Univerzális programozás

Írd meg a saját programozás tankönyvedet!

Ed. BHAX, DEBRECEN,
2019. május 08, v. 1.0.0

COLLABORATORS			
	TITLE : Univerzális programozás		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Bátfai, Norbert Ã©s Veress, Máté	2019. május 8.	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai
0.0.5	2019-03-03	Első fejezet kész.	theefues
0.0.6	2019-03-11	Második fejezet kész.	theefues
0.0.7	2019-03-18	Harmadik fejezet kész.	theefues
0.0.9	2019-03-26	Negyedik fejezet kész.	theefues

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.1	2019-03.29	Olvasónaplók készen.	theefues
0.1.1	2019-04-02	Ötödik fejezet kész.	theefues
0.1.2	2019-04-10	Hatodik fejezet kész.	theefues
0.1.3	2019-04-24	Hetedik fejezet kész.	theefues
0.1.4	2019-04-30	Kilencedik fejezet kész.	theefues
1.0.0	2019-05-08	Első teljes kiadás.	theefues

Ajánlás

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

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

Tartalomjegyzék

Ábrák jegyzéke

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!

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

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

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

A DocBook XML 5.1 új neked?
Ez esetben forgasd a https://tdg.docbook.org/tdg/5.1/ könyvet, a végén találod az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/tree/master/turing/loops

Tanulságok, tapasztalatok, magyarázat:

Jelen feladat három részből áll; írnunk kell egy olyan végtelen ciklust, amely:

- a gépben található minden processzormagot megdolgoztat 100%-on
- a gépben található összes processzormag közül egyet dolgoztat meg 100%-on
- a gépben található egyik processzormagot sem dolgoztatja

A linkelt repóban megtalálható mindegyik feladat forrásfájla. Ahhoz, hogy minden processzormagot megdolgoztassunk, szükségünk van az OpenMP-re, amit a <omp.h> könyvtárral tudunk includeolni a programunkhoz. Hozzá kell adnunk a #pragma omp parallel {} sort a kódunkhoz, és azon belül kell elhelyeznünk a végtelen ciklusunkat, ami jelen esetben egy while függvény, valahogy így:

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

int main() {
  #pragma omp parallel {  /* ezzel tudjuk kiküldeni a terhelést minden  ↩
     magra */
             while(1){}
         }
         return 0;
}
```

Ezt lefordítva, majd elindítva láthatjuk, hogy a terhelés mindegyik processzormagon egységes. Ahhoz, hogy csak egy mag dolgozzon nincs más teendőnk, mint a fentebb említett függvényt egyszerűen kiszedni a programunkból, ami ez után valahogy így néz ki:

```
int main() {
  while(1){}
  return 0;
}
```

Ezt szintén lefordítva és elindítva láthatjuk, hogy az előzővel ellentétben már csak egy mag dolgozik 100%-on. Az utolsó feladattal jóval egyszerűbb a dolgunk, mint az elsőnél volt, itt ugyanis a végtelen ciklusunkon belül mindössze egy sleep() függvényre van szükségünk, ehhez viszont includeolni kell a <unistd.h> könyvtárat, amiben a sleep függvény deklarálva van. A kódunk ez után így nézhet ki:

```
#include <unistd.h>

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

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.

Tanulságok, tapasztalatok, magyarázat:

Nem tudunk olyan programot írni, ami megmondja egy másikról, hogy le fog-e fagyni, vagy nem, azaz van -e benne végtelen ciklus, vagy sem, mert lehetetlen eldönteni.

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/theefues/prog1/blob/master/turing/varchange.c

Tanulságok, tapasztalatok, magyarázat:

Adott két szám, jelen esetben a és b. Ha a helyére felvesszük az a+b összeg értékét, majd b helyére felvesszük az immáron (a+b)-b eredményét, majd azt visszarakjuk az a változóba úgy, hogy (a+b)-((a+b)-b), akkor felcseréljük a két eredeti változó értékét.

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/theefues/prog1/tree/master/turing/labda

Tanulságok, tapasztalatok, magyarázat:

Fordítás: gcc labda.c -o labda -lncurses

Futtatás: ./labda

A forrás mindkét esetben adott volt. Mindenek előtt, includeolnunk kell a szükséges könyvtárakat. Most az stdio, curses és az unistd könyvtárakra lesz szükségünk. Az if-es módszerrel egyszerű a dolgunk: csak meg kell nézni, hol jár jelenleg a labda. Le kell kérnünk, mekkora a terminál ablakunk. Ezt az initscr() függvénnyel tehetjük meg. Emellé deklarálnunk kell segédváltozókat, ami a labda helyzetének, lépegetési értékének és a terminálablak méretének meghatározásában segít majd minket.

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

int main(void) {
WINDOW *ablak;
ablak = initscr();

int x = 0; /* labda helyzete */
int y = 0;

int xnov = 1; /* lépegetési érték */
int ynov = 1;

int mx; /* terminál mérete */
```

```
int my;
}
```

A végtelen ciklusunkba fel kell vennünk egy getmaxyx() függvényt, így dinamikusan fog működni a programunk, szóval ha változtatunk a terminál méretén, a labdánk hozzá igazodik.

Kirajzoltatjuk a labdánkat a mvprintw() függvénnyel, ahol a már korábban dekralált változókban tároljuk a helyzetét, valamint a karakterét, ami itt egy nagy O betű lesz.

A refresh() függvénnyel frissítjük a programot, hogy a labda ne csak egy helyben álljon, hanem "pattogjon" a képernyőn. Az usleep() függvénnyel pedig a gyorsaságát állíthatjuk. Minél kisebb az érték, annál gyorsabb a labda.

Az x és y értékekhez hozzáadjuk a lépegetési értékeket, ezzel halad előre a labdánk. Végül if függvényekkel megvizsgáljuk, hogy elérte-e a terminálablak széleit és ennek megfelelően irányítjuk a továbbiakban. Negatív értéket adunk neki, így az ellenkező irányba fog tovább haladni és nem megy a határvonalon kívülre.

```
for ( ;; ) {
  getmaxyx ( ablak, my , mx );
  mvprintw ( y, x, "O" );
  refresh ();
        usleep ( 100000 );
  x = x + xnov;
        y = y + ynov;
if ( x>=mx-1 ) {
            xnov = xnov * -1;
        }
        if ( x<=0 ) {
            xnov = xnov * -1;
        }
        if ( y<=0 ) {
            ynov = ynov * -1;
        }
        if ( y>=my-1 ) {
            ynov = ynov * -1;
        }
}
```

Incurses
Figyelnünk kell, hogy fordításnál gcc esetében adjuk hozzá az -lncurses kapcsolót a végére, másképp nem fog lefordulni!

Ugyanez a feladat megvalósítható logikai függvények használata nélkül is: magyarán nincs szükségünk ifekre, hogy megnézzük, mikor csapódik neki a labda az ablak szélének.

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/theefues/prog1/blob/master/turing/bitshift.c

Tanulságok, tapasztalatok, magyarázat:

Fordítás: gcc bitshift.c -o bitshift

Futtatás: ./bitshift

Adott a szó, jelen esetben az 1, ami a b integer változóban van felvéve. A b-t elshifteljük eggyel addig, amíg 0-át nem kapunk, tehát be nem telítjük a memóriánkat, amin túl már nem tudunk többet írni. Így megkapjuk, mekkora egy szó mérete bitben.

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/theefues/prog1/blob/master/turing/pagerank.c

Tanulságok, tapasztalatok, magyarázat:

Fordítás: gcc pagerank.c -o pagerank -lm

Futtatás: ./pagerank

A pagerank egy olyan algoritmus - amelyet például a Google keresőmotorja is használ, a PageRank, mint kifejezés az ő tulajdonukban áll - ami különböző oldalakhoz számokat rendel, ezáltal rangsorolja őket. Arra épít, hogy ha A oldalról átkattintasz hiperlink segítségével B oldalra, akkor az A oldal után a B oldal tartalma is érdekel, tehát egyfajta szavazatként fogható fel a B oldal számára. Minél több hiperlink mutat erre az oldalra, annál fontosabb, így feljebb kerül a rangsorban.

Ebben a feladatban adott 4 oldal, amikre a program kiszámolja a pagerank értékeket. Egy 4 elemű tömbben helyezkednek el az oldalak, mindegyik külön értékekkel rendelkezik.

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

  int i;

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

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

```
  int i;
  double osszeg=0;

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

  return sqrt(osszeg);
}

void
pagerank(double T[4][4]){
  double PR[4] = { 0.0, 0.0, 0.0, 0.0 };
  double PRv[4] = { 1.0/4.0, 1.0/4.0, 1.0/4.0, 1.0/4.0};

  int i, j;

  for(;;){


    for (i=0; i<4; i++){
      PR[i]=0.0;
      for (j=0; j<4; j++){
        PR[i] = PR[i] + T[i][j]*PRv[j];
      }
    }

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


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

  kiir (PR, 4);
}
```

Elkészítünk három funkciót: a kiir, a tavolsag és a pagerank függvényeket. A kiir a pagerank értékek kiírását fogja elvégezni, a tavolsag az oldalak közötti távolság négyzetgyökét adja majd vissza, míg a pagerank kiszámítja nekünk a pagerank értékét a fentebb említett 2 másik függvény segítségével.

A pagerankon belül felbeszünk két, 4 elemű tömböt PR és a PRv néven. A PR minden eleme 0 lesz, igyanis ebbe fogjuk belerakni az oldalak pagerankját később, a PRv pedig a mátrixműveletekben lesz segítségünkre.

Felvesszük az int i és j változókat a for ciklusok változóinak, majd nyitunk egy végtelen ciklust, ami akkor fog kilépni, ha a távolság 0.0000000001-nél kisebb lesz.

A végtelen cikluson belül el kell helyeznünk 2 for ciklust, ami összeszorozza az argumentumokban megadott T mátrixot a PRv tömbbel, ezzel kiszámítva a pagerankot.

Majd ha ezekkel megvagyunk, átrakjuk a PRv értékeit a PR tömbbe, aztán a kiir függvénnyel kiíratjuk.

A mainen belül már nincs más dolgunk, csak deklarálnunk a pagerank mátrixunkat és a pagerank(mátrix) segítségével kiíratni az értékeket.

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

  printf("PageRank\n");
  pagerank(L);

  return 0;
}
```

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

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

```
library(matlab)

stp <- function(x){

    primes = primes(x)
    diff = primes[2:length(primes)]-primes[1:length(primes)-1]
    idx = which(diff==2)
    t1primes = primes[idx]
    t2primes = primes[idx]+2
    rt1plust2 = 1/t1primes+1/t2primes
    return(sum(rt1plust2))
}

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

Ez az R szimuláció bebizonyítja, hogy véges sok számú ikerprím létezik. Minél nagyobbak a számok, annál kisebb az ikerprímek előfordulásának esélye.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

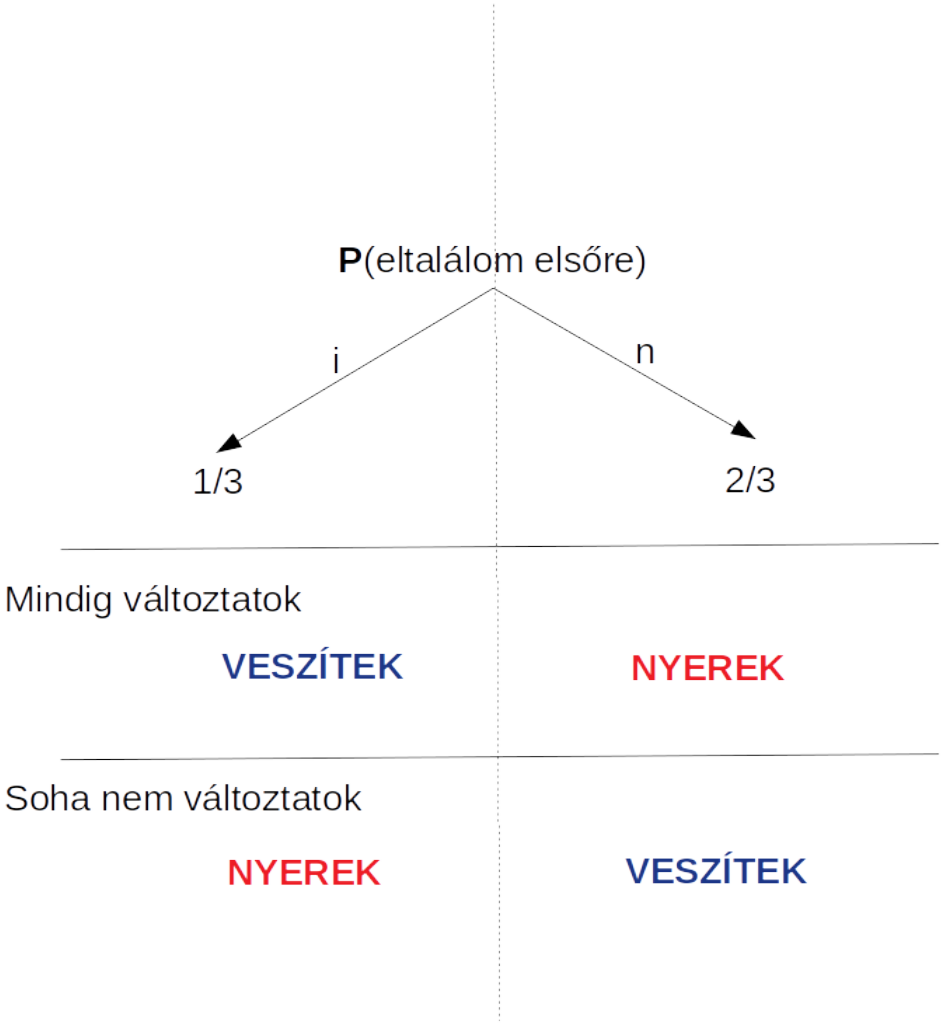
Megoldás videó: https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcsan

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

Tanulságok, tapasztalatok, magyarázat:

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

for (i in 1:kiserletek_szama) {

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

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

    }else{

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

    }

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

}

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

for (i in 1:kiserletek_szama) {

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

}

valtoztatesnyer = which(kiserlet==valtoztat)


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

Adott három ajtó. A háromból 2 ajtó mögött nincs semmi, míg a maradék egy mögött van. Ez a paradoxon azt vizsgálja, hogy megéri-e variálni a nyitás sorrendjén annak érdekében, hogy közelebb kerüljünk a nyereményhez.



2.1. ábra. A Monty Hall paradoxon - Kép: Bátfai Norbert

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/theefues/prog1/blob/master/chomsky/decun.c

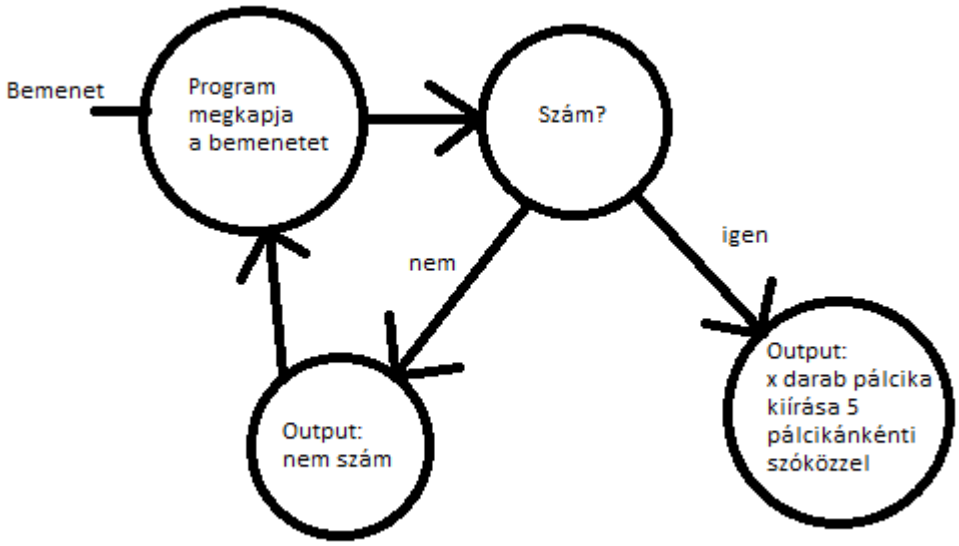
Tanulságok, tapasztalatok, magyarázat:

Tutorált: Dankó Zsolt

Fordítás: gcc decun.c -o decun

Futtatás: ./decun

Az unáris a legegyszerűbb számrendszer. Csak természetes számok ábrázolására alkalmazható, tehát tört számokat nem lehet vele leírni. Egy szám (szám * pálcika) darab pálcika segítségével írható fel, a könnyebb átláthatóság érdekében öt pálcikánkénti jelöléssel, ami a programban a szóköz lesz. Például az egy unárisan |, a kettő ||, a nyolc ||||| ||| és így tovább.



3.1. ábra. A progam működése

```
#include <stdio.h>

int main() {
        int szam;
        printf("Adj meg egy számot: ");
        if (scanf("%d", &szam)) {
                for(int i = 0; i < szam; i++){
                        if(i % 5 == 0 && i != 0)
                        printf(" ");

                        printf("|");
                }
                printf("\n");
        } else {
                printf("Kérlek, számot adj meg.\n");
        }
}
```

Bekérjük a szam változóba a számunkat, majd megvizsgáljuk, hogy tényleg szám-e. Ha nem, hibát írunk ki, ha igen, annyi pálcikát íratunk ki for ciklus segítségével, amennyi a szám értéke. Közben folyamatosan figyeljük, hogy 5 pálcikánként raknunk kell egy szóközt, ami a for ciklus i változójának 5-tel való osztásának maradékával számítható ki. Ha a maradék 0, szóközt rakunk. Ha nem, pálcikát.

```
theefues@fgt_srv:~/prog1$ ./dec
Adj meg egy számot: 23
||||| ||||| ||||| ||||| |||
theefues@fgt_srv:~/prog1$ ./dec
Adj meg egy számot: nem szám
Kérlek, számot adj meg.
```

3.2. ábra. A progam működés közben

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

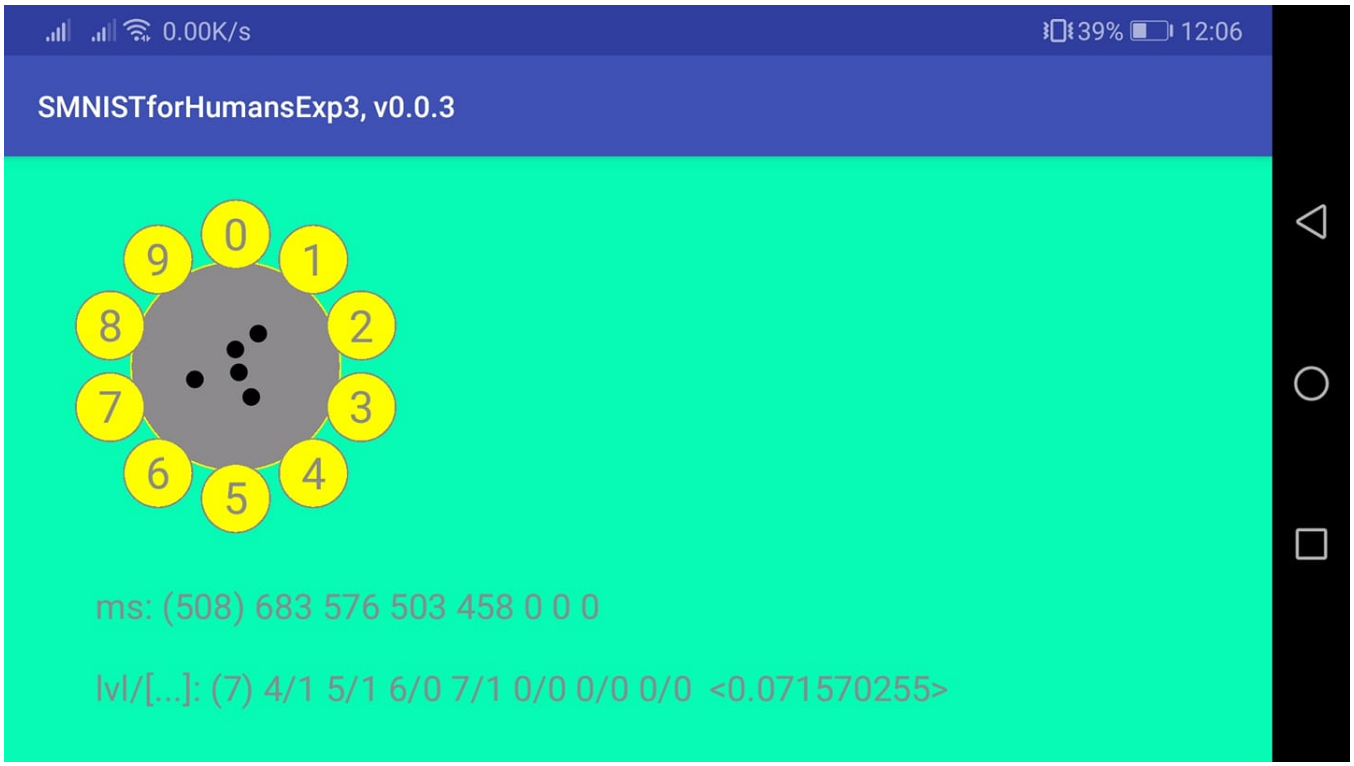
Mutass be legalább két környezetfüggő generatív grammatikát, amely ezt a nyelvet generálja!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Passzolva, +1 passzolási lehetőség az SMNIST miatt:



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/theefues/prog1/blob/master/chomsky/c89.c99

Tanulságok, tapasztalatok, magyarázat:

Az utasítások a leírásuk sorrendjében hajtódnak végre, kivéve akkor, ha valahol jelezzük a kívánr sorrendet. Többféle utasításfajta létezik, például:

Összetett utasítás: Néhány programkörnyezetben a fordítóprogram csak egy utasítást fogad el. Erre jók az összetett utasítások, amik több utasítást fűznek össze eggyé.

Iterációs utasítás: Egy ciklust határoznak meg.

Vezérlésátadó utasítások: A vezérlés feltétel nélküli átadására használatosak.

A for ciklus szintaktikája C89-ben és C99-ben különbözik.

A már jól ismert

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

nem fordul le C89-ben, mivel ott az int i-t a cikluson kívül kell dekralálni, tehát így:

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

Tutor: Dankó Zsolt

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/theefues/prog1/blob/master/chomsky/lex.l

Tanulságok, tapasztalatok, magyarázat:

Tutorált: Dankó Zsolt

Lexelés: lex -t lex.c > lex_l.c

Fordítás: gcc lex_l.c -o lex

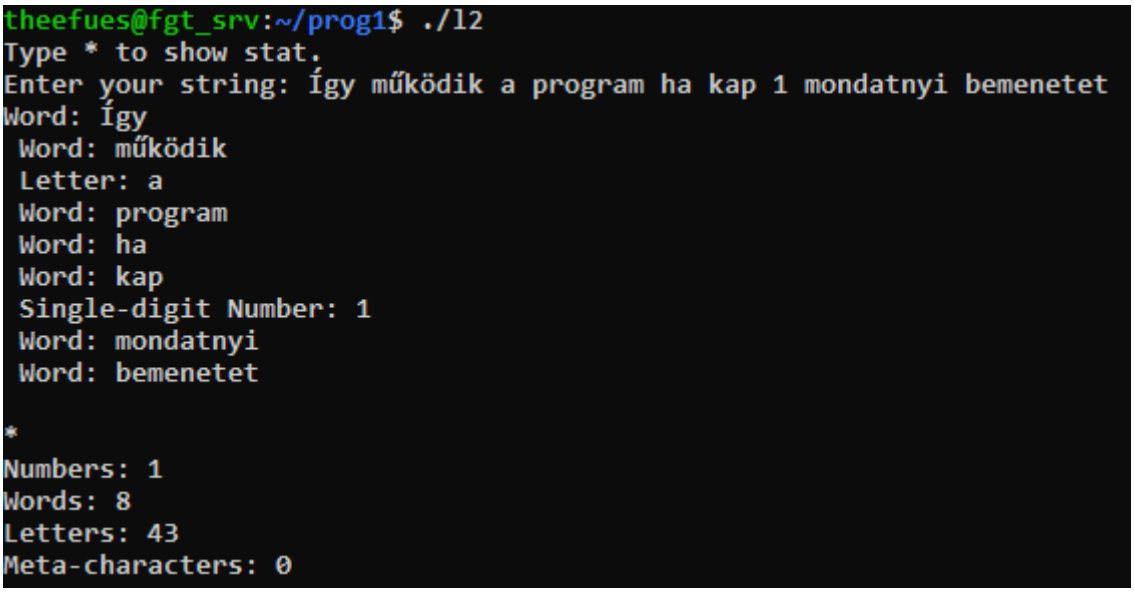
Futtatás: ./lex

A megoldás nagyon egyszerű volt: meg kellett modnanunk a lexerünknek, hogy miként kategorizálja az egyes karaktereket, illetve stringrészeket. Én a számszámoláson kívül raktam még bele több felismerést is.

```
%{
#include <string.h>
int letter_count = 0, word_count = 0, number_count = 0, metacharacter_count ↩
    = 0, stat = 0;
%}
```

```
%%
[0-9]   {++number_count; printf("Single-digit Number: %s\n",yytext);}
[a-zA-ZöüóőúűáéíÖÜÓŐÚÉÁŰÍ]      {++word_count; printf("Letter: %s\n",yytext ↩
   );}
[0-9]+  {++number_count; printf("Multi-digit Number: %s\n",yytext);}
[0-9]+\.[0-9]+? {++number_count;
        printf("Fraction: %s\n",yytext);}
[a-zA-Z0-9öüóőúűáéíÖÜÓŐÚÉÁŰÍ][a-zA-Z0-9öüóőúűáéíÖÜÓŐÚÉÁŰÍ]*     {++ ↩
   word_count; letter_count += strlen(yytext); printf("Word: %s\n",yytext) ↩
   ;}
[\-/._,$><'"+:=?!%()#&@] {++metacharacter_count;
                printf("Meta-character: %s\n", yytext);}
[*] {++stat; printf("Numbers: %d\nWords: %d\nLetters: %d\nMeta-characters:  ↩
   %d\n",number_count,word_count,letter_count,metacharacter_count);}
%%
int main() {
        printf("Type * to show stat.\nEnter your string: ");
        yylex();
        printf("All occured numbers: %d\n", number_count);
        return 0;

}
```

A fentebbi kódrészlet így értelmezhető: a %{-en belül includeoljuk a C nyelvben már ismert könyvtárat, majd felvesszük a változóinkat, amik számolni fogják nekünk, melyik karakterekből, illetve szavakból mennyi fordult elő. A %% utáni rész a lényeg. Itt megadjuk a lexerünknek reguláris kifejezések segítségével, hogy milyen karaktercsoportokra mit adjon válaszul. Az első az egyjegyű számokat veszi figyelembe, tehát ami 0-9 között előfordul. A második a - magyar nyelv magánhangzóit is belevéve - betűket figyeli. Azokat a karaktereket, amelyek a felsorolásban szerepelnek, de nem áll sem előttük, sem utánuk semmi a szóközön kívül. A harmadik a többjegyű számokat figyeli, míg a negyedik a törtszámokat. Ez lehet például 3.14, de lehet a 21412412412.345436346 is. Nem számít, hány jegy áll a pont előtt vagy után. Az ötödik a szavakat veszi figyelembe. Azokat az egymás után következő karaktereket, amik nem számok és nem is metakarakterek. A 4 például egy szám, de a P4 már szó. Az utolsó, azaz a csillag pedig kiíratja a statunkat, hogy a jel beírásáig hol tartanak az egyes számlálóink. Ezzel nem kell CTRL+D-vel lezárnunk a programunkat, hogy megtudjuk, mennyi számunk van eddig. Az mainen belül pedig felvesszük az yylex() függvényt, ami a kódunk lexelése és fordítása után működtetni fogja a gépezetet.



3.3. ábra. A progam működés közben

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/chomsky/l33t.l

Tanulságok, tapasztalatok, magyarázat:

Össze kell állítanunk egy olyan .l fájlt, amit a lexer értelmezni tud. Első sorban includeolnunk kell a könyvtárakat, amiket használni fogunk a későbbiekben, majd defineolnunk kell a L337SIZE-t, ami elosztja a l337dlc7 tömb méretét a cipher struktúra méretével, ezzel megkapjuk a string hosszát.

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

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

Ez után felvesszük a fentebb említett tömböt és struktúrát. A cipheren belül a c char típusú változó az adott stringet fogja tárolni, a leet pointer pedig a 1337dlc7 tömbből az adott karakter leet megfelelőjét. Felvesszük a l337dlc7 nem előre definiált méretű tömb értékeit, vagyis az angol ábécé minden betűjéhez hozzárendel 4 karaktert, amiből random választ majd egyet nekünk a lexer.

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

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

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

  };

%}
```

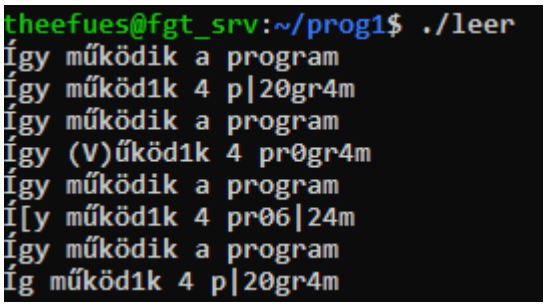
Ha ezzel megvagyunk, fel kell vennünk a szabályokat, hogy mivel mit csináljon a lexer. Mivel nem szavakat keresünk, hanem karaktereket szeretnénk átírni leet-re, ezért elég, ha csak a .-nak, azaz egy, bármilyen tetszőleges karakterhez rendelünk szabályt. Felvesszük a found integer változót, majd létrehozunk egy for ciklust, ami végig fog menni a stringünk minden karakterén, és a szabálynak megfelelően helyettesíti be őket a tömbben már deklarált értékre. A cikluson belül felveszünk egy if függvényt, ami megnézi, hogy a karakterünk szerepel-e a tömbben, és egy r random változót, ami 0 és 100 között generál egy számot. Ez

abban lesz segítségünkre, hogy eldöntse, a tömb melyik elemére cseréljük ki a karakterünket. Pl. ha az r kisebb, mint 91, akkor a tömb első, azaz a [0] elemét válassza ki. Az a betű esetében ez a 4-es lesz. Ha a szám 95-nél kisebb, akkor a második elemét és így tovább. Ha végzett ezzel, átírja a found változó értékét 1-re, tehát nem üresjáratban ment a for ciklusunk, hanem talált is valami neki megfelelőt az inputból. Ennél fogva, ha found nem 0, írja ki, milyen a már átalakított stringünk.

```
%%
. {

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

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

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

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

        found = 1;
        break;
      }

    }

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

  }
```

A mainen belül pedig csak simán behívjuk a yylex() függvényt, majd egy retrun 0-át, hogy a program sikeresen lefutott

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



3.4. ábra. A progam működés közben

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: https://github.com/theefues/prog1/blob/master/turing/signal.c

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat:

Fordítás: gcc signal.c -o signal

Futtatás: ./signal

A signal manuáljában láthatjuk, hogy 2 értékre van szüksége a működéshez: egy signalra és egy handlerre.

A signal jelen esetben adott, a 2-es, azaz a terminate signal. Linux alatt ezt a CTRL+C billentyűkombinációval küldhetjük ki a porgramnak. A jelkezelo kezelő függvényét nekünk kell megírnunk. Ezt a main-en kívül, voiddal tudjuk megtenni úgy, hogy a függvényünk kérjen egy int változót, amit elneveztem sig_num-nak. Ez fogja tárolni az elkapott signal számát. A függvényünkün belül printf függvénnyel kiíratjuk, mit mondjon a program, ha le akarjuk lőni: Elkapva %d\n. A sortörés nélkül nekem nem írt ki semmit.

```
void jelkezelo(int sig_num) {
        printf("Elkapva %d\n", sig_num);
}
```

Itt a %d a sig_num decimális alakja lesz, vagyis a 2. Ha ezekkel megvagyunk, nincs más dolgunk, mint végtelen ciklusba rakni a forrásanyagban feltüntetett if függvényt.

```
for(;;) {
  if(signal(SIGINT, jelkezelo)==SIG_IGN)
  signal(SIGINT, SIG_IGN);
}
```

A kóddal immár megvagyunk, utolsó teendőként lefordítjuk, majd elindítjuk és megnézzük, jól csináltuk-e, amit csináltunk. Ha minden sikerült, a CTRL+C lenyomásakor az Elkapva 2 üzenetet kell kapnunk.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

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

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

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

Tanulságok, tapasztalatok, magyarázat:

Az 1) azt mondja, hogy sok prímszám létezik. Kiolvasva: Minden x-re van olyan y, hogy x kisebb mint y és akkor y prím.

A 2) azt mondja, hogy sok ikerprímszám létezik. Kiolvasva: Minden x-re van olyan y, hogy x kisebb mint y, és y és SSy ikerprím.

A 3) és a 4) is azt mondja, hogy véges sok prímszám létezik. 3) Kiolvasva: Van olyan y, amire minden x prím, ebből következik, hogy x kisebb mint y.

4) Kiolvasva: Van olyan y, amire minden x nagyobb mint y, ebből következik, hogy x nem prím.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

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

Mit vezetnek be a programba a következő nevek?

- Egész integer

```
int a;
```

- Egész integerre mutató mutató

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

- Egész integer referenciája

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

- 5 elemű üres c tömb

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

- 5 elemből álló tömb referenciája

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

- 5 elemből álló tömbbre mutató mutató

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

- Egy függvényre mutató mutató

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

- Egy mutató függvényre mutató mutató függvény

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

- Egy egészet visszaadó és két egészet kapó mutató függvény

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

- Egy függvényre mutató függvény mutató, ami egy egészet ad vissza és két egészet kap

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

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/chomsky/declaration.c

Tanulságok, tapasztalatok, magyarázat:

A mainbe téve a fentebb leírt deklarációkat, a program sikeresen lefut.

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/caesar/double.c

Tanulságok, tapasztalatok, magyarázat:

Fordítás: gcc double.c -o double

Futtatás: ./double

A háromszögmátrixunk 5 sorból fog állni.

```
#include <stdio.h> #include <stdlib.h>
int main () { int nr = 5; double **tm;
printf("%p\n", &tm);
if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL) {
    return -1;
}
```

Includeoljuk a szükséges könyvtárakat, majd a mainen belül felveszünk két változót, az int nr-t, ami a sorok számát adja meg, illetve a dobule **tm változónkat. Ez után kiíratjuk a tm változó tartalmát. Egy if függvénnyel megvizsgáljuk, van-e hely a memóriában a további mu˝veletekhez. Ehhez a malloc függvényt használjuk, ami egy pointert fog nekünk visszaadni. Értékét a nr értékével és a double * változó értékével kell szorozni. Ha nincs elég memória, a progam kilép -1-es hibakóddal. Ha van, megy tovább.

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

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

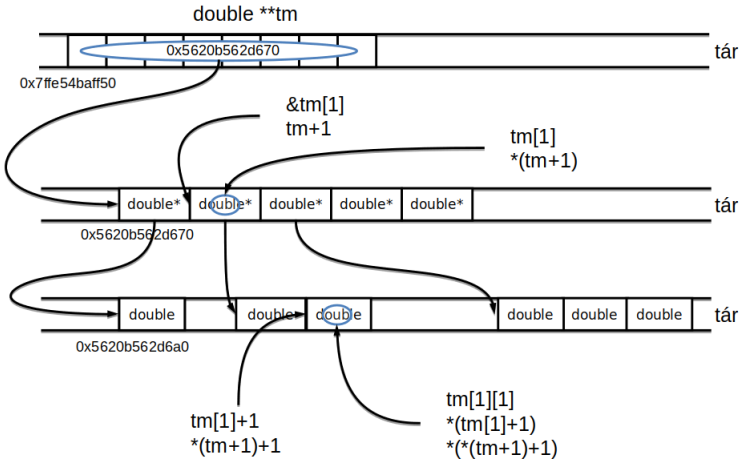
```
    }

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

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

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

Kiírjuk a tm változó értékét. Az els˝o for ciklusunkban addig megyünk, amíg az nr, azaz a sorok száma tart, jelen esetben 5. Mindig eggyel arrébb mutatunk a memóriában, ahol eggyel több double * számára van lefoglalva a hely így az első sorban 1 érték szerepel, a másodikban már 2, a harmadikban három és így tovább, amíg a for ciklus igaz. A tm[] tömbben ugrálunk mindig eggyel nagyobb érték felé. Ha útközben betelik a memória, a program kilép.

A következ˝o for ciklus el˝ott kiírjuk a tm[] tömb 0-ás, azaz első elemét. Ez után elkészítjük a mátrixunkat, ugyanúgy a nr változós for ciklus segítségével, majd ki is íratjuk őket.

```
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)
        free (tm[i]);

    free (tm);

    return 0;
}
```

Ha mindennel megvagyunk, felszabadítjuk a tm által lefoglalt memóriaterületet, és nullával kilépünk a programmal.



4.1. ábra. Double** háromszögmátrix a memóriában - Kép: Bátfai Norbert

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/caesar/exorc/exor.c

Tanulságok, tapasztalatok, magyarázat:

Fordítás: gcc exor.c -o exor

Futtatás: ./exor kulcs < tiszta.szoveg > titkos.szoveg

A C exor titkosítónk működése a kulcs alapján történő karakterkódolás. A kulcsban szereplő karakterek alapján a titkosító program összekeveri a karaktereket, így egy érthetetlen kódhalmazt kapunk a tiszta szöveg helyett.

A kódunkban includeolnunk kell a szükséges könyvtárakat és definiálnunk kell a továbbiakban használatos kifejezéseket. Ezek a MAX_KULCS és a BUFFER_MERET. A max kulcs a megadható kulcs hosszát, míg a BUFFER_MERET értelem szerűen a buffer méretét adja meg.

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

#define MAX_KULCS 100
#define BUFFER_MERET 256
```

Mivel a programunk argumentumokkal dolgozik, a mainben ezt jeleznünk kell neki. Deklarálnunk kell az argc integer változót, valamint a **argv char típusú pointert.

A mainen belül fel kell vennönk a kulcs és buffer változókat. Ezek értékét a már fentebb definiált MAX_KULCS és BUFFER_MERET határozza meg.

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

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

  int kulcs_index = 0;
  int olvasott_bajtok = 0;

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

Az int kulcs_meret változóban az inputban megadott kulcs hosszát olvassuk be, majd összehasonlítjuk a MAX_KULCS-csal.

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

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

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

  }

      write (1, buffer, olvasott_bajtok);

    }
}
```

A while ciklusunkban beolvassuk az inputfájlt bájtonként a bufferbe, a BUFFER_MERET értékén belül.

A for cikluson belül végigmegyünk a buffer tartalmán és mindegyiket exorosan titkosítjuk a megadott kulcs alapján.

Ha megvan, kicseréljük az eredeti buffert az új, titkosított bufferre és kiadjuk az outputnak.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/caesar/exor.java

Tanulságok, tapasztalatok, magyarázat:

Tutor: Iloshvai Áron

Fordítás: javac exor.java

Futatás: java exor.java kulcs < tiszta.szoveg > titkos.szoveg

Az exor titkosító hasonlóképp működik javaban, mint c-ben. Annyi a különbség, hogy itt az exor függvényt class-ba kell rakni, míg C-ben nem volt muszáj.

```
public class ExorTitkosito {

    public ExorTitkosito(String kulcsSzoveg,
            java.io.InputStream bejovoCsatorna,
            java.io.OutputStream kimenoCsatorna)
            throws java.io.IOException {

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

        while((olvasottBajtok =
                bejovoCsatorna.read(buffer)) != -1) {

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

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

            }

            kimenoCsatorna.write(buffer, 0, olvasottBajtok);

        }

    }
```

ExorTitkosito néven létrehozunk egy publikus class-t, azon belül pedig felvesszük az ExorTitkosito függvényt, ami a kulcsSzoveg stringet, az input és output csatornákat fogja tartalmazni.

Innentől minden ugyanaz, mint a C-ben, tehát beolvassuk a kulcsot valamint a buffert, exorozzuk majd elmentjük a titkosított szöveget.

```
    public static void main(String[] args) {

        try {

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

        } catch(java.io.IOException e) {

            e.printStackTrace();
```

```
        }

    }

}
```

Innentől más dolgunk nincs, mint a mainben meghívni a fenti classt és megcsinálni az inputtal a titkosítást. A try-catch blokkal pedig elkapjuk az esetleges hibákat.

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/theefues/prog1/blob/master/caesar/exorc/toro.c

Tanulságok, tapasztalatok, magyarázat:

Fordítás: gcc toro.c -o toro

Futtatás: ./toro < titkos.szoveg > tistza.szoveg

Ha elfelejtettük a titkosításnál megadott kulcsot, akkor ez a program végigmegy az összes lehetséges kulcson, hogy megtalálja a megfelelőt. Ennek megkönnyítésére bevezetünk pár funkciót.

```
#define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 5
#define _GNU_SOURCE

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

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

  return (double) titkos_meret / sz;
}

int
tiszta_lehet (const char *titkos, int titkos_meret)
{

  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");

}

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

  int kulcs_index = 0;

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

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

    }

}

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

  exor (kulcs, kulcs_meret, titkos, titkos_meret);

  return tiszta_lehet (titkos, titkos_meret);

}
```

Először defineoljuk a szükséges kifejezéseket, mint a kulcsméret, olvasási buffer és a beolvasható titkos szöveg maximális mérete. Ez után includeoljuk a szükséges könyvtárakat, majd elkezdjük megírni a funkciókat.

Az atlagos_szohossz megnézi, mekkora a szövegben előforduló szavak átlag hosszúsága, ezt visszaadja egy double változóként.

A tiszta_lehet funkció lecsökkenti a lehetséges kulcsok számát, ezzel is felgyorsítva a folyamatot. Ez jelenleg csak a magyar nyelvre van optimalizálva

Az exor próbálja feltörni a szöveget, az exor_tores pedig megnézi, hogy sikerült-e neki.

```
int
main (void)
{

  char kulcs[KULCS_MERET];
  char titkos[MAX_TITKOS];
```

```
  char *p = titkos;
  int olvasott_bajtok;

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

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

  // osszes kulcs eloallitasa
  for (int ii = 'A'; ii <= 'Z'; ++ii)
    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);
      }

  return 0;
}
```

A mainben behívjuk a fentebb már megírt függvényeket a megfelelő helyre. Beolvassuk a titkos fájlunk tartalmát a bufferbe, majd végigmegyünk az összes lehetséges kulcson, közben folyamatosan ellenőrizzük a kimenetet, hogy tiszta-e már a szövegünk, vagyis megtalálhatóak benne a tista_lehet funkción belül felvett szavak.

4.5. Neurális OR, AND és EXOR kapu

R

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

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

Tanulságok, tapasztalatok, magyarázat:

A neurális hálózat logikai kapuja egy vagy több bemeneti értéket kap, majd azokra elvégzi az adott műveletet, ami után ad egy kimeneti, szintén logikai értéket. Ezzel a kimeneti adatot egyből, konvertálás nélkül továbbíthatjuk egy másik logikai kapunak, így egy bonyolult és összetett logikai rendszert építhetünk fel a segítségével. Összesen hétféle kapu létezik: AND OR NOT NAND NOR EXOR EXNOR, ezek csak kétféle kimenetet adhatnak. Ebből nekünk csak az OR, tehát a vagy, az AND, tehát az és, valamint az EXOR, azaz a kizáró vagyra van szükségünk a feladatban.

XOR

b \ a	1	0
1	0	1
0	1	0

AND

b \ a	1	0
1	1	0
0	0	0

XOR

b \ a	1	0
1	1	1
0	1	0

4.2. ábra. AND, OR és EXOR igazságtáblái

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/caesar/perceptron.cpp

Tanulságok, tapasztalatok, magyarázat:

A perceptron a gépi tanulásban a felügyelt tanulással kapcsolatos algoritmus. Megfelelő tanítás után képes arra, hogy két, lineárisan szétválasztható bemeneti mintahalmazt kettészedjen.

Felépítéséből láthatóan egy lineáris kombinációt valósít meg, amelynek kimenetén egy küszöbfüggvény szerepel, segítve ezzel a hibaképzést. Forrás: Wikipedia

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

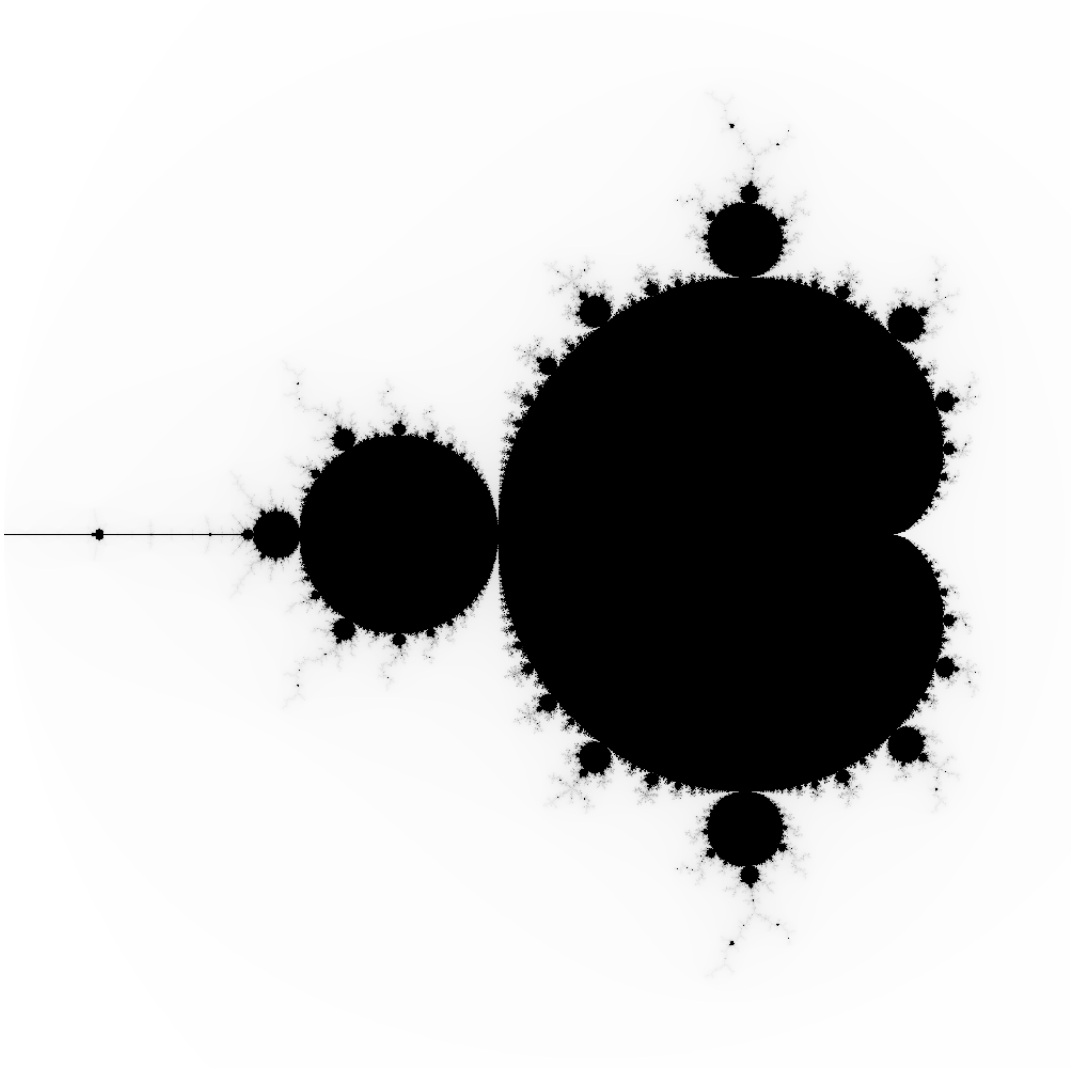
Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/mandelbrot/halmaz.cpp

A Mandelbrot-halmaz azon c komplex számok síkja, amelyekre igaz ez a sorozat, hogy $x_1 = c$ és $x_{n+1} = (x_n)^2 + c$

A komplex számsíkon ábrázolva úgynevezett fraktálalakzatot kapunk. Ezek az alakzatok elsőre összevisszának tűnhetnek, ám aprólékosabban megnézve észrevesszük, hogy komoly matematikai számítások eredményei, tehát minden valamilyen adott szabály alapján történik. A halmaz a nevét felfedezőjéről, Benoît Mandelbrotról kapta.

Ábrázolása nagy teljesítményt igényel, így csak számítógépes környezetben generálható. Ha a kép nem statikus, minél jobban belenagyítunk, annál kuszább, ám szabályszerű alakzatokat láthatunk, amik valahogyan hasonlítanak az eredeti alakzatra.



5.1. ábra. A Mandelbrot-halmaz

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

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/mandelbrot/complex.cpp

Fordítás: g++ complex.cpp -o complex -lpng

Futtatás: ./complex fájlnév szélesség magasság n a b c d

Ez a program a megadott paraméterekkel kiszámolja a Mandelbrot halmazt, és elmenti egy .png fájlba.

Ehhez szükségünk van a png++ könyvtárra is. Így az iostream és a complex mellett azt is includeoljuk, miután letöltöttük.

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

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

  int szelesseg = 1920;
  int magassag = 1080;
  int iteraciosHatar = 255;
  double a = -1.9;
  double b = 0.7;
  double c = -1.3;
  double d = 1.3;

  if ( argc == 9 )
    {
      szelesseg = atoi ( argv[2] );
      magassag =  atoi ( argv[3] );
      iteraciosHatar =  atoi ( argv[4] );
      a = atof ( argv[5] );
      b = atof ( argv[6] );
      c = atof ( argv[7] );
      d = atof ( argv[8] );
    }
  else
    {
      std::cout << "Hasznalat: " << argv[0] << "fajlnev szelesseg magassag  ↩
         n a b c d" << std::endl;
      return -1;
    }
```

Mivel a programunk megadott adatokkal fog dolgozni, a mainben fel kell vennünk az argv* és argc változókat, ez után pedig deklarálnunk kell az alapértelmezett értékeket a szükséges vátlozókhoz.

szelesseg, magassag, iteraciosHatar : A kép szélessége, magassága és a határ, hogy meddig mehet el a rajzolás.

a, b, c, d : az x és y tengely minimum és maximum értékei.

Ha minden argumentum megvan az inputban, helyettesítse be azokra az értékekre. Ha nincs, írja ki, mit kellene csinálnia, majd kilép a programból. Az argv[0] itt a legenerált fájlnevünk lesz. Tehát ha például manként mentettük el a lefordított fájlt, akkor a ./man -al tudjuk futtatni, tehát az argv[0] értéke ./man lesz.

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

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

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

A korábban felvett magassag és szellesseg változókkal megmondjuk a png++ könyvtárnak, hogy mekkora legyen a kép mérete.

a dx és az dy a magasság, illetve szélesség függvényében fog egyre jobban terjeszkedni. Minél nagyobb a kép mérete, annál pontosabb képet kapunk a halmazról: annál jobban bele lehet zoomolni.

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

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

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

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

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

              ++iteracio;
            }

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

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

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

}
```

Két forciklussal végigmegyünk a sorokon és az oszlopokon, és kiszínezzük az iteráció változó segítségével kapott színnel azokat a kapott komplex számokat a síkon, amik megfelelnek a Mandelbrot halmaz alapjául szolgáló sorozatnak.

A folyamat haladását tudatnunk kell a felhasználóval, így százalékértéket íratunk ki neki, hogy jelenleg hol jár a render.

Ha midnen sikeresen lefutott, elmentjük a képet az argv[1] helyen megadott érték nevén.

5.3. Biomorfok

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

Megoldás forrása: https://github.com/theefues/prog1/blob/master/mandelbrot/biomorf.cpp

Tanulságok, tapasztalatok, magyarázat:

Fordítás: g++ biomorf.cpp - o biomorf -lpng

Futtatás: ./biomorf fajlnev szelesseg magassag n a b c d reC imC R

A program az előző feladat kódján alapul, kisebb módosításokkal.

```
int szelesseg = 1920;
int magassag = 1080;
int iteraciosHatar = 255;
double xmin = -1.9;
double xmax = 0.7;
double ymin = -1.3;
double ymax = 1.3;
double reC = .285, imC = 0;
double R = 10.0;
```

Az előző kódban az xmin, xmax helyett a és b volt, az ymin és ymax helyett pedig c és d. Ezeket átítruk, majd hozzáadunk további 3 változót, az reC-t, az imC-t és az R-t. Az reC a C szám valós részét fogja tárolni, az imC pedig a C szám imaginárius részét, így az visszavetíthető a komplex számsíkon. Az R egy tetszőleges valós szám.

Továbbá ki kell még bővítenünk az argumentumtartományunkat is az új változók függvényében:

```
if ( argc == 12 )
{
    szelesseg = atoi ( argv[2] );
    magassag =  atoi ( argv[3] );
    iteraciosHatar =  atoi ( argv[4] );
    xmin = atof ( argv[5] );
    xmax = atof ( argv[6] );
    ymin = atof ( argv[7] );
    ymax = atof ( argv[8] );
    reC = atof ( argv[9] );
    imC = atof ( argv[10] );
    R = atof ( argv[11] );
```

A 9-10-11 számú argumentumoknak felvesszük az imént említett 3 változót, valamint növeljük az if függvényben az argc számát, mivel immár 9 helyett 12-vel fogunk dolgozni.

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

Hozzáadjuk ezt a komplexizáló függvényt, ami az reC és az imC-t veszi alapul.

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

A második for cikluson belül volt az előző programban az reC = a + k * dx. Mivel ezeket megváltoztattuk, itt is átírjuk őket a neki megfelelőre annyi különbséggel, hogy reC helyett reZ-t, valamint imC helyett imZ-t írunk és ezeknek vesszük a komplexét.

```
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 ));
        }
```

Az előző programban a while ciklust is átírtuk for ciklusra. Azon belül a z_n változót a harmadikra emeljük, majd hozzáadjuk a cc-t, ami az reC és az imC komplexe. Ha z_n valós vagy imaginárius része nagyobb mint R, akkor álljon le a for ciklus, és az iteráció legyen a ciklus i változója. Ezt követően csak le kell generálnunk a png képet és az iteráció változó segítségével az adott módon beszínezni. A program többi része innentől nem változik.

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/mandelbrot/cuda.cpp

Fordítás: g++ cuda.cpp -o cuda -lpng

Futtatás: ./cuda fájlnév

Ez a program a videokártyánk CUDA magjait használva generál egy Mandelbrot halmazt. Minél több CUDA magunk van, annál gyorsabb a képgenerálás. Ezt a technológiát az NVIDIA szabadalmaztatta a grafikai feladatok gyorsabb végrehajtása érdekében.

A programunk így épül fel:

Includeolnunk kell a szükséges könyvtárakat. Itt is, mint az eddigi programoknál, szükségünk lesz a png++ könyvtárra. Defineolnunk kell még továbbá a MERET és ITER_HAT változókat, amelyeket a továbbiakban használni fogunk.

Felveszünk egy mandel nevű void funkciót, amely egy mátrixot kér be, amely terjedelme a MERET változó értéke lesz, jelen esetben 600x600. Ez lesz a képünk felbontása.

A funkción belül felveszünk 2 időmérőt, amely megmondja nekünk, mennyi idő volt a kép legenerálása másodpercben.

Deklaráljuk az előző programokból már jól ismert a, b, c és d változókat fix értékkel, valamint a szelesseg, magassag és iteraciosHatar változókat, amelyekhez a fentebb defineolt változók értékeit rendeljük.

```
#include <iostream>
#include "png++/png.hpp"
#include <sys/times.h>

#define MERET 600
#define ITER_HAT 32000

void
mandel (int kepadat[MERET][MERET]) {
    clock_t delta = clock ();
    struct tms tmsbuf1, tmsbuf2;
    times (&tmsbuf1);

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

Fel kell vennönk a dx, dy értékeket, amelyből a dx a b és a változók különbségének a szelesseg változóval osztott hányadosa, a dy pedig a d és c különbségének a magassag változóval osztott hányadosa. Deklaráljuk továbbá az reC, imC, reZ, ujreZ és ujimZ változókat, valamint az iteráció számontartó változót.

2 for ciklus segítségével végigmegyünk az oszlopokon vagy sorokon. A belső for cikluson belül elkezdjük ábrázolni a komplex számokat a síkon, így kirajzolva a Mandelbrot halmazt. Addig keressük a pontokat, amíg a Z abszolút értéke kisebb, mint kettő, vagy nem érjük el a 255-ös iterációs határt. Ha ezen kritériumoknak megfelel az adott pont, akkor az a halmaz eleme, így ábrázolni kell. Továbbiakban kiírjuk még az eltelt időt, ezzel véget is ért a mandel funkció.

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

    int iteracio = 0;

    for (int j = 0; j < magassag; ++j)
    {
        for (int k = 0; k < szelesseg; ++k)
        {
            reC = a + k * dx;
            imC = d - j * dy;
            reZ = 0;
            imZ = 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;
```

```
            }

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

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

    delta = clock () - delta;
    std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << std::endl;

}
```

A main argumentumai között fel kell vennünk az argc és *argv[] változókat, hiszen bemenettel dolgozunk. Ha nincs bemenet, tudatjuk a felhasználóval és kilépünk a programból. Ha van, felvesszük a kepadat változót, amely egy 600x600-as mátrix. Meghívjuk a mandel függvényünket, majd a png++ könyvtár segítségével elkezdjük legenerálni a képünket. Ha megvagyunk vele, kiírjuk az eltelt időt, valamint azt, hogy az outputfájl mentésre került. Ha minden jól sikerült, a kimeneti fájlunk egy 600x600-as Mandelbrot halmazt ábrázoló png kép.

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://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/Qt/Frak/

A program futtatásához szükségünk van a libqt4-dev csomagra, amit Linux alatt az apt install libqt4-dev paranccsal tölthetünk le.

Ezt követően kiadjuk a qmake parancsot abban a mappában, amiben a fájlainkat elhelyeztük, ami legenerál nekünk egy Makefile-t. Ezt a make paranccsal lefuttatjuk. Ha ezzel megvagyunk, a ./Frak paranccsal tudjuk futtatni a programunkat.

Így kapunk 4 ablakot, amiken különböző fraktálalakzatok láthatóak, 3 közülük random helyeken bezoomolt, kinagyított kép az eredeti halmazból.

6 fájlunk van: Frak.pro - a projekt fájl a Qt-hoz, amiben dolgozunk - , frakszal.cpp és frakszal.h - ami a Mandelbrot halmazt rajzolja ki nekünk - , frakablak.cpp és frakablak.h - ami az ablakot rajzolja meg nekünk -, valamint a main.cpp-t, ami segítségével a nagyításokat is kirajzoltatjuk.

frakablak.h

Ebben tároljuk a FrakAblak osztályt, ami a mandelbrot halmazt rajzolja ki nekünk.

Inclideoljuk a szükséges Qt könyvtárakat, valamint a frakszal.h headerfájlt, ami a FrakSzal osztályt fogja tartalmazni. A FrakAblak osztály tartalmaz egy FrakAblak funkciót, valamint egy vissza és egy paontEvent funkciót is.

```
        #ifndef FRAKABLAK_H
#define FRAKABLAK_H

#include <QMainWindow>
#include <QImage>
#include <QPainter>
#include "frakszal.h"

class FrakSzal;

class FrakAblak : public QMainWindow
{
    Q_OBJECT

public:
    FrakAblak(double a = -2.0, double b = .7, double c = -1.35,
              double d = 1.35, int szelesseg = 600,
              int iteraciosHatar = 255, QWidget *parent = 0);
    ~FrakAblak();
    void vissza(int magassag , int * sor, int meret, int hatar) ;

protected:
    void paintEvent(QPaintEvent*);

private:
    QImage* fraktal;
    FrakSzal* mandelbrot;

};

#endif
```

frakaszal.h

Ebben tároljuk a FrakSzal osztályt, ami a nagyított mandelbrot halmazt rajzolja ki nekünk.

Inclideoljuk a szükséges Qt könyvtárakat, valamint a frakablak.h headerfájlt, ami a FrakAblak osztályt fogja tartalmazni. A FrakSzal osztály tartalmaz egy FrakSzal funkciót, valamint az összes szükséges változót, protected formában, tehát csak az osztályon belül férhetünk hozzájuk.

```
#ifndef FRAKSZAL_H
#define FRAKSZAL_H

#include <QThread>
#include <QImage>
#include "frakablak.h"

class FrakAblak;

class FrakSzal : public QThread
{
```

```
    Q_OBJECT

public:
    FrakSzal(double a, double b, double c, double d,
             int szelesseg, int magassag, int iteraciosHatar, FrakAblak * ↩
                frakAblak);
    ~FrakSzal();
    void run();

protected:
    // A komplex sík vizsgált tartománya [a,b]x[c,d].
    double a, b, c, d;
    // A komplex sík vizsgált tartományára feszített
    // háló szélessége és magassága.
    int szelesseg, magassag;
    // Max. hány lépésig vizsgáljuk a z_{n+1} = z_n * z_n + c iterációt?
    // (tk. most a nagyítási pontosság)
    int iteraciosHatar;

    FrakAblak* frakAblak;
    int* egySor;

};

#endif
```

frakaszal.cpp

Ebben rajzoljuk ki a fő Mandelbrot halmazt.

Includeoljuk a frakszal.h headert, hogy hozzáférjünk a FrakSzal osztályhoz, azon belül is a FrakSzal funkcióhoz. A benne lévő változók értékét átírjuk az alábbi változók értékére. Kirajzoljuk a halmazt, majd átadjuk a FrakAblaknak további műveletekre.

```
#include "frakszal.h"

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];
}
```

```
FrakSzal::~FrakSzal()
{
    delete[] egySor;
}

void FrakSzal::run()
{

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

    int iteracio = 0;

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

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

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

            reZ = 0;
            imZ = 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;

            }

            egySor[k] = iteracio;
        }

        frakAblak->vissza(j, egySor, szelesseg, iteraciosHatar);
    }
}
```

frakablak.cpp

Ebben deklaráljuk azt az ablakot, amiben a programunk futni fog..

```
#include "frakablak.h"

FrakAblak::FrakAblak(double a, double b, double c, double d,
                     int szelesseg, int iteraciosHatar, QWidget *parent)
```

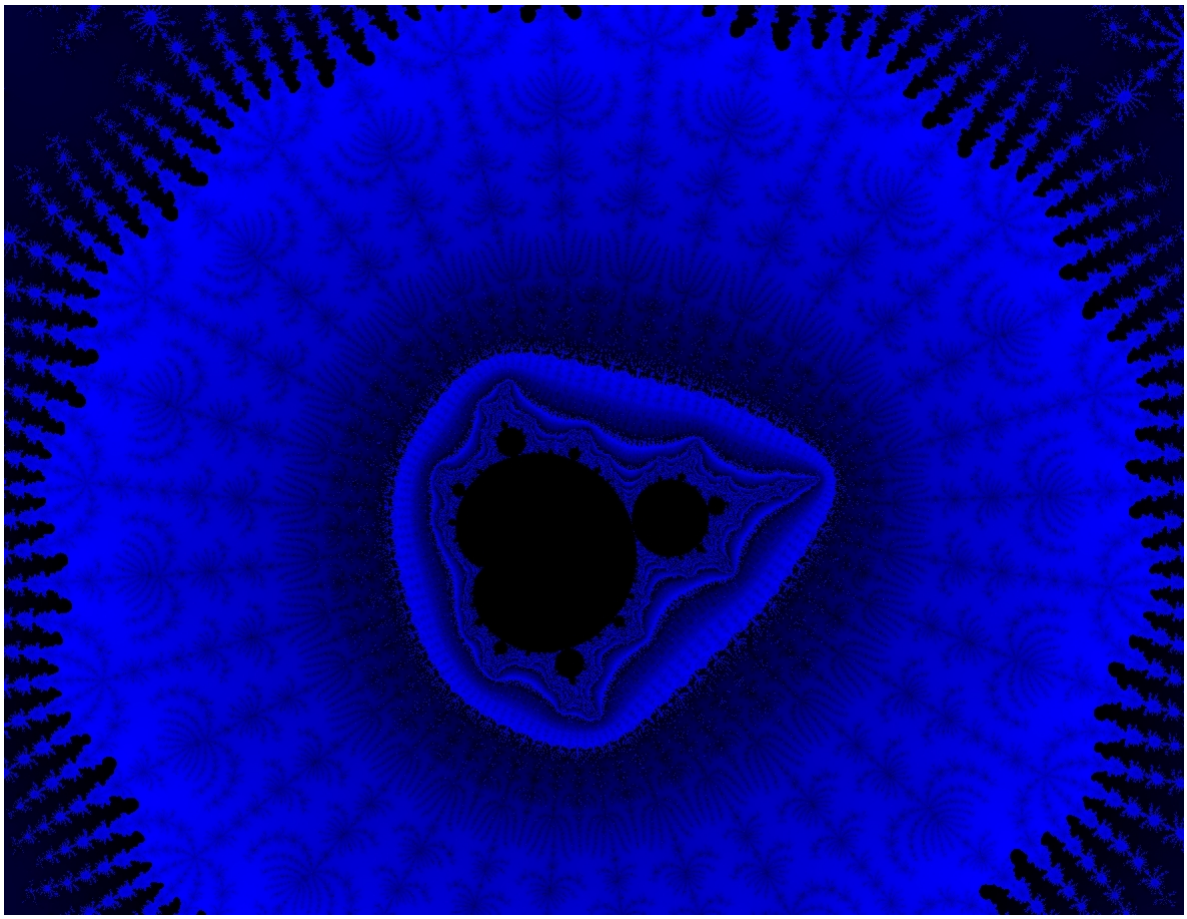
```
                        : QMainWindow(parent)
{
    setWindowTitle("Mandelbrot halmaz");

    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();

}

FrakAblak::~FrakAblak()
{
    delete fraktal;
    delete mandelbrot;
}

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

void FrakAblak::vissza(int magassag, int *sor, int meret, int hatar)
{
    for(int i=0; i<meret; ++i) {
        //        QRgb szin = qRgb(0, 255-sor[i], 0);
        QRgb szin;
        if(sor[i] == hatar)
            szin = qRgb(0,0,0);
        else
            szin = qRgb(
                    255-sor[i],
                    255-sor[i]%64,
                    255-sor[i]%16 );

        fraktal->setPixel(i, magassag, szin);
    }
    update();
}
```

main.cpp

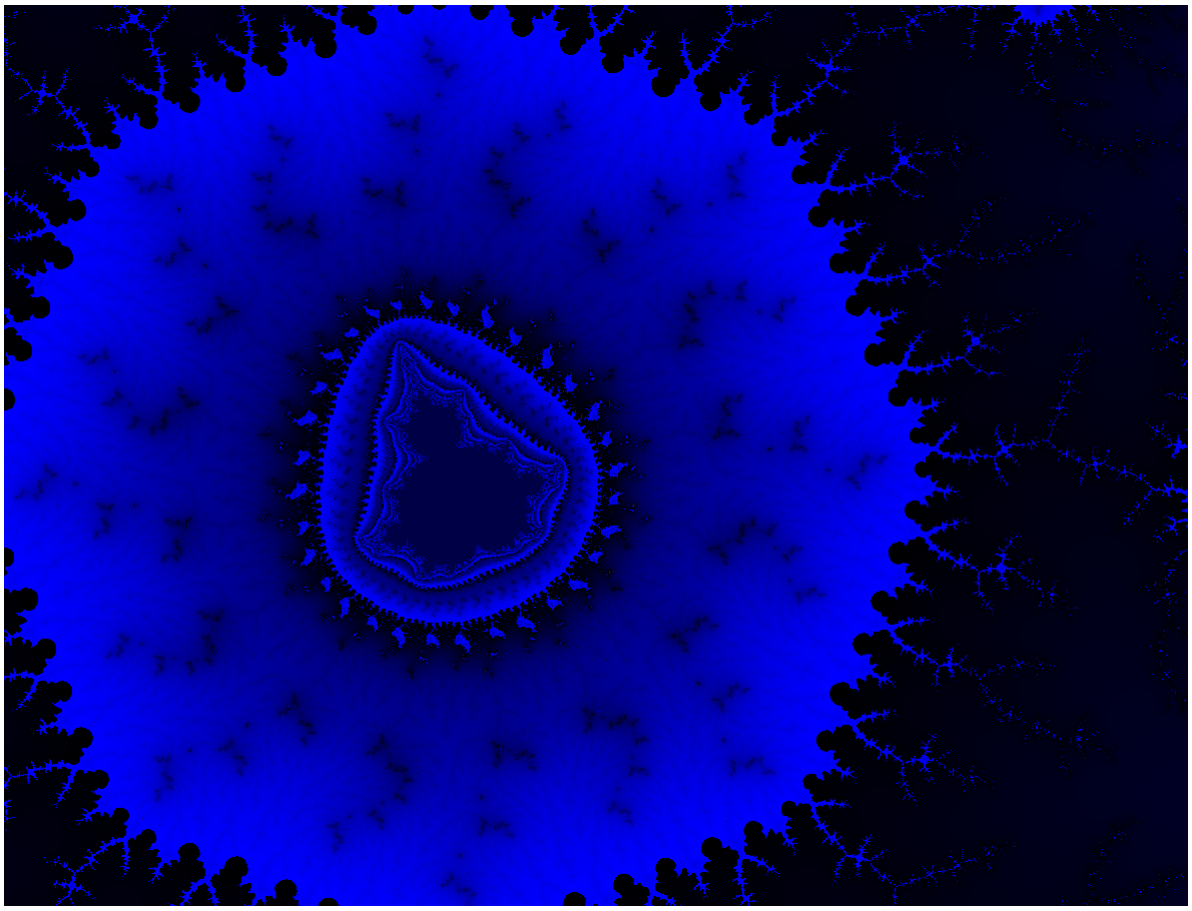
Itt fog megtörténni az egyesítés, itt hozzuk létre a nagyításokat is előre megadott paraméterekkel.

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

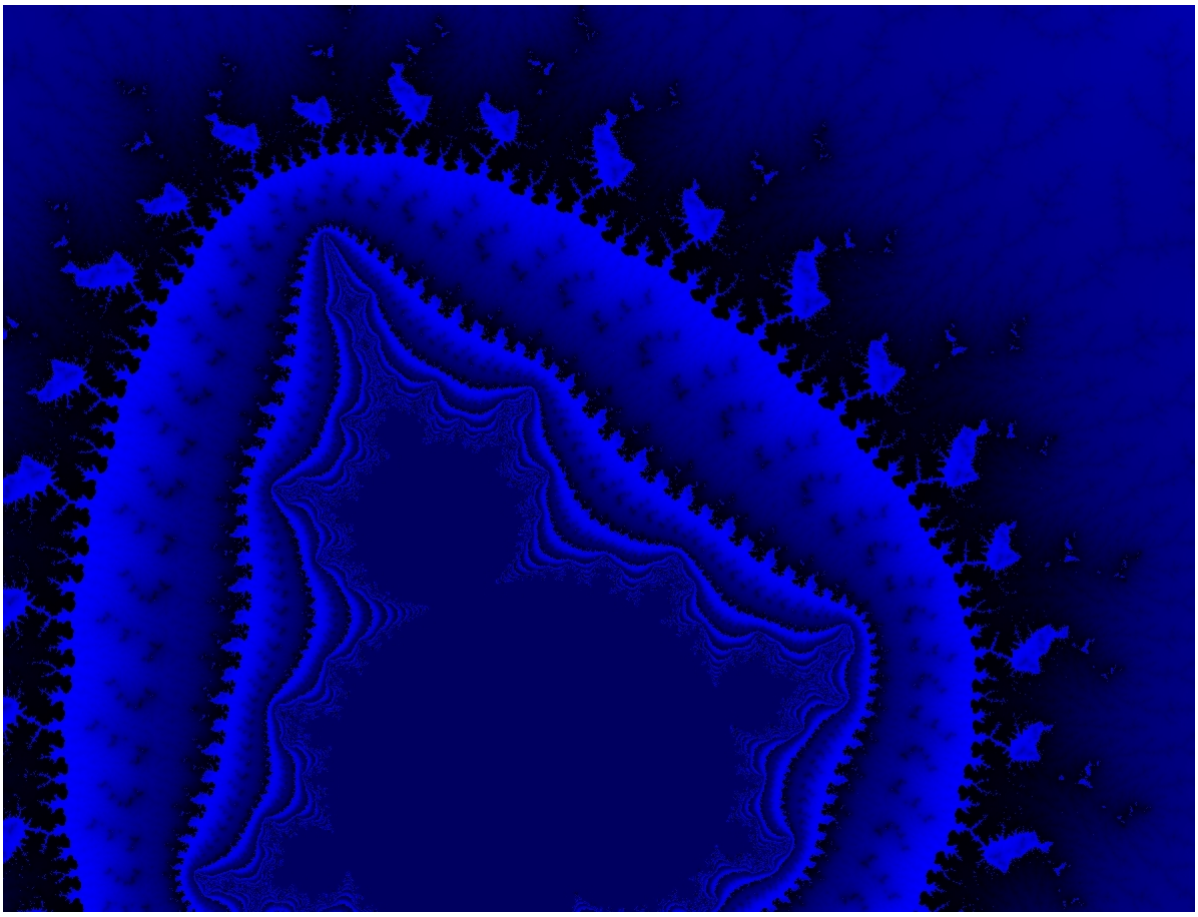
```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    // További adatokat olvashatsz le innen:
    // http://www.tankonyvtar.hu/informatika/javat-tanitok-2-3-080904
    FrakAblak w1,
    w2(-.08292191725019529, -.082921917244591272,
       -.9662079988595939, -.9662079988551173, 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();
}
```



5.2. ábra. A Mandelbrot-halmaz nagyító



5.3. ábra. A Mandelbrot-halmaz nagyító



5.4. ábra. A Mandelbrot-halmaz nagyító

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása: https://github.com/theefues/prog1/blob/master/mandelbrot/MandelbrotHalmazNagy%C3%AD
https://github.com/theefues/prog1/blob/master/mandelbrot/MandelbrotHalmaz.java

Tutorált: Iloshvai Áron

Fordítás: javac MandelbrotHalmazNagyító.java

Futtatás java MandelbortHalmazNagyító

A mandelbrot nagyító hasonlóképp működik, mint a C++os verziója annyi különbséggel, hogy ez interaktív. Oda zoomol be, ahova kattintunk.

Ehhez mindössze két fájlra lesz szükségünk: a MandelbrotHalmaz.java fájlra, ami mindössze annyit csinál, hogy kirajzolja a Mandelbrot halmazt, valamint a MandelbrotHalmazNagyító.java fájlra. A csoda az utóbbiban történik.

A MandelbrotHalmaz.java-n különösebb magyarázni való nincs, hiszen ugyanazt csinálja, mint a C++-os testvére: végigzongorázza az egész mátrixhálót azokért a pontokért, amik megfelelnek a feltételnek.

Kibővítjük a MandelbrotHalmaz osztályt a MandelbrotHalmazNagyító osztályával, így hozzáférhetünk a Mandelbrot halmazunkhoz, nem kell újra megírnunk hozzá a kódot.

Deklaráljuk a szükséges változókat: x,y a kijelölt terület bal felső sarkának a koordinátái, valamint az mx, my ami nagyítandó terület szélessége és magassága.

Ez után létrehozzuk a MandelbrotHalmazNagyító tagot, ami örökölni fogja a MandelbrotHalmaz osztály super() tagját, ami a rajzoló funkciónk. A setTitle segítségével elnevezzük az ablakunkat.

Hozzáadjuk, hogy egérrel ki tudjunk jelölni bizonyos részeket a képen. Ezt az addMouseListener függvénnyel tesszük meg, majd a korábban deklarált változókat felhasználva elmentjük az adatokat későbbi használatra, hogy tudjuk, mit kell kinagyítnai.

Mikor felengedjük az egeret, a kijelölt terület egy új ablakban újrarajzolódik, így kapunk egy zoomolt képet az eredeti halmazról.

Az addMouseMotionListenerrel figyeljük, merre jár az egér és hozzáadjuk a koordinátáit az mx és my változókhoz.

```
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {

    private int x, y;

    private int mx, my;
    public MandelbrotHalmazNagyító(double a, double b, double c, double d,
            int szélesség, int iterációsHatár) {

        super(a, b, c, d, szélesség, iterációsHatár);
        setTitle("A Mandelbrot halmaz nagyításai");

        addMouseListener(new java.awt.event.MouseAdapter() {

            public void mousePressed(java.awt.event.MouseEvent m) {

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

            public void mouseReleased(java.awt.event.MouseEvent m) {
                double dx = (MandelbrotHalmazNagyító.this.b
                        - MandelbrotHalmazNagyító.this.a)
                        /MandelbrotHalmazNagyító.this.szélesség;
                double dy = (MandelbrotHalmazNagyító.this.d
                        - MandelbrotHalmazNagyító.this.c)
                        /MandelbrotHalmazNagyító.this.magasság;

                new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+ ↩
                   x*dx,
                        MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
                        MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
                        MandelbrotHalmazNagyító.this.d-y*dy,
                        600,
```

```
                        MandelbrotHalmazNagyító.this.iterációsHatár);
            }
        });

        addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {

            public void mouseDragged(java.awt.event.MouseEvent m) {
                // A nagyítandó kijelölt terület szélessége és magassága:
                mx = m.getX() - x;
                my = m.getY() - y;
                repaint();
            }
        });
    }
```

Minden nagyításnál készítünk egy pillanatképet a halmazról. Ezt a pillanatfelvétel() funkcióval tesszük meg. Ha a számítás éppen fut, ezt pirossal jelöljük. Elkészítjük a képet, nevet adunk neki MandelbrotHalmazNagyitas_szám néven, majd a nevébe még belevesszük, hogy melyik tartományban található az adott halmaz.

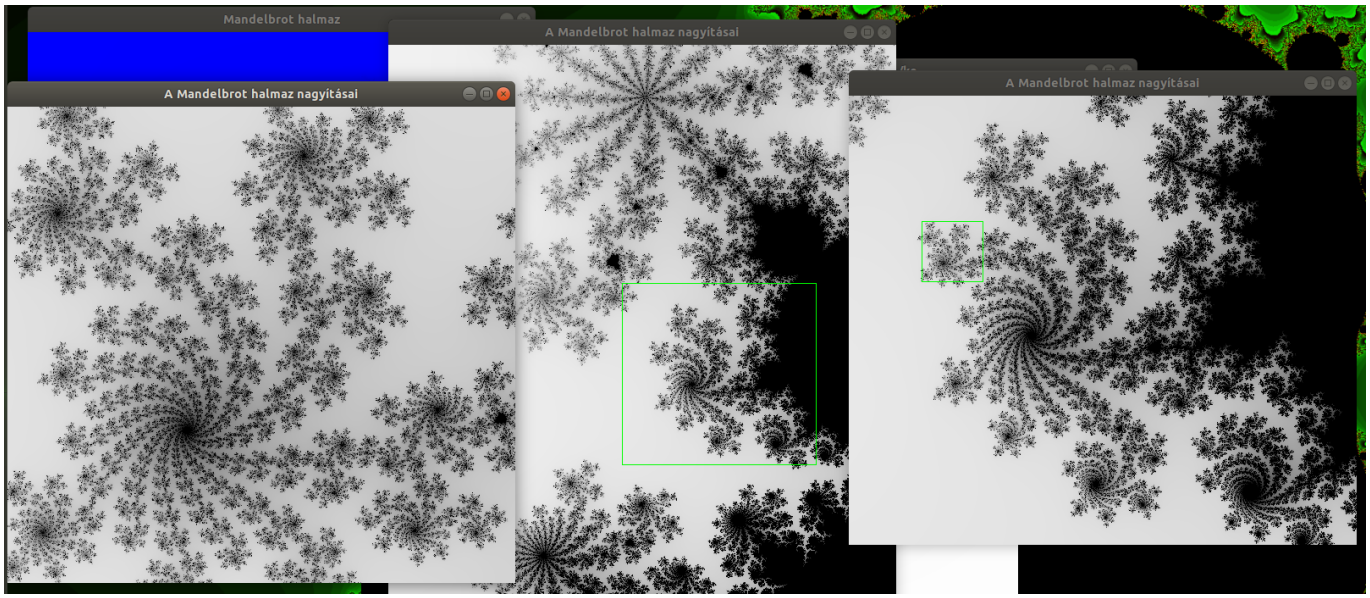
Elmentjük az elkészített képet .png formátumban.

A nagyítási terület kijelölését szemléltető paint() függvényünket is meg kell írnunk, hogy tudjuk, mit jelölünk ki éppen. Ha fut a számítás, akkor pirossal rajzolunk.

Ez után nincs más teendőnk, csak a mainben meghívni a MandelbrotHalmazNagyító tagot adott paraméterekkel, és a programunk tudni fogja, mit kell csinálni.

```
  public void pillanatfelvétel() {

        java.awt.image.BufferedImage mentKép =
                new java.awt.image.BufferedImage(szélesség, magasság,
                java.awt.image.BufferedImage.TYPE_INT_RGB);
        java.awt.Graphics g = mentKép.getGraphics();
        g.drawImage(kép, 0, 0, this);
        g.setColor(java.awt.Color.BLUE);
        g.drawString("a=" + a, 10, 15);
        g.drawString("b=" + b, 10, 30);
        g.drawString("c=" + c, 10, 45);
        g.drawString("d=" + d, 10, 60);
        g.drawString("n=" + iterációsHatár, 10, 75);
        if(számításFut) {
            g.setColor(java.awt.Color.RED);
            g.drawLine(0, sor, getWidth(), sor);
        }
        g.setColor(java.awt.Color.GREEN);
        g.drawRect(x, y, mx, my);
        g.dispose();

        StringBuffer sb = new StringBuffer();
        sb = sb.delete(0, sb.length());
        sb.append("MandelbrotHalmazNagyitas_");
```

```
        sb.append(++pillanatfelvételSzámláló);

        sb.append("_");
        sb.append(a);
        sb.append("_");
        sb.append(b);
        sb.append("_");
        sb.append(c);
        sb.append("_");
        sb.append(d);
        sb.append(".png");
        // png formátumú képet mentünk
        try {
            javax.imageio.ImageIO.write(mentKép, "png",
                    new java.io.File(sb.toString()));
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
    }

    public void paint(java.awt.Graphics g) {

         g.drawImage(kép, 0, 0, this);

        if(számításFut) {
            g.setColor(java.awt.Color.RED);
            g.drawLine(0, sor, getWidth(), sor);
        }

        g.setColor(java.awt.Color.GREEN);
        g.drawRect(x, y, mx, my);
    }

    public static void main(String[] args) {

        new MandelbrotHalmazNagyító(-2.0, .7, -1.35, 1.35, 600, 255);
    }
}
```



5.5. ábra. A Mandelbrot-halmaz nagyító

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/theefues/prog1/blob/master/welch/polargen.java

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

```
public double PolarGet()  {
    if (!bExists)
{
  double u1, u2, v1, v2, w;

  do{
    u1 = cRandomGenerator.nextInt (RAND_MAX) / (RAND_MAX + 1.0);
    u2 = cRandomGenerator.nextInt (RAND_MAX) / (RAND_MAX + 1.0);
    v1 = 2 * u1 - 1;
    v2 = 2 * u2 - 1;
    w = v1 * v1 + v2 * v2;
      }
  while (w > 1);

  double r = Math.sqrt ((-2 * Math.log (w)) / w);

  dValue = r * v2;
  bExists = !bExists;

  return r * v1;
}
```

```
  else
  {
    bExists = !bExists;
    return dValue;
  }
  };
```

Felvesszük a PolarGet() publikus függvényt. Ha van korábban tárolt érték, akkor azt adjuk vissza. Ha nincs, akkor készítónk egyet. Létrehotunk 2 random számot, megszorozzuk kettővel és elveszünk belőlük egyet. A kapott 2 értéknek vesszük a négyzetösszegét. Ezt addig csináljuk, amíg a kapott érték nagyobb, mint 1.

Ez után vesszük a -2szeres logaritmus hányadosának a négyzetgyökét, majd ennek a v1-es szorzatát visszaadjuk.

```
double
PolarGen::kovetkezo ()
{
  if (nincsTarolt)
    {
      double u1, u2, v1, v2, w;
      do
  {
    u1 = std::rand () / (RAND_MAX + 1.0);
    u2 = std::rand () / (RAND_MAX + 1.0);
    v1 = 2 * u1 - 1;
    v2 = 2 * u2 - 1;
    w = v1 * v1 + v2 * v2;
  }
      while (w > 1);

      double r = std::sqrt ((-2 * std::log (w)) / w);

      tarolt = r * v2;
      nincsTarolt = !nincsTarolt;

      return r * v1;
    }
  else
    {
      nincsTarolt = !nincsTarolt;
      return tarolt;
    }
}
```

C++-ban pedig ugyanez így néz ki.

A normálist visszaadó függvény javaban:

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

synchronized public double nextGaussian() {
```

```
if (haveNextNextGaussian) {
    haveNextGaussian = false;
    return nextNextGaussian;
} else {
    double v1, v2, s;
    do {
        v1 = 2 * nextDouble() -1; between -1 and 1
        v2 = 2 * nextDouble() -1; between -1 and 1
        s = v1 * v1 + v2 * v2;
    } while (s >= 1 || s == 0);

    double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
    nextNextGaussian = v2 * multiplier;
    haveNextNextGaussian = true;
    return v1 * multiplier;
}
```

Ha van tárolt érték, adja vissza azt. Ha nincs, csinálja meg. Generálunk 2 random számot -1 és 1 között, abból pedig elveszünk egyet. Majd vesszük a kettő négyzetösszegét. Ezt addig csináljuk, amíg s nagyobb egyenlő mint 1 vagy egyenlő 0-val.

Ez után felvesszük a multiplier változót, ami az s -2szeres logaritmus hányadosának a négyzetgyöke. A v2-őt megszorozzuk vele, ez lesz a tárolt érték. Majd kiadjuk, hogy van tárolt érték és visszaadjuk a v1 * multiplier eredményét.

6.2. LZW

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

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/welch/lzw.c

Fordítás: gcc lzw.c -o lzw

Futtatás: ./lzw infile -o outfile

Az LZW (Lempel-Ziv-Welch) egy veszteségmentes tömörítési algoritmus, melyet Terry Welch publikált az Abraham Lempel és Jacob Ziv által publikált LZ78 algortimus továbbfejlesztéseként. Innen jött a neve is.

A tömörítési eljárás alapja annyi, hogy a kódoló csak egy szótárbeli indexet küld ki. Welch 8 bites sorozatát kódolja 12 bites kóddá.

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

typedef struct binfa
{
  int ertek;
```

```
  struct binfa *bal_nulla;
  struct binfa *jobb_egy;
} BINFA, *BINFA_PTR;

BINFA_PTR
uj_elem ()
{
  BINFA_PTR p;

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

Includeoljuk a szükséges könyvtárakat majd strukturáljuk a binfát, ami egy érték változóból és a bal és jobb oldali faágakból fog állni.

Létrehozzuk az BINFA_PTR típus uj_elem függvényét, ami egy új elemet ad hozzá a fához, amennyiben a memóriában van hely. Ellenkező esetben hibát dob ki.

```
extern void kiir (BINFA_PTR elem);
extern void ratlag (BINFA_PTR elem);
extern void rszoras (BINFA_PTR elem);
extern void szabadit (BINFA_PTR elem);

int
main (int argc, char **argv)
{
  char b;
  int egy_e;
  int i;
  unsigned char c;
                //>BinfaPTR== user által definiált típus
  BINFA_PTR gyoker = uj_elem ();
  gyoker->ertek = '/';
  gyoker->bal_nulla = gyoker->jobb_egy = NULL;
  BINFA_PTR fa = gyoker;
  long max=0;
while (read (0, (void *) &b, sizeof(unsigned char)))
    {
        for(i=0;i<8; ++i)
        {
            egy_e= b& 0x80;
            if ((egy_e >>7)==0)
                c='1';
            else
                c='0';
        }
//      write (1, &b, 1);
```

```
    if (c == '0')
{
  if (fa->bal_nulla == NULL)
    {
      fa->bal_nulla = uj_elem ();
      fa->bal_nulla->ertek = 0;
      fa->bal_nulla->bal_nulla = fa->bal_nulla->jobb_egy = NULL;
      fa = gyoker;
    }
  else
    {
      fa = fa->bal_nulla;
    }
}
    else
{
  if (fa->jobb_egy == NULL)
    {
      fa->jobb_egy = uj_elem ();
      fa->jobb_egy->ertek = 1;
      fa->jobb_egy->bal_nulla = fa->jobb_egy->jobb_egy = NULL;
      fa = gyoker;
    }
  else
    {
      fa = fa->jobb_egy;
    }
}
  }

printf ("\n");
kiir (gyoker);
```

Externáljuk a kiir, ratlag, rszoras és szabadit függvényeket, amiket a main után deklaráltunk.

A mainben fel kell tüntetnünk az argc és argv változókat, mivel fájlinputtal fogunk dolgozni. Ez után deklaráljuk a szükséges változókat.

Ha a BINFA_PTR elérte a gyökeret, új elemet ad hozzá a fához. Ha az érték gyökér, hozzáadja a / jellel. Ha a gyökér elérte bal ágat, akkor áttér arra.

Amíg van mit beolvasni, minden 8 bitet arrébb shiftelünk, ezzel megtudjuk, hogy bal vagy jobb oldali ághoz kell hozzáadnunk, ezt jelzi a c változó. Ha 0 az értéke, akkor bal fához ad hozzá, ha 1, akkor a jobb oldalihoz.

Ha ezzel megvagyunk, a kiir függvényünkkel kiíratjuk a gyökeret.

```
extern int max_melyseg, atlagosszeg, melyseg, atlagdb;
 extern double szorasosszeg, atlag;

 printf ("melyseg=%d\n", max_melyseg - 1);
```

```
  /* Átlagos ághossz kiszámítása */
  atlagosszeg = 0;
  melyseg = 0;
  atlagdb = 0;
  ratlag (gyoker);
  // atlag = atlagosszeg / atlagdb;
  // (int) / (int) "elromlik", ezért casoljuk
  // K&R tudatlansági védelem miatt a sok () :)
  atlag = ((double) atlagosszeg) / atlagdb;

  /* Ághosszak szórásának kiszámítása */
  atlagosszeg = 0;
  melyseg = 0;
  atlagdb = 0;
  szorasosszeg = 0.0;

  rszoras (gyoker);

  double szoras = 0.0;

  if (atlagdb - 1 > 0)
    szoras = sqrt (szorasosszeg / (atlagdb - 1));
  else
    szoras = sqrt (szorasosszeg);

  printf ("altag=%f\nszoras=%f\n", atlag, szoras);

  szabadit (gyoker);
}
```

Externálva behozzuk a szükséges változókat. majd kiírjuk, mennyi a mélysége a fájlnak. Kiszámítjuk az átlagos ághosszt az ratlag függvénnyel, valamint a szórást az rszoras függvénnyel.

Ha az átlag darabszám - 1 nagyobb mint nulla, akkor a szórás a szórásösszeg és átlag darab - 1 hányadosának a négyzetgyöke lesz. Ha nem, akkor csak simán a szórásösszeg négyzetgyöke.

Ez után kiíratjuk az átlagot és a szórást, majd a szabadit függvénnyel felszabadítjuk a gyökeret.

```
int atlagosszeg = 0, melyseg = 0, atlagdb = 0;

void
ratlag (BINFA_PTR fa)
{

  if (fa != NULL)
    {
      ++melyseg;
      ratlag (fa->jobb_egy);
      ratlag (fa->bal_nulla);
      --melyseg;

      if (fa->jobb_egy == NULL && fa->bal_nulla == NULL)
```

```
  {

    ++atlagdb;
    atlagosszeg += melyseg;

  }

    }

}


double szorasosszeg = 0.0, atlag = 0.0;

void
rszoras (BINFA_PTR fa)
{

  if (fa != NULL)
    {
      ++melyseg;
      rszoras (fa->jobb_egy);
      rszoras (fa->bal_nulla);
      --melyseg;

      if (fa->jobb_egy == NULL && fa->bal_nulla == NULL)
  {

    ++atlagdb;
    szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));

  }

    }

}

//static int melyseg = 0;
int max_melyseg = 0;

void
kiir (BINFA_PTR elem)
{
  if (elem != NULL)
    {
      ++melyseg;
      _melyseif (melyseg > maxg);
  max_melyseg = melyseg;
      kiir (elem->jobb_egy);
      // ez a postorder bejáráshoz képest
```

```
      // 1-el nagyobb mélység, ezért -1
      for (int i = 0; i < melyseg; ++i)
  printf ("---");
      printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ↩
         ,
        melyseg - 1);
      kiir (elem->bal_nulla);
      --melyseg;
    }
}

void
szabadit (BINFA_PTR elem)
{
  if (elem != NULL)
    {
      szabadit (elem->jobb_egy);
      szabadit (elem->bal_nulla);
      free (elem);
    }
}
```

Itt pedig deklarálva van az összes függvény. Mindegyiket inicializálnunk kell, hogy ne legyen belőle félreérthető eredmény.

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: https://github.com/theefues/prog1/blob/master/welch/lzwbinfa.cpp

Az alapból inorder bejárású fát át kell írnunk pre- és postorderre.

A feladat nagyon egyszerű; meg kell keresnünk az alábbi kódrészletet a forrásfájlban és átírni így:

```
//inorder
void kiir (Csomopont * elem, std::ostream & os)
    {
        if (elem != NULL)
        {
          ++melyseg;
         for (int i = 0; i < melyseg; ++i)
                os << "---";
       kiir ( elem->nullasGyermek (), os);
            os << elem->getBetu () << "(" << melyseg - 1 << ")" << std:: ↩
               endl;
      kiir ( elem->egyesGyermek (), os);
      --melyseg;
    }
```

```
    }

output:
------0(1)
---------0(2)
------------0(3)
---------------1(4)
------------1(3)
---------------1(4)
------------------0(5)
---/(0)
---------1(2)
------------0(3)
------------1(3)
------1(1)
---------0(2)
------------1(3)
---------1(2)
depth = 5
mean = 3.33333
var = 1.0328

//preorder
void kiir (Csomopont * elem, std::ostream & os)
    {
        if (elem != NULL)
        {
          ++melyseg;
         for (int i = 0; i < melyseg; ++i)
                os << "---";
            os << elem->getBetu () << "(" << melyseg - 1 << ")" << std:: ↩
               endl;
      kiir ( elem->nullasGyermek (), os);
      kiir ( elem->egyesGyermek (), os);
      --melyseg;
    }
    }

output:
---/(0)
------0(1)
---------0(2)
------------0(3)
---------------1(4)
------------1(3)
---------------1(4)
------------------0(5)
---------1(2)
------------0(3)
------------1(3)
```

```
------1(1)
---------0(2)
------------1(3)
---------1(2)
depth = 5
mean = 3.33333
var = 1.0328

//postorder
void kiir (Csomopont * elem, std::ostream & os)
    {
        if (elem != NULL)
        {
          ++melyseg;
         for (int i = 0; i < melyseg; ++i)
                os << "---";
            kiir ( elem->nullasGyermek (), os);
      kiir ( elem->egyesGyermek (), os);
            os << elem->getBetu () << "(" << melyseg - 1 << ")" << std:: ↩
               endl;
      --melyseg;
    }
    }


output:
------0(1)
---------0(2)
------------0(3)
---------------1(4)
------------1(3)
---------------1(4)
------------------0(5)
---------1(2)
------------0(3)
------------1(3)
------1(1)
---------0(2)
------------1(3)
---------1(2)
---/(0)
depth = 5
mean = 3.33333
var = 1.0328
```

Inorder bejárásnál a gyökér középen van, a fa bal és jobb ága pedig felette és alatta helyzekedik el.

Preorder bejárásnál a gyökér van felül, a bal és jobb oldal pedig alatta.

Postorder bejárásnál pedig a gyökér van legalul, felette pedig a bal és a jobb oldali ágak.

Nyilván mindegyik variációhoz újra kell fordítanunk a programunkat, így (logikusan) egyszerre csak egy bejárást alkalmazhatunk.

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/theefues/prog1/blob/master/welch/lzwbinfa.cpp

A gyökér csomópontot bele kell ültetnünk az LZWBinFa osztályba:

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

Tagfüggvényként túlterheljük a >> operátort, így nyomhatjuk a fába az inputot a binFa >> b segítségével, ahol a b vagy 0 vagy 1.

Mivel tagfüggvény, így annak a fának vannak értelmezve a függbényei, amibe az adatokat akarjuk rakni.

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

Megnézzük, hogy egyest vagy nullát kell betenni. Ha nullát, ellenőrizzük, hogy van-e nullás gyermeke a csomópontnak. Ha van, arra épülünk tovább, ha nincs, elkészítjük. Új csomópontot hozunk létre, és bejegyezzük, hogy immár van 0-ás gyermek. Hozzáadjuk, majd visszatérünk a gyökérre.

Ha 1-es, akkor ugyanezt megcsináljuk vele, csak 0 helyett 1 lesz.

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/theefues/prog1/blob/master/welch/lzwmutato.cpp

A z3a7.cpp programunkat fogjuk módosítani a következőképp: az LZWBinFa osztály public részén belül az LZWBinFa ():fa (&gyoker) konstruktor tagot átírjuk arra, hogy:

```
LZWBinFa ()
{
    gyoker = new Csomopont();
    fa = gyoker;
}
```

Így deklaráljuk a gyökeret a fában.

Ha ezzel megvagyunk, át kell írnunk a ~LZWBinFa() destruktor tagot is a következők szerint:

```
~LZWBinFa ()
{
    szabadit (gyoker->egyesGyermek ());
    szabadit (gyoker->nullasGyermek ());
    szabadit (gyoker);
}
```

Ezzel deklaráljuk, hogy mit kell felszabadítanunk, mikor a program lefut.

Ez után már csak annyi a dolgunk, hogy lecseréljük a &gyoker-eket sima gyoker-ekre, hiszen nekünk az értékre van szükség, nem a mutatóra.

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/theefues/prog1/blob/master/welch/lzwmozgato.cpp

Tutor: Győri Márk Patrik

```
    LZWBinFa & operator= (const LZWBinFa & cp) {
        if(&cp != this)
            rekurzioIndutasa(cp.gyoker);
        return *this;
    };
```

Létrehozunk egy operátort, ami ha nem a gyökeret tartalmazza, akkor lemásolja saját magát. A másolás egy rekurzióval végződik, ami a fa minden ágát újra létrehozza egy másik gyökérre.

```
    void rekurzioIndutasa(Csomopont csm){
        if(csm.nullasGyermek()){
            fa = &gyoker;
            Csomopont *uj = new Csomopont ('0');
            fa->ujNullasGyermek (uj);
            fa = fa->nullasGyermek();
            std::cout << "GYOKER: nullas van"  << std::endl;
            rekurzioAzAgakon(csm.nullasGyermek());
        }
        if(csm.egyesGyermek()){
            fa = &gyoker;
            Csomopont *uj = new Csomopont ('1');
            fa->ujEgyesGyermek (uj);
            fa = fa->egyesGyermek();
            std::cout << "GYOKER: egyes van"  << std::endl;
            rekurzioAzAgakon(csm.egyesGyermek());
        }
    }

    void rekurzioAzAgakon(Csomopont * csm){
        if (csm->nullasGyermek()) {
            std::cout << "====van nullas" << std::endl;
            Csomopont *uj = new Csomopont ('0');
            fa->ujNullasGyermek(uj);
        }
        if (csm->egyesGyermek()){
```

```
            std::cout << "====van egyes" << std::endl;
            Csomopont *uj = new Csomopont ('1');
            fa->ujEgyesGyermek(uj);
        }
        Csomopont * nullas = fa->nullasGyermek();
        Csomopont * egyes = fa->egyesGyermek();
        if(nullas){
            fa = nullas;
            rekurzioAzAgakon(csm->nullasGyermek());
        }
        if(egyes){
            fa = egyes;
            rekurzioAzAgakon(csm->egyesGyermek());
        }
    }
```

A rekurzioIndutasa függvény indítja el a rekurziót, ha van nullás gyermeke akkor azon fut tovább, ha van egyes gyermeke akkor arra is meghívásra kerül. A fő eljárást maga a rekurzioAzAgakon függvény végzi, ez fut át az összes ágon, és létrehozza az új csomópontokat.

```
LZWBinFa binFa2;
    binFa2 = binFa;
```

A másolás már csak az egyenlőség jel operátorral meghívva történik, így az alap binFa átmásolódik a binFa2-be.

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/theefues/prog1/tree/master/conway

Tanulságok, tapasztalatok, magyarázat:

A program futtatásához szükségünk van aa libqt4-dev csomagra, amit Linux alatt az apt install libqt4-dev paranccsal tölthetünk le.

Ezt követően kiadjuk a qmake -project parancsot abban a mappában, amiben a fájlainkat elhelyeztük, majd csak simán a qmake parancsot, ami legenerál nekünk egy Makefile-t. Ezt a make paranccsal lefuttatjuk. Ha ezzel megvagyunk, a ./myrmecology paranccsal tudjuk futtatni a programunkat.

A main függvény bekéri a szükséges argumentumokat az osztálye előkészítéséhez. Az antwin.cpp és antwin.h fájlokban található kódok jelenítik meg a hangyákat, míg az antthread.cpp és antthread.h számolja ki, hova tegye a a hangyákat, merre menjenek.

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/theefues/prog1/edit/master/conway/SiklokilovoGameofLife.java

Tanulságok, tapasztalatok, magyarázat:

A program futtatásához szükség lesz a JDK egyik verziójára. Így kapni fogunk egy ablakot, amiben fekete pixelek mászkálnak keresztül kasul a képernyőn. Bal alul 2 sejt megy balra és jobbra, így generál 'ágyúgolyókat', amik átlósan mennek az ablakban. Ezt egészen addig folytatják, amíg ezek a golyók el nem érik őket. Ha ez megtörténik, meghalnak. A 2x2-es fekete négyzetek nem váltanak alakot, nem mozognak. Fixen ott maradnak, ahol vannak. Létrehozhatunk sejteket is, ha elkezdünk rajzolni a négyzetrácsos

síkon. Ezekre az alakzatokra is bizonyos szabályok fognak érvényesülni és ennek függvényében dől el, hogy meddig maradnak életben és ha "végük", milyen alakzatot hagynak hátra, ha hagynak egyáltalán.

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

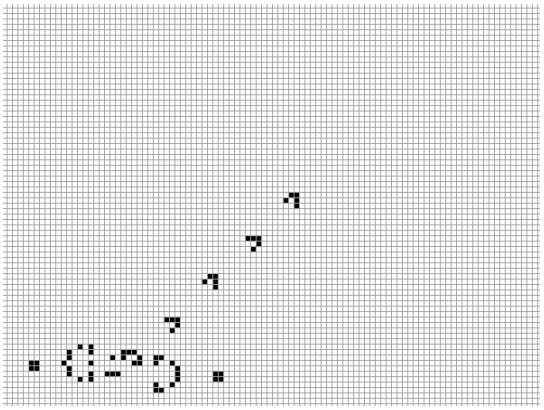
Megoldás forrása: https://sourceforge.net/p/udprog/code/ci/master/tree/source/kozepes/Qt/sejtautomata/

Tanulságok, tapasztalatok, magyarázat:

A program futtatásához szükségünk van aa libqt4-dev csomagra, amit Linux alatt az apt install libqt4-dev paranccsal tölthetünk le.

Ezt követően kiadjuk a qmake -project parancsot abban a mappában, amiben a fájlainkat elhelyeztük, majd csak simán a qmake parancsot, ami legenerál nekünk egy Makefile-t. Ezt a make paranccsal lefuttatjuk. Ha ezzel megvagyunk, a ./sejt paranccsal tudjuk futtatni a programunkat.

Így kapni fogunk egy ablakot, amiben fekete pixelek mászkálnak keresztül kasul a képernyőn. Bal alul 2 sejt megy balra és jobbra, így generál 'ágyúgolyókat', amik átlósan mennek az ablakban. Ezt egészen addig folytatják, amíg ezek a golyók el nem érik őket. Ha ez megtörténik, meghalnak. A 2x2-es fekete négyzetek nem váltanak alakot, nem mozognak. Fixen ott maradnak, ahol vannak.



7.1. ábra. Életjáték

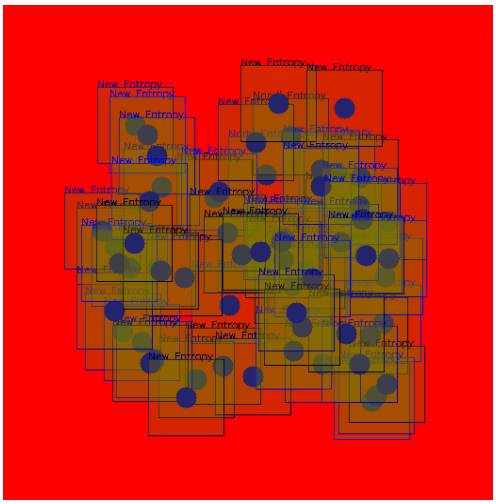
7.4. BrainB Benchmark

Megoldás videó:

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

Tanulságok, tapasztalatok, magyarázat:

A program futtatásához szükség van az OpenCV-re. Mikor elindítjuk a programot, egy vörös képernyő fogad minket, rajta négyzetekkel, amiknek egy nagy kék pötty van a közepén. A feladat, hogy a Samu_Entropy nevű entity keretein belül tartsuk a kurzorunkat.



7.2. ábra. A BrainB működés közben - Kép: Bátfai Norbert

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

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

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

Tanulságok, tapasztalatok, magyarázat...

Passz
Ezt a feladatot passzoltam.

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Passz
Ezt a feladatot passzoltam.

8.3. Minecraft-MALMÖ

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Passz
Ezt a feladatot passzoltam.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:

Tutor: Győri Márk

A megoldás iteratívan egy ciklussal van kezdve. Ennek a magja számolja ki az n-edik faktort. A prod változóba számolja ki a faktoriális szám összegét. Rekurzívan létre van hozva egy funkción (fact_rec) belüli funkció, ami rekurzívan van meghívva 'n - 1'-re addig amíg az n el nem éri az 1-et.

```
(defun fact (n)
    (do
        ((i 1 (+ 1 i))
        (prod 1 (* i prod)))
        ((equal i (+ n 1)) prod)))
(defun fact_rec (n)
    (defun fact-iter (n result)
        (if (= n 1)
            result
    (fact-iter (1- n) (* result n))))
(fact-iter n 1))
;(write "Add meg a szamot: ")
;(setq a (read))
;(print (fact a))
;(print (fact_rec a))

(print (fact 5))
(print (fact_rec 5))
```

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

Tanulságok, tapasztalatok, magyarázat:

Krómos színátmenetet készítünk egy bemeneti szövegre. Beimportáljuk a szükséges scriptet a GIMP-be.

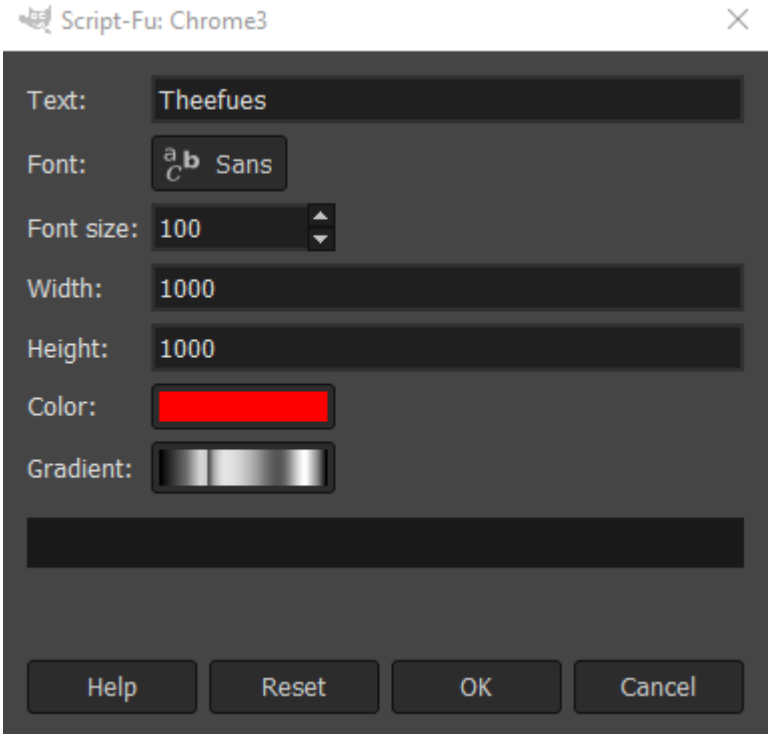
A kódot .scm kiterjesztésben bemásoljuk a GIMP scriptkönyvtárába, majd ráfrissítünk.

```
 (define (color-curve)
    (let* (
        (tomb (cons-array 8 'byte))
    )
        (aset tomb 0 0)
        (aset tomb 1 0)
        (aset tomb 2 50)
        (aset tomb 3 190)
        (aset tomb 4 110)
        (aset tomb 5 20)
        (aset tomb 6 200)
        (aset tomb 7 190)
    tomb)
)

;(color-curve)

(define (elem x lista)

    (if (= x 1) (car lista) (elem (- x 1) ( cdr lista ) ) )

)

(define (text-wh text font fontsize)
(let*
    (
        (text-width 1)
        (text-height 1)
    )

    (set! text-width (car (gimp-text-get-extents-fontname text fontsize  ↩
       PIXELS font)))
    (set! text-height (elem 2  (gimp-text-get-extents-fontname text  ↩
       fontsize PIXELS font)))

    (list text-width text-height)
    )
)

;(text-width "alma" "Sans" 100)
```

```
(define (script-fu-bhax-chrome text font fontsize width height color  ↩
   gradient)
(let*
    (
        (image (car (gimp-image-new width height 0)))
        (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" 100  ↩
           LAYER-MODE-NORMAL-LEGACY)))
        (textfs)
        (text-width (car (text-wh text font fontsize)))
        (text-height (elem 2 (text-wh text font fontsize)))
        (layer2)
    )

    ;step 1
    (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))

    (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)))

    (set! layer (car(gimp-image-merge-down image textfs CLIP-TO-BOTTOM- ↩
       LAYER)))

    ;step 2
    (plug-in-gauss-iir RUN-INTERACTIVE image layer 15 TRUE TRUE)

    ;step 3
    (gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE)

    ;step 4
    (plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE)

    ;step 5
    (gimp-image-select-color image CHANNEL-OP-REPLACE layer '(0 0 0))
    (gimp-selection-invert image)

    ;step 6
    (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)

    ;step 7
  (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)))

    ;step 8
    (plug-in-bump-map RUN-NONINTERACTIVE image layer2 layer 120 25 7 5 5 0  ↩
       0 TRUE FALSE 2)

    ;step 9
    (gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))

    (gimp-display-new image)
    (gimp-image-clean-all image)
    )
)

;(script-fu-bhax-chrome "Bátf41 Haxor" "Sans" 120 1000 1000 '(255 0 0) " ↩
   Crown molding")

(script-fu-register "script-fu-bhax-chrome"
    "Chrome3"
    "Creates a chrome effect on a given text."
    "Norbert Bátfai"
    "Copyright 2019, Norbert Bátfai"
    "January 19, 2019"
    ""
    SF-STRING       "Text"      "Bátf41 Haxor"
    SF-FONT         "Font"      "Sans"
    SF-ADJUSTMENT   "Font size" '(100 1 1000 1 10 0 1)
    SF-VALUE        "Width"     "1000"
    SF-VALUE        "Height"    "1000"
    SF-COLOR        "Color"     '(255 0 0)
    SF-GRADIENT     "Gradient"  "Crown molding"
)
(script-fu-menu-register "script-fu-bhax-chrome"
    "<Image>/File/Create/BHAX"
)
```

Ha sikerült az importálás, a Create fülből elérve ezt az ablakot kapjuk:



9.1. ábra. Chrome script

A legenerált képünk pedig így néz majd ki:

9.2. ábra. Chrome script

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Tanulságok, tapasztalatok, magyarázat:

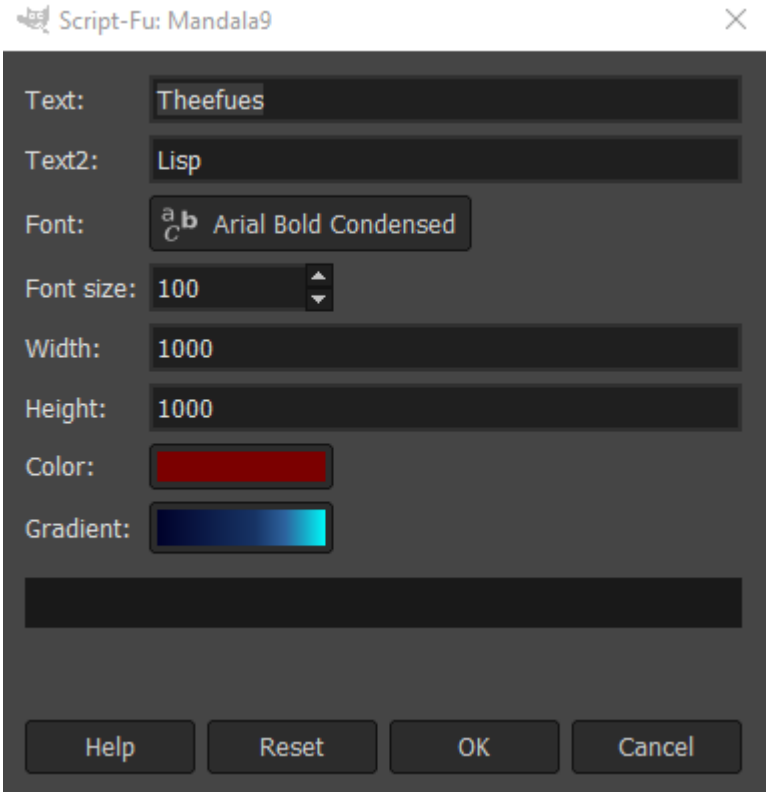
Ebben a feladatban az a dolgunk, hogy egy mandalát készítsünk GIMP segítségével. A GIMP képes kezelni különféle nyelvben megírt scripteket, ezzel megadott formákat képes legenárlni kép formátumban. Mi a Scheme nyelven megírt mandala név készítő scriptet fogjuk kipróbálni.

```
(define (script-fu-bhax-mandala text text2 font fontsize width height color ↩
    gradient)
(let*
    (
        (image (car (gimp-image-new width height 0)))
        (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" 100  ↩
           LAYER-MODE-NORMAL-LEGACY)))
        (textfs)
        (text-layer)
        (text-width (text-width text font fontsize))
        ;;;
        (text2-width (car (text-wh text2 font fontsize)))
        (text2-height (elem 2 (text-wh text2 font fontsize)))
        ;;;
        (textfs-width)
        (textfs-height)
        (gradient-layer)
    )

    (gimp-image-insert-layer image layer 0 0)

    (gimp-context-set-foreground '(0 255 0))
    (gimp-drawable-fill layer FILL-FOREGROUND)
    (gimp-image-undo-disable image)

    (gimp-context-set-foreground color)

    (set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ↩
       ))
    (gimp-image-insert-layer image textfs 0 -1)
    (gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2))  (/  ↩
       height 2))
    (gimp-layer-resize-to-image-size textfs)

    (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)))

    (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)
    (set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ↩
       -LAYER)))

    (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* 4) TRUE 0 0)
    (set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ↩
       -LAYER)))

    (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* 6) TRUE 0 0)
    (set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ↩
       -LAYER)))

    (plug-in-autocrop-layer RUN-NONINTERACTIVE image textfs)
    (set! textfs-width (+ (car(gimp-drawable-width textfs)) 100))
    (set! textfs-height (+ (car(gimp-drawable-height textfs)) 100))

  (gimp-layer-resize-to-image-size textfs)

    (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)

    (gimp-context-set-brush-size 22)
    (gimp-edit-stroke textfs)

    (set! textfs-width (- textfs-width 70))
    (set! textfs-height (- textfs-height 70))

    (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)

    (gimp-context-set-brush-size 8)
    (gimp-edit-stroke textfs)

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

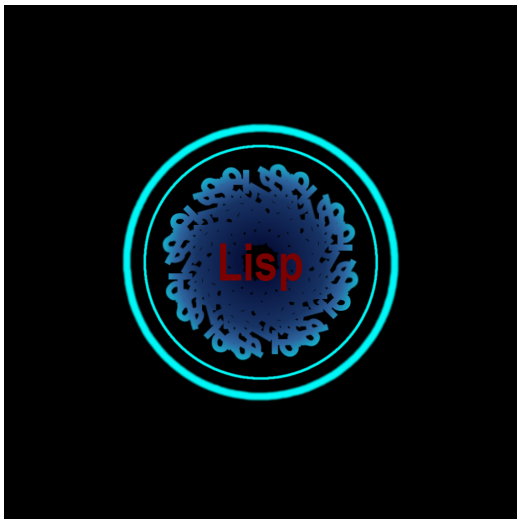
```
    (gimp-image-insert-layer image gradient-layer 0 -1)
  (gimp-image-select-item image CHANNEL-OP-REPLACE textfs)
  (gimp-context-set-gradient gradient)
  (gimp-edit-blend gradient-layer BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY  ↩
     GRADIENT-RADIAL 100 0
  REPEAT-TRIANGULAR FALSE TRUE 5 .1 TRUE (/ width 2) (/ height 2) (+ (+ (/  ↩
     width 2) (/ textfs-width 2)) 8) (/ height 2))

  (plug-in-sel2path RUN-NONINTERACTIVE image textfs)

    (set! textfs (car (gimp-text-layer-new image text2 font fontsize PIXELS ↩
       )))
    (gimp-image-insert-layer image textfs 0 -1)
    (gimp-message (number->string text2-height))
    (gimp-layer-set-offsets textfs (- (/ width 2) (/ text2-width 2)) (- (/  ↩
       height 2) (/ text2-height 2)))

    ;(gimp-selection-none image)
    ;(gimp-image-flatten image)

    (gimp-display-new image)
    (gimp-image-clean-all image)
    )
)
```

Hozzáadjuk a scriptet a GIMP scriptkönyvtárához, majd ráfrissítünk. Ezt az Edit fül alatt található Preferences tabon belül tehetjük meg. Megkeressük a Folder csoportot, ami tartalmazza a scriptkönyvtárakat. Valamelyikbe belerakjuk a .scm fájlunkat a fenti kóddal, majd a Filters - Script-Fu - Refresh Scripts segítségével befrissítjük. Ezt követően a File - Create - BHAX - Mandala9 füllel ez az ablak fogad minket:



9.3. ábra. Mandala script

Ha legeneráltuk a képünket, ilyen lesz a végeredmény:



9.4. ábra. Mandala script

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak - Pici könyv

Bevezetés

A világban mindenféle objektum van, amelyeknek vannak tulajdonságaik és bonyolult kapcsolatirendszer van közöttük. Reagálnak egymásra, viselkednek. Viselkedésük eltérő, ez alapján különböztetjük meg őket, így kategorizálhatók, osztályozhatók.

A valós világ túl összetett ehhez, így csak megközelítő értékekkel tudunk dolgozni. Ha valami hasonló, van közös, lényeges tulajdonságuk, egy csoportba soroljuk őket. Ezzel létrejön a valós világ modellje. Az ember is modelleket használ, mikor egy problémán gondolkodik, így próbálja megérteni azokat.

Egy modellel szemben három követelmény van: a leképzeés követelménye, a leszűkítés követelménye és az alkalmazhatóság követelménye. A számítógép lehetővé tette az emberi gondolkodás bizonyos elemeinek automatizálását. Egyes tulajdonságokat adatokkal, viselkedésmódot pedig programokkal tudjuk kezelni, ezzel egy újabb fajta modellt megadva.

Alapfogalmak

A számítógépek progamozására kialakult nyelveknek három szintjét különböztetjük meg: gépi, assembly szintű és magas szintű nyelv.

A magas szintű nyelven megírt programot forrásprogramnak vagy forrásszövegnek nevezzük. A formai szabályok összességét szintaktikai szabályoknak hívjuk. A tartalmi, jelentésbeli szabályokat pedig szemantikai szabályoknak.

Minden processzor saját gépi nyelvvel rendelkezik és csak az adott gépi nyelven írt programokat tudja végrehajtani. Ezért a magas szintű nyelven megírt szöveget át kell fordítanunk gépi nyelvre. Erre a fordítóprogramos és az interpreteres technikák léteznek.

A fordítóprogram egy olyan program, ami a magas szintű nyelvben írt kódot átfordítja gépi kódra, így úgynevezett tárgyprogramot állít elő. Működés közben a következő lépéseket hajta végre: lexikális elemzés, szintaktikai elemzés, szemantikai elemzés és kódgenerálás. A lexikális elemzés során feldaramolja a kódot lexikális egységekre. Ha nem talált semmilyen hibát, kész a programunk, ami immár gépi nyelvű, de nem futtatható. Ehhez a kapcsolatszerkesztőre lesz szükségünk. A futtatható programot a betöltő elhelyezia tárban és átadja neki a vezérlést.

A fordítóprogramok általában tetszőleges nyelvről tetszőleges nyelvre fordítanak. Léteznek olyan nyelvek is, amiben olyan forrásprogramot lehet írni, amelyek tartalmaznak nem nyelvi elemeket is. Ilyen például a C.

Minden programnyelvnek megvan a saját szabványa, amit hivatkozási nyelvnek hívunk. Ebben vannak definiálva a szintaktikai és szemantikai szabályok. Napjainkban a programok írásához grafikus IDE-t használunk.

A programnyelvek osztályozása

3 főbb csoportba osztályozzuk a nyelveket: imperatív, deklaratív és máselvű (egyéb) nyelvek. Az imperatív nyelvek tulajdonságai közé tartoznak például az algoritmikus nyelvek. A program utasítások sorozatából áll, legfőbb porgramozói eszköz a változó, szorosan kötődnek a neumann architektúrához. Két alcsoportját különböztetjük meg: eljárás- és objektumorientált nyelvek.

A deklaratív nyelvek csoportjába tartozik a nem algoritmikus nyelvek. Nem kotődnek szorosan a Neumann architektúrához. A programozó itt csak a problémát adja meg, nem a programkódot. Nincs lehetőség memóriaműveletekre. Két alcsoportja: funckionális és logikai nyelvek.

Egyéb nyelvek csoportjába sorolunk minden olyan nyelvet, ami a fentebbi kettőbe nem csoportosítható bele.

Kifejezések

A kifejezéses szintaktiaki aeszközök. Két komponensük van: érték és típus.

Egy kifejezés formálisan a következő összetevőkből áll: operandusok, operátorok és kerek zárójelek. A legegyszerűbb kifejezés egyetlen operandusból áll. Operandusszámtól függően lehet egyoperandusú(unáris), kétoperandusú(bináris) vagy háromoperandusú(ternáris) operátork

10.2. Programozás bevezetés - KERNIGHANRITCHIE könyv

[KERNIGHANRITCHIE]

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

Alapismeretek

C nyelvben a kiírást a printf függvénnyel érhetjük el. A program futtatása mindig az adott operációs rendszertő függ. Unix rendszerekben .c végződésű fájlt kell létrehoznunk és azt lefordítanunk. Ha sikerült, visszakapjuk a megadott stringet. A C beli függvények a FORTRAN függvényeihez hasonlítanak. A függvény neve bármi lehet, de a programunk végrehajtása mindig a mainnel kezdődik. A függvények közötti adatátadást argumentumok segítségével érhetjük el. A main általában argumentum nélküli, ezt így jelöljük: ()

A stringeket mindig rakjuk két " közé, hogy ne legyen semmi fennakadás. Új sort csak a \n karakterrel kezdhetünk.

Megjegyzést a // vagy /* karakterekkel adhatunk meg. Egy sor komment //, több sor esetén /* */ közé kell írni. Az int az egész, a float a lebegő pontos változókat jelölik. Mindkét fajta változó pontossága az adott számítógéptől függ. Léteznek még más adattípusok is: char, short, long és double. Ezek méretei is a számítógéptől függenek. Ezekből épiülnek fel a tömbök, sturktúrák és az unionok. Mutatók mutathatnak rájuk, függvények térhetnek vissza velük.

A while utasítás egy adott alkalommal fut végig. A printf %d értéke decimális számot, az %o oktálist, az %x hexadecimálist, a %c karaktert, az %s karakterláncot ad vissza.

A for utasítás azt tudja mint a while, csak általánosítva. 3 értéke van, az i lépésváltozó, a kritérium, és a lépésváltozó léptetése. Hozhatunk létre állandókat, ezt a #define paranccsal érhetjük el. Ezek konstansok, nem változnak a program során, mindig fix az értékük, így nem jelennek meg a deklarációban sem.

Vannak függvények, amivel inputot tudunk olvasni. Ez például a getchar() függvény. Ez minden híváskor beolvassa a következő bemeneti karaktert és a visszatérési értéke is ez a karakter lesz. Ennek az ellentéte a putchar(). Ez egy adott karaktert ír ki az outputra. Ezzel a módszerrel lehet megszámolni például a karaktereket.

10.3. Programozás - BME C++ könyv

[BMECPP]

Kivételkezelés

A program futtatása során a hibás esetek kezelése a C nyelvben jellemzően visszaadott hibakód alapján történik. A C++ nyelvben a kivételek alkalmazásával ennél sokkal strukturáltabb, átláthatóbb és könnyebben karbantartható a hibakezelés.

Hagyományos hibakezelés: main() -> Wrap()->Save()->ValidateAndPrepare(), DoSave()

Ha a hívási lánc mélyén hibát fedezünk fel, és a hibát nem tudjuk helyben elképzelni, akkor a hívó függvény számára visszatérési értékben megadott, a hibára jellemző hibakóddal jelezzük.

A kivételkezelés olyan mechanizmus, amely biztosítja, hogy ha hibát detektálunk valahol, akkor a futás azonnal a hibakezelő ágon folytatódjon. A kivételkezelést azonban nem véletlen hívják kivétel és nem hibakezelésenk.

Ha a védett blokkban nem találunk hibát, akkor a {} közötti védett blokk valamennyi utasítáa lefut. Ezt követően a vezérlés a catch ág után folytatódik. Mivel a catch a hibákat veszi észre, így normál esetben a benne lévő kódok nem futnak le.

Ha a kódban hibás feltétlet detektálunk, akkor a throw kulcsszóval kiévtelt dobunk, amit a catch elkap. A main függvényben található kódot egy try-catch blokkban helyezzük el. A throw által dobott adat a kivételobjektum.

A try-catch blokkok egymásba is ágyazhatóak. Egy kivétel dobásakor annak elkapásáig a függvények híbási láncában felfelé haladva az egyes függvények lokális változói felszabadulnak. Ez a folyamat a hívási verem visszacserélése.

A kivételkezelés mechanizmusa az, hogy kivétel esetén azonnal a kivételkezelőre ugorjunk. Például ha írunk egy programot, amiben arra kérjük a felhasználót, hogy írjon be egy nullánál különböző számot, és létrehozunk egy kivételt, hogy abban az esetben, ha a bekapott input 0, akkor tudassa a felhasználóval, hogy a bemeneti érték nem lehet nulla.

III. rész

Második felvonás

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

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

Köszönet illeti a NEMESPOR, https://groups.google.com/forum/#!forum/nemespor, az 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.