a szövegünk nem csak fehér színből áll, amit a harmadik lépésben korigálunk. Így kapva egy lekerekített szélekkel rendelhező fehér szöveget.

```
(gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE)
```

A negyedik lépésben ismét haszálunk egy kis elmosást, de ezuttal kisebb mértékben. Ennek jelenleg nem látszik sok értelme, de majd a 8. lépsénél lesz jelentősége.

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE)
```

Az ötödik először kijelöljük a fekete részeket, majd ezt megfordítva megkapjuk magát a szöveget.

```
(gimp-image-select-color image CHANNEL-OP-REPLACE layer '(0 0 0))
(gimp-selection-invert image)
```

Ezután létrehozunk egy új réteget.

```
(set! layer2 (car (gimp-layer-new image width height RGB-IMAGE "2" 100 ↩
   LAYER-MODE-NORMAL-LEGACY)))
(gimp-image-insert-layer image layer2 0 0)
```

Ezután a hetedik lépésben az új rétegen kijelölt részt felöltjük szürke árnyalatokkal

```
(gimp-context-set-gradient gradient)
(gimp-edit-blend layer2 BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY ↩
   GRADIENT-LINEAR 100 0 REPEAT-NONE
FALSE TRUE 5 .1 TRUE width (/ height 3) width (- height (/ height 3)))
```

Nyolcadik lépésben az első réteget felhasználva bump mapet használunk az második rétegen.

```
(plug-in-bump-map RUN-NONINTERACTIVE image layer2 layer 120 25 7 5 5 0 0 ↩
   TRUE FALSE 2)
```

Végül a a kilencedik lépésben alkalmazzuk a szingörbért, amit a program legelején létrehoztunk és készen is vagyunk.

```
(gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve)) ; színgörbe ↩
   alkalmazása

(gimp-display-new image)
(gimp-image-clean-all image) ; utóbbi két sor annyira nem fontos, ezzel ↩
   csak magát a szöveget fogjuk látni
```

A progrmot pedig zárjuk a register-rel, hogy minden tökéletesen működjön.

9.3. Gimp Scheme Script-fu: név mandala

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

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

Megoldás forrása: https://github.com/DonatPataki/University/blob/master/prog1/bhax_mandala9.scm

Röviden a script a megadott névből készít egy mandalát, mert a mandalák nyugtató hatásuak és így a vizsgaidőszak előtt sosem árt egy kis nyugalom. Részletes leírás itt található, amit tanár úr is használt. https://gimplearn.net/viewtopic.php?f=10&t=268

Hasonlóan az előzőhöz képest itt is pár előkészületet, mint például mely pixeleket hazsnálja, háttérszín babrálása stb. Most viszont nem tervezem teljesen részletesen leírni, mivel nem akarom a végtelenségig írni a lépéseket. Ehelyett inkább csak a lényegesebb részeket kiemelni.

Szóval röviden a szöveget ismét beállítjuka a kép közepére, majd egy új rétegre másoljuk, ahol 180 fokkal elforgatjuk és a két réteget egyesítjük. Így megkapva a nevünket kétszer leírva. És ezzel a forgatgatásokkal szeretnénk elérni, hogy valamennyire kör alakban legyen kiírva a nevünk, ami igazából egy tetszőleges szöveg.

```
(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate-simple text-layer ROTATE-180 TRUE 0 0)
(set! textfs (car(gimp-image-merge-down image text-layer  ↩
   CLIP-TO-BOTTOM-LAYER)))
```

A fent leírt lépéseket megismételjük 90, 45 és harminc fokkal is. Azaz folyton létrehozunk egy új réteget, ahova bemásoljuk a szövegtet. Majd elforgatjuk és a két réteget egymásbaillesztjük.

```
(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 2) TRUE 0 0) ; ezt az  ↩
   egészet pi/4 és pi/6-al is megcsináljuk
(set! textfs (car(gimp-image-merge-down image text-layer  ↩
   CLIP-TO-BOTTOM-LAYER)))
```

Ezt köveetően én már azt mondanám, hogy ez elég jó nekem mandalának, de sajnos sem a részletes leírás sem pedig tanár úr nem volt megelégedve ezzel, szóval még hogy mégjobban meglegyen a kör nyugtató hatása hozzá fogunk adni, két különböző fastagságú kört. A körök rajzolása teljsen megegyezik.

```
(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width 2) (/  ↩
   textfs-width 2)) 18)
(- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) (+  ↩
   textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)
```

Ez azért van mert az ecsetvastagságot külön változtatjuk, illetve ez igaz a kép méretére is, azaz a körök megrajzolása maradhat változatlanul.

Mostmár megkaptuk a kör nyugtató hatását, már csak annyi baj van vele, hogy kicsit unalmasan egyszínű. Szóval kijelöljük a szövegeket és a köröket és adunk hozzá egy is színátmenetet. Amivel meg is van a végső simitás. És a scriptet zárjuk a register-rel és mindenki boldog.

DRAFT

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

A programozási nyelveket, több csoportba lehet sorolni. Megkülönböztetünk gépi, assembly és magas szintű nyelveket. Minden processzorhoz tartozik egy saját gépi nyelv, ami alapján végre tudja hajtani az utasításokat. Ezért valamilyen fordítóprogramra van szükség, ami a magasabb szintű nyelvekből gépi kódot formál. Erre az egyik lehetséges megoldás fordítóprogram használata, ami több szempontból is elemzi a forráskódot, majd abból tárgykódot generál. Ez a tárgy kód még nem futattható, mivel a kapcsolatszerkesztő még nem rakta össze teljesen. Miután ez is megvan megkaptuk a futattható kódot. A mások módszer az interpeteres megvalósítás, ami ugyanúgy több szempontból elemzi a forráskódot, de abból nem készít tárgykódot, hanem egységenként értelmezi.

Programozási nyelvek tulajdonságaik alapján több féle kategóriába sorolhatók. Léteznek imperatív nyelvek, amik algoritmusosan gondolkodnak, utasításokkal és változókkal dolgoznak. Ez tovább bontható eljárásorientált és objektumorientált nyelvekre. A következő nagy csoport a deklaratív nyelvek, amik nem kötődnek annyira szorosan a Neumann-architektúrához. Ide tartoznak a logikai és funkcionális nyelvek. Valamint léteznek más elvű nyelvek, amiket nem lehet besorolni az előbbi két kategóriába.

Megkülönböztetünk típusos és ne típusos programozási nyelveket. Az adattípus jellemzői közé tartozik, hogy mettől meddig vehet fel értéket, milyen műveleteket lehet elvégezni és hogyan van tárolva. Programozási nyelvtól függően akár lehetőségünk is van saját típust definiálni a tulajdonságait megadva. Megkülönböztetünk egyszerű és összetett adatszerkezeteket. Az előbbí nem bontható tovább, míg az utóbbi több fajta adattípusból tevődik össze.

Az egyszerű típushoz tartoznak a numerikus típusok, amik lehetnek egészek vagy valós számok. Akár lehetőség van előjel nélküli típusra is. Továbbá léteznek karakteres típusok, logikai típusok és felsorolás típusok.

Összetett típusokból a két legfontosabb a tömb és a rekord. A tömb egy adott adattípus láncolata, ahol az elemekre lehet hivatkozni. Programozási nyelvtől függően akár többdimenziós tömbök megadása is lehetséges. Azt hogy a tömb milyen típust taratalmaz, hogyan indexelhető és hogyan adható meg az elemszámok határa minden nyelv magának dönti el. A rekord adattípus különböző adattípusokat fog egy kupacba mint egy mezőt és lehetőséget ad a rekord mezőinek hivatközéséra.

Létezik még mutató típus is, ami egy egyszerű típus azzal a tulajdonsággal, hogy memóriacímeket lehet benne tárolni, és elérni. Ha nem mutat sehova, akkor null értéket vesz fel.

DRAFT

Konstansok használata során, ha egyszer megadtuk az értékét, azt később a program futása során már nem lehet módosítani. Ennek annyi értelme van, hogy jól ismert értékeket például névvel lehet ellátni vagy pedig ha valami többször is előfordúl és szeretnénk az értékét megváltoztanti, akkor elég ezt egyhelyen megtenni.

A változóknak négy komponense van. Név, attribútum, cím és érték. Név alapján tudjuk azonosítani. Az atríbútum álltalában a típusát jelöli. Az értéke attól függ, hogy deklarációkor adtunk-e neki, ha nem akkor nyelvtől függő, hogy kap-e alapértelmezett értéket vagy sem. Valamint a cím tartalmaza, a memóricímet, ahol található a változó. Ez utóbbi is akár lehet programozó álltal meghatározott. Az értékadó utasítás bal oldalán található a változó és jobb oldalon, hogy mit szeretnénk hozzárendelni. Vannak olyan nyelvek, ahol nem lehet használni a változót, ameddig nem rendeltünk hozzá értéket.

Kifejezések segítségével új értékeket határozhatunk meg. Az operandusok száma szerint különböztetjük meg őket. Azaz léteznek egyoperandusú, kétoperandosú stb operátorok. Az operandus helye változhat. Valamnt a művelet végrehajtásának sorrendje is különböző lehet. De az adott precedencia szerint hajtódnak végre. Ha a kifejezés logikai operátorokat is tartalmaz, akkor előfordúlhat nem kell teljesen elvégezni a kiértékelést, ahhoz, hogy végeredményt kapjunk mert a logikai összekötésnek köszönhetően egyértelművé válik. Nyelvtől függően lehet konverzió két különböző típúsu érték összehasonlítása során és az is lehet, hogy két különböző típús hasonlítása nem megengedett. Előbbi persze csak akkor lehetséges, ha a két típús között van konverzió. Azonban lehetséges, hogy kerektés forg történni konverzió során.

Megkülönboztetünk deklarációs és végrehajtható utasításokat. Előbbi a fordítóprogramnak szól, míg utóbbiból tárgykód is készül. Ami ezek közül izgalmasabb az az elágazó utasítsok, ami lehet két vagy többirányú. Kétirányú az if, ahol adott feltétel szerinti kódrész kerül lefutásra. Többirányú a switch case, ami szintén a megadott feltételnek megfelelő kódrészt futatt le, de itt lehetőség van default rész használatára, ami akkor fut le ha egyik case sem egyezett meg. Továbbá vannak ciklusok, amik jellemzője, hogy állnak egy fejből, magból és egy végből. Az hogy meddig kell ismételni, vagy a fejben, vagy a végben találtható és a mag utasításait hajtaja végre. A ciklusoknak több fajtája van, ezt mindenkinek magának kell eldöntenie, hogy a program során milyet érdemes használni. Van belőlük feltételhez szaabott, előírt lépésszámú stb. Végezetűl aóvan még három vezérlő utasítás. A continue segítségével ki lehet hagyni az adott iterációban az utáni kódokat. Break megtöri a ciklust. A return pedig megtöri és visszaad egy értéket.

Az eljárás orientált programozási nyelvek programegységekre bonthatók és az adott programozási nyelven múlik, hogy az adott részeket, hogyan kezeli. Az egyik ilyen programegység, az alprogram, aminek előnye, hogy nem kell többször ugyanazt megírni, hanem meghívhatjuk ugyanazt az alprogramot többször is. Szintén tulajdonságuk, hogy lehet paraméterezni őket, azaza nem teljesen statikusak. Attól függően, hogy az alprogram értéket ad vissza vagy csak elvégez pá műleteket, megkülöböztetünk eljárást és függvényt. Ez utóbbi lesz aminek célja, hogy egy értéket adjon vissza. Persze itt figyelni, kell hogy legyen mit visszaadnia. Továbbá midig van egy főprogram, ami mindig lefut, ha a programot elindítjuk. A programegységek meghívása során az is lehetséges, hogy a meghívott programegység hívjon meg további programegységeket. Ez lehet akár saját maga is vagy már korábban hívott programegység. Így kialakítva hívási láncokat vagy rekurziót.

Paraméterátadás segítségével lehet az alprogram hívása során megadni, hogy milyen információkkal dolgozzon. Ez a paraméterátadás történhet többféleképpen is. Ha érték szerint adjuk át, akkor az értékét adjuk át és azt az alprogramon belűl letárolja. Lehet cím szerint is. Ilyenkor az alprogramon belűl nem készűl másolat, hanem az eredeti változóval dolgozik. Eredmény szerinti átadás az előző kettő keveréke, ahol készül másolat, de a végén átmásolja az eredményt. Lehet még név és szöveg szerint is átadni, de persze ezeket programozási nyelv válogatja, hogy melyiket támogatja.

A programon belűl lehetnek blokkok, amiknek szorosan kapcsolódik például egy változó hatásköre. Ez alatt azt kell érteni, hogy az adott változó csak a blokkon belűl érhető el. Az ilyen változót szokás lokális

DRAFT

változónak is hívni. Valamint globális változónak, azt amit a program bármely részén el lehet érni.

A blokkokon belűl is létre lehet hozni blokkokat. A függvények alapértelmezetten int típust adnak vissza c-ben. A c lehetőséget ad más fordítási egységek eszközeinek használatára is.

A generikus programozás alapgondolata, hogy újra felhasználható legyen a kód. Ennek paraméterei fixek.

A perifériákkal történő kommunikáció vagy más néven I/O műveleteket az operációs rendszer írja le. Az állományokon a következő műveleteket leht végrehajtani deklaráció, összerendelés, állomány megnyitása, feldolgozás, lezárás. C-ben a standard könyvtár része az I/O műveletek.

A kivételkezelést használva maszkolni lehet az operációs rendszerek álltal dobot kivételeket. Ennek oka lehet maga a hibás algoritmus vagy egyszerűen olyan dolog is amit nem lehet értelmezni pl nullával való osztás.

Ezeket most meg kéne válaszolnom és ha igen akkor c-re vagy melyik nyelvre?

Milyen beépített kivételek vannak a nyelvben?

Fogalmam sincs. A standard libraryn belűl van egy pár.

Definiálható-e a programozó saját kivételt?

Igen.

Milyenek a kiételkezelő hatásköri szabályai?

A kivételkezelő köthető-e programelemekhez?

Odarakja az ember ahova akarja szóval gondolom igen.

Hogyan folytatódik a program a kivételkezelés után?

Fut tovább.

Mi történik, ha kivételkezelőben következik be kivétel?

Ha a trí blokkon belűl van megy a catchre, feltéve ha úgy van megadva a catch és ha van olyan amelyikre mehet. ha a catchben van a gond az szívás.

Van-e a nyelvben beépített kivételkezelő?

Try catch?

Van-e lehetőség arra, hogy bármely kivételt kezelő kivételkezelőt irjunk?

catch(...)

Lehet-e parametizálni a kivételkezelőt?

Igen

Neumann-architektúra szerint a gépke szekvenciálisak, azaz egyszerre egy feladatot végeznek el egymás után. Viszont lehetőség van párhuzamostítani, mivel manapság többmagos gépek léteznek. De ilyenkor figyelni kell, hogy a folyamatok kommunikáljanak egymással. Hogy egyszerre ne tudjanak módosítani az adatot stb.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

DRAFT

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

Egy kifejezés akkor válik utasítássá, ha azt ;-vel zárjuk le. {} hsznélatával blokkokat lehet létrehozni, amik lehetővé teszik az összetett utasítások használatát például ciklusoknál.

If ustasítás utatsítás utáni blokk akkor hajtódik végre, hogy ha a feltétel igazként értékelődött ki. Továbbá lehetőség van else rész használatára, am hamis érték esetén fut le. Az if utasításba további ifeket lehet beszúrni. Lehet még else helyett else if-t is használni, ami annyit tesz hogy megadhatunk mégegy feltételt és ugyanúgy viselkedik mint az if. Azaz ebbe is lehet további else if-t, else-t használni.

A switch case hasonló az if else if kifejezéshez, azonban itt egy kifejezést adunk meg a legelején és a case-nél adunk meg egy állandó értéket. Így a megkeresi azt, ahol egyezik a kifejezés értéke a megadott konstansokkal és az az ág fog lefutni. Lehet persze adni egy default részt is, ami akkor fut le ha nem talált egyező case-t. Valamint a case-et break-kel kell elválasztani.

A while egy előtesztelős ciklus, ami annyit jelent, hogy addig hajtódik végre a ciklusmag, amíg a feltétel igaz. A for előírt lépésszémú. Itt a fejben kell megadni három kifejezést. Persze ezek megadása nem feltétlen szükséges. Nem mindet muszáj megadni, de ez végtelen ciklushoz vezethet. A do while hasonló a while-hoz csak ez hátultesztelős ciklus, ami azt jelenti, hogy egyszer mindeképp lefut és utána nézi meg, hogy a feltétel teljesül-e, ha igen akkor ismét lefut. Continue használatával lehet ugrani az ciklusban egy iterációt és break-el lehet megszakítani az egész ciklust.

Az utasításuk sorrendben hajtódnak végre, ahogyan le lettek írva alapértelmezetten, de ezt lehet módosítani.

Cimke utasításhoz tartozik a switch-nek a case és default. Ez annyit jelent, hgoy futás során egyszerűen majd csak a megfelelő cimkéhez fog menni a program.

Kifejezésutasítások a lehetnek függvényhívások vagy valamilyen értékadás.

Összetett utasításokat úgy tudunk létrehozni, hogy egy blokkba írjuk az utasításainkat, így kijátszva azt, hogy például csak egyetlen utasítást fogadjon el. Mert így a blokk összes utasítása lefut például egy ciklus után. Egy blokkon belűl egy aazonsosítót csak egyszer lehet deklarálni.

Kiválasztó utasítás az if és a switch case. Az if után megadunk egy kifejezést és ha az igaznak bizonyosúlt akkor lefut az if utáni blokk vagy utasítás. Lehet hozzá használni else ágat is, ez akkor fut le ha hamis volt az utasítás és persze ifeket lehet egymásba is ágyazni. A switch case igazából hasonlít az ifhez, csak itt a switch elején megadjunk égy értéket, vagy kifejezést és a megfelelő case blokja fog végrehjtódni.Lehetődésg van hozzáadni egy default részt, ami akkor fut le ha egyig case sem egyezett meg.

Iterációs utasításhoz tartoznak a ciklusok. Létezik feltételhezt kötött ciklus ez lehet a while és a do while. A kettő között csak annyi a különbség, hogy az első előtesztelős a másik utótesztelős tehát egyszer mindeképpen lefog futni. Valamint van megadott lépésszámú ciklus is ez pedig a for ciklus, ami megadott számig fog lefutni. Ehhez 3 kifejezés tartozik.

Végül vannak még vezérlő utasítások, mint például goto continue break és retrunk. Goto-val programon belűl megadott azonosítóhoz lehet menni. A continue a ciklusmagon belűl a következető iterációra ugrik, a break pedig megszakítja azt. A return hasonlü a breakhez csak még értéket is ad vissaz.

10.3. Programozás

[BMECPP]

DRAFT

C-ben ha a nem adunk meg függvényparamétert, akkor az azt jelenti, hogy neki tetszőleges lehet , míg c++ ez azt jelenti, hogy nincs neki. Ha mégis tetszőleges számú paraméterrel akarunk természetesen az is megoldható. További különbség, hogy c-ben nincs logikai változó, hanem ott int értékkként kezeli, míg c++-ban van külön logikai változó. C++ lehetőség van függévnyek túlterhelésére is, ezt úgy érhetjük el, hogy azonos néven, de különböző paraméterszámmal deklarálunk függvényeket. Így igazából a forráskódban ugyanaz a nevük de a linker majd megváltoztatja azt. Adhatunka függvényeknek alapértelmezett értéket is. Paraméterátadást végezhetünk referencia szerint, ez azt jelenti, hogy ilyenkor nem készül a híváson belűl egy másolat az átadott paraméterről, hanem azt a címet kapja meg, ahol a változó van tárolva.

Az objektumorientált programozás, hasonló a való élethez, mivel megengedi az osztályok használatát. Eze lényege az, hogy létrehozhatunk egy mintát. Például egy osztályba leírjuk, hogy milyen egy banszámla és ezt az osztályt példányosíthatjuk, akár különböző értékekkel. Az osztályok továbbá lehetőségetadnak, az adatok elrejtéséreí így kívűlről nem lehet eldönteni, hogy mi van az osztály belsejében, ha szeretnénk. Az osztályok örökölhetnek egymástól. Például ha B osztály örököl A osztálytól, akkor B-nek lesznke olyan részei, ami teljesen A-tól származik és, ehhez még hozzájön, hogy B-t hogyan szeretnénk még pontosabbá tenni. Igazából az osztályok olyanok mint egy továbbgondolt struktúra, mert például, ha stuktúrában létrehozunk egy koordináta rendszert, akkor csak a struktúrán kívűl lehet létrehozni függvényt, ami használja azon értékeit. De az osztály használatával ez a probléma megszűnik, mivel függvénykeket is deklarálhatunk benne. Ha adatookat szeretnénk elrejteni, akkor azt a private szóval thetjük meg, ez annyit tesz, hogy az osztályon kívűl nem lehet elérni a változókat/függvényket. Ha azt szeretnénk, hogy elérhető legyen akkor a public kulcsszót kell használnunk. Az osztályokat a konstruktor hozza létre, amit mi is deklarálhatunk magunktól. Ehhez nem kell mást tennünk, mint az osztály nevével létrehozunk egy függvényt. A konstruktor akkor fut le, amikor az osztályból készül egy példány és itt lehet megadni, hogy hogyan mit csináljon az osztály létrehozáskor. Például milyen értékei legyenek, írjon-e ki vaalmit esetleg az osztályon kívűl csináljon valamit vagy hívjon-e meg függvényt stb. A destruktor hasonló a konstruktorhoz, de ez akkor fut le, amikor az adott példány befejezte életét. Ezt is meg lehet adni ugyanúgy mint a konstruktort, de előtte használni kell a ~ jelet. Létezik még másoló konstruktor is, amit arra használunk, hogy osztályok értékeit másoljuk át egy új osztályba. Ezt magunk is megadhatjuk, de ha az osztályban nincs mutató felesleges, hacsak nem akarunk még valami plusz feladatot adni a másoláson kívűl. Mivel a beépített másoló konstruktor addig működik jól, ameddig nem használunk mutatuókat. Korábban említettem, hogy a private szóval lehet adatokat elrejteni, ezzel viszont az a baj, hogy ilyenkor az osztály nem örökli a private-ként megjelölt részeket. Ha mégis szeretnénk, hogy valamennyire legyeenk elrejtve az adatok és mégis tudjanak örökölni, akkor azt a friend kulcsszóval tehetjük meg. Továbbá lehetőség van static kulcsszóval olyan változót deklarálni, ami minden osztályban ugynaaz lesz és nem tagváltozóként fog létrejönni, hanem egy közös memóriacímen. Viszont ezt tagfüggvényekkel nem oldhatjuk meg. Végül ha nagyon szeretnénk, használhatunk beágyazott definíciót, ami nem tesz mást mint hogy más névvel tudunk adott dolgokra hivatkozni.

Az operátor túlterhelés annyit tesz, hogy a már létező operátorok működését megváltoztathatjuk. Ezt a legegyszerűbben talán másoló értékadással lehet elmagyarázni. Ha már létrehoztunk egy osztályon belűl egy másoló konstruktort, akkor az akkor fog lefutni, ha példányosítjuk és akkor adjuk meg az értékét. De ha már van egy példányunk és azt szeretnénk egy egyenlővé tenni egy másik osztály értékeivel ezt is megoldható. Ehhez nem kell mást tenni mint kiválasztani egy operátort és túlterhelés során hozzáírni a másoló konstruktor lépésit. Például alapesetben az = jel osztályok esetén nem jelent sok mindent, de ha az előbb leírtak alapján túlterhejük akkor működni fog.

A c++ több féle adatfolyamot különböztet meg. Létezik standard bemenet, standard kimenet, hiba kimenet és log kimenet. Ezek szerintem maguktól értetődnek, hogy melyik micsoda. Persze ezeket lehet variálni, hogy hogyan hogyan működjenek. Fájlműveleteket az fstream segítségével tudunk elvégezni. Itt van egy pár lehetőség, hogy hogyan nyissuk meg vagy hogy egyáltalán mit akarunk a fájllal.

DRAFT

Előfordulhat, hogy a program futása során hibakezeléssel kell foglalkoznunk, mert azt akarjuk, hogy valami bizonyos értékkel fusson le vagy csak mert a kódban van olyan rész, ami hibát eredményezhet. Ez lehet például egy számológép esetén 0-val való osztás, ami kifog a cpu-n. Ezen problémák kezelésére tökéletes a try catch blokkok. Ez úgy működik hogy egy try blokkba belerakjuk, azt a kódot, ami hibához vezethet és ha valóban hibát okoz, akkor rögtön ugrik a catch részre, már amennyiben úgy adtuk meg, hogy mindent kapjon el vagy bizonyos kivételeket. Végrehajtja ami ott van és fut tovább a program mintha misem történt volna. Ha például azt akarjuk, hogy beolvasás folyamán bozonyos értéket ne lehessen ezt úgy thejük meg, hogy után throw-al dobunk egy hibát és így ismét a catch rész fog lefutni.

DRAFT

III. rész

Második felvonás

DRAFT

Bátf41 Haxor Stream
A feladatokkal kapcsolatos élő adásokat sugároz a https://www.twitch.tv/nbatfai csatorna, melynek permanens archívuma a https://www.youtube.com/c/nbatfai csatornán található.

DRAFT

11. fejezet

Helló, Berners-Lee!

11.1. Szoftverfejlesztés C++ nyelven

C-ben ha a nem adunk meg függvényparamétert, akkor az azt jelenti, hogy neki tetszőleges lehet , míg c++ ez azt jelenti, hogy nincs neki. Ha mégis tetszőleges számú paraméterrel akarunk természetesen az is megoldható. További különbség, hogy c-ben nincs logikai változó, hanem ott int értékkként kezeli, míg c++-ban van külön logikai változó. C++ lehetőség van függévnyek túlterhelésére is, ezt úgy érhetjük el, hogy azonos néven, de különböző paraméterszámmal deklarálunk függvényeket. Így igazából a forráskódban ugyanaz a nevük de a linker majd megváltoztatja azt. Adhatunka függvényeknek alapértelmezett értéket is. Paraméterátadást végezhetünk referencia szerint, ez azt jelenti, hogy ilyenkor nem készül a híváson belűl egy másolat az átadott paraméterről, hanem azt a címet kapja meg, ahol a változó van tárolva.

Az objektumorientált programozás, hasonló a való élethez, mivel megengedi az osztályok használatát. Eze lényege az, hogy létrehozhatunk egy mintát. Például egy osztályba leírjuk, hogy milyen egy banszámla és ezt az osztályt példányosíthatjuk, akár különböző értékekkel. Az osztályok továbbá lehetőségetadnak, az adatok elrejtéséreí így kívűlről nem lehet eldönteni, hogy mi van az osztály belsejében, ha szeretnénk. Az osztályok örökölhetnek egymástól. Például ha B osztály örököl A osztálytól, akkor B-nek lesznke olyan részei, ami teljesen A-tól származik és, ehhez még hozzájön, hogy B-t hogyan szeretnénk még pontosabbá tenni. Igazából az osztályok olyanok mint egy továbbgondolt struktúra, mert például, ha stuktúrában létrehozunk egy koordináta rendszert, akkor csak a struktúrán kívűl lehet létrehozni függvényt, ami használja azon értékeit. De az osztály használatával ez a probléma megszűnik, mivel függvénykeket is deklarálhatunk benne. Ha adatookat szeretnénk elrejteni, akkor azt a private szóval thetjük meg, ez annyit tesz, hogy az osztályon kívűl nem lehet elérni a változókat/függvényket. Ha azt szeretnénk, hogy elérhető legyen akkor a public kulcsszót kell használnunk. Az osztályokat a konstruktor hozza létre, amit mi is deklarálhatunk magunktól. Ehhez nem kell mást tennünk, mint az osztály nevével létrehozunk egy függvényt. A konstruktor akkor fut le, amikor az osztályból készül egy példány és itt lehet megadni, hogy hogyan mit csináljon az osztály létrehozáskor. Például milyen értékei legyenek, írjon-e ki vaalmit esetleg az osztályon kívűl csináljon valamit vagy hívjon-e meg függvényt stb. A destruktor hasonló a konstruktorhoz, de ez akkor fut le, amikor az adott példány befejezte életét. Ezt is meg lehet adni ugyanúgy mint a konstruktort, de előtte használni kell a ~ jelet. Létezik még másoló konstruktor is, amit arra használunk, hogy osztályok értékeit másoljuk át egy új osztályba. Ezt magunk is megadhatjuk, de ha az osztályban nincs mutató felesleges, hacsak nem akarunk még valami plusz feladatot adni a másoláson kívűl. Mivel a beépített másoló konstruktor addig működik jól, ameddig nem használunk mutatuókat. Korábban említettem, hogy a private szóval lehet adatokat elrejteni, ezzel viszont az a baj, hogy ilyenkor az osztály nem örökli a private-ként megjelölt részeket.

DRAFT

Ha mégis szeretnénk, hogy valamennyire legyeenk elrejtve az adatok és mégis tudjanak örökölni, akkor azt a friend kulcsszóval tehetjük meg. Továbbá lehetőség van static kulcsszóval olyan változót deklarálni, ami minden osztályban ugynaaz lesz és nem tagváltozóként fog létrejönni, hanem egy közös memóriacímen. Viszont ezt tagfüggvényekkel nem oldhatjuk meg. Végül ha nagyon szeretnénk, használhatunk beágyazott definíciót, ami nem tesz mást mint hogy más névvel tudunk adott dolgokra hivatkozni.

Az operátor túlterhelés annyit tesz, hogy a már létező operátorok működését megváltoztathatjuk. Ezt a legegyszerűbben talán másoló értékadással lehet elmagyarázni. Ha már létrehoztunk egy osztályon belűl egy másoló konstruktort, akkor az akkor fog lefutni, ha példányosítjuk és akkor adjuk meg az értékét. De ha már van egy példányunk és azt szeretnénk egy egyenlővé tenni egy másik osztály értékeivel ezt is megoldható. Ehhez nem kell mást tenni mint kiválasztani egy operátort és túlterhelés során hozzáírni a másoló konstruktor lépésit. Például alapesetben az = jel osztályok esetén nem jelent sok mindent, de ha az előbb leírtak alapján túlterhejük akkor működni fog.

A c++ több féle adatfolyamot különböztet meg. Létezik standard bemenet, standard kimenet, hiba kimenet és log kimenet. Ezek szerintem maguktól értetődnek, hogy melyik micsoda. Persze ezeket lehet variálni, hogy hogyan hogyan működjenek. Fájlműveleteket az fstream segítségével tudunk elvégezni. Itt van egy pár lehetőség, hogy hogyan nyissuk meg vagy hogy egyáltalán mit akarunk a fájllal.

Előfordulhat, hogy a program futása során hibakezeléssel kell foglalkoznunk, mert azt akarjuk, hogy valami bizonyos értékkel fusson le vagy csak mert a kódban van olyan rész, ami hibát eredményezhet. Ez lehet például egy számológép esetén 0-val való osztás, ami kifog a cpu-n. Ezen problémák kezelésére tökéletes a try catch blokkok. Ez úgy működik hogy egy try blokkba belerakjuk, azt a kódot, ami hibához vezethet és ha valóban hibát okoz, akkor rögtön ugrik a catch részre, már amennyiben úgy adtuk meg, hogy mindent kapjon el vagy bizonyos kivételeket. Végrehajtja ami ott van és fut tovább a program mintha misem történt volna. Ha például azt akarjuk, hogy beolvasás folyamán bozonyos értéket ne lehessen ezt úgy thejük meg, hogy után throw-al dobunk egy hibát és így ismét a catch rész fog lefutni.

11.2. Java útikalauz programozóknak 5.0

A java nyelv ugyanúgy objektum orientált programozási nyelv mint a c++ és ezért mindkét nyelv ugyanazt a progrtamozási gondolkozást igényli. A két nyelv szintaxisa is elég hasonló. A fontosabb különbségek közé tartozik az, hogy java nem használ pointereket. Helyette mindent referenciaként kezel. C++-hoz hasonlóan itt is van egy main függvény, ami a program indításakor kerül először futattásra, de ehhez java esetén még kötelező a static kulcszót használni. A static igazából csak annyit csinál, hogy nem engedi példányostani, van egy verziónk belőle és kész. Ami még szembetűnő az, hogy java forráskód esetén fordítás után szükség van egy külön programra, hogy a kódunkat tudjuk futattni. Ennek az az oka, hogy míg c++ esetén a compiler gépi kódra fordít, addig java esetén egy köztes nyelvre, amit a a java virtuális géppel tudunk futattni. Ennek az az előnye, hogy így elég egyfajta forráskódot megírnunk, mivel a virtuális gép gondoskodik róla, hogy minden platformon ugyanúgy működjön a kódunk, ahogy mi azt eltervetük.

Változókat tekintve, ha az alap típusokat nézzük illeetve, hogy hogyan tudjuk azokat deklarálni/értéket adni nekik, akkor elmondhatjuk, hogy ugyanúgy működik mint c++ esetén. Ami viszont érdekesebb, hogy javaban nem lehet az operátorokat túlterhelni, csak a függvényeket. Ugyanazokkal a feltételekkel, mint c++-ban, azaaz különböző attribútum számúnak kell lennie és/vagy különböző típusú attrbútomokat kérjen. Java esetén, ha konstansokkal szeretnénk dolgozni, akkor azt a final kulcsszóval tehetjük meg. A konstansok nem mások mint nevesített értékek. Akkor érdemes őket használni, ha van érték, ami többször is előfordul a program írása során és az értke mindig megyegyezik. Erre kitűnő példa például a pi értéke. Előfordúlhat,

DRAFT

hogy többször is szükségünk lesz rá a program futása során és ilyenkor egyszerűbb egy konstanst használni. Java esetén ugyanúgy lehet használni a kommenteket mind c++ esetén, vagyis // az egysoros és /* szöveg */ a többsoros komment. Azonban létezik mégegy fajta komment javaban /** szöveg */ ez esetben ugyanúgy figyelmen kívűl hagyja a kódot, de a dokumentációban meg fog jelenni. Illetve érdemes megjegyezni, ezt a fajta kommentet akármennyi * jellel fel lehet dúzasztani.

Nem meglepő módon javaban az osztályok is meglehetősen hasonlóan működnek. A lényegesebb eltérések öröklődés esetén jönnek be. Valamint példányosítás során használnunk kell a new kulcsszót, hogy a memóriában fogalaljon helyet a változóknak. Osztályok metódusait hasonlóan kell megadni mint c++ esetén. Annyi eltérés viszont van, hogy itt a metódus elején kell megadnunk, hogy milyet szeretnénk public, private, protected, de ezt leszámítva ugyanaz a szintaxisa. Mint c++ esetén itt is van lehetőség kivételkezeésre try catch blokk használtával, ami igazából megint mondhatni teljesen megegyezik. Az itt lévő jelentősebb különbség az, hogy ha szeretnénk kivételt dobni a throw kulcsszóval az szintaxisában kicsit eltérő, de ugyanúgy működik. A kivételkezelés amúgy akkor fontos, ha olyan kódot akarunk futattni, ami hibához vezet. Például ha két számot osztunk és egy felhasználótól kértünk be számokat, akkor figyelnünk kell rá, hogy a nullával való osztás hibához fog vezetni. Ezért ezt a részt belerakjuk egy try blokkba és ha a program futása során hiba történt, akkor azt a catch blokk el fogja kapni és nem fog leállni a programunk.

A java programozási nyelv ugynúgy mint c++ megadja a lehetőségét, annak, hogy többszálon futó programokat írjunk. Java esetén kétféleképpen lehet új szálat létrehozni. Vagy a Thread osztályt örököljük és a megfelelő dolgokat megváltoztatva már a run metódus eghívásával létre is hotuk az új szálat, vagy pedig haszálhatjuk a runnable interface-t, annak érdekében, hogy új szálat hozzunk létre. Valamint lehetőség van állítani az egyes szálaknak a prioritását egy egytől tízes skálán, ahol az alapértelmezett érték az 5.

11.3. Bevezetés a mobilprogramozásba

Kedves naplóm elolvastam a könyv pythonra vonatkozó részét, ami azért volt csodás, mert rengeteg új tudásra tettem szert ezzel a viszonylag új programozási nyelvvel kapcsolatban. Python egy magasszintű programozási nyelv, amit előszerettel használnak prototípúkészítésre, mivel elképesztően gyorsan lehet benne kódokat írni. Valamint maga a nyelv szintaxisa is egyszerű, ami azt eredményezi, hogy a tanulási fázisa meglepően rövid. Valószínűleg ez okból és azért mert gyorsan lehet látványos szintre jutni rengeteg függvénykönyvtár található hozzá. A programoozási nyelv amúgy egy interpreteres nyelv, ami annyit tesz, hogy nincs meg a szokásos forráskód fordítás ciklus mint más nyelveknél és így erre nem is kell várni mivel az interpreter egyből tudja olvasni a python "szkriptet". Python egy igazán hasznos jellemzője továbbá még az is, hogy jól lehet vele használni más nyelven írt modulokat így megnövelve a produktivítást. Ez allat az értendő, hogy a programunknak van egy része, ami elég erősen erőforrás igényes azt például megírhatjuk c++-ban és optimalizálva azt a részt. A c++ modult pedig pythonnal együtt tudjuk használni. Rengeted AI kutatás használja ezt a módszert, ahol maga a az AI core funkciói c++-t használva vannak megírva viszont pythont használva fel lehet gyorsítani a fejlesztést, mivel gyorsan lehet kódokat írni és ez kárpótol a gyengébb teljesítményért.

A python nyelv szintaxisa meglehetősen egyszerű/könnyen olvasható. Például nincs sorok végén ; mint java vagy c/c++ esetén helyette az új ugyanezt a funkciót tölti be. Ha mindenesetben egy sorban szeretnénk két statement-et akkor az új sor helett tabot is használhatunk. Jelentős különbség még az is, hogy például egy függvény esetén, ha azt szeretnénk, hogy maga a függvény törzse több sorból/utasításból álljon, akkor { } közé kellet írnunk a kódokat számos más programozási nyelv esetén. Ez python esetén annyiból áll hogy ezt a kódblokkot elég ha csak beljebb kezdjük. Ezt megtehetjük tab használatával vagy szóközökkel.

DRAFT

Ez eleinte szokatlan lehet, de igazából ennek és az előző tulajdonságnak köszönhetően a python forráskód olvasása hatalmasat nő.

Egy másik sajátossága magának a nyelvnek, hogy habár létezik több változótípus magában a nyelvben azokat automatikusan osztja ki mondhatni kitalálja, hogy milyen változónak amúgy mi is a típusa. Továbbá a változótípusok közötti konverziót is automatikusan kezeli, ami kezdő programozóbaráttá teszi. Maga nyelv olyan téren, hogy milyen típusokat ismer nem mútat semmi újat. Ez alatt azt értem, hogy javaban például megtalálható a pythonban jelenlévő összes típus. Ezek a számok, karakter/karakterlánc, tömbök stb... Hasonlóan javahoz python esetén is a már nem használt szemetet a garbage collector rendezi, így a memóriakezeléssel sem kell foglalkozni. Python esetén is léteznek globális és lokális változók. Függvényen belűl a változók lokalásik, de ha globálissá szerertnénk tenni, akkor azt a global kulcsszóval megtehetjük.

For ciklus python esetén a java féle enhanced for loopnak felel meg vagy C# esetén for each-nek. Ennek igazából csak annyi jelentőssége van, hogy az itt lévő for loopokkal tömbökön tudunk iterálni egyesével. Ha viszont minden esetben szeretnénk mondjuk 1-től 100-ig eljutni egytől eltérő lépéközzel, akkor előbb egy olyan tömböt kell létrehoznunk, ami azokat a számokat tartalmaza, hogy később azon egyesével végig tudjunk menni. Szerencsére range() használatával az összes ilyen jellegű problémát könnyen lehet orvosolni. Mivel python is OOP szemléletet követi, így lehetőség van függvények és classok definiálására a megfelelő kulcsszó használatával. Nem meglepő módon itt is lehetőségünk van hibakezelésre, amit a try except kulcsárossal tudunk elérni. Ez igazából nem szorúl sok magyarázatra, mivel mondhatni ugyanúgy működik mint a try catch java/C++ esetén. Amit érdemes megemlítenni hogy a finally kulcszóval lehetőség van egy kódot lefuttatni mindenképpen, attól függen hogy volt-e kivétel. Ha volt akkor előbb az except blokk fut le majd a finally. Ez akkor lehet lényeges, ha egy fájlba akarunk írni vagy onnan olvasni és közben valamiféle hiba történik, így a finally részben kérhetjük, hogy csuklya be a fájlt.

DRAFT

12. fejezet

Helló, Arroway!

12.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.2. 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...

DRAFT

IV. rész

Irodalomjegyzék

DRAFT

12.3. Általános

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

12.4. C

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

12.5. C++

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

12.6. Lisp

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

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

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

DRAFT