Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

COLLABORATORS			
	TITLE : Univerzális programozás		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Dankó, Zsolt	2019. május 10.	

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
1.0.0	2019-05-09	Elkészült a könyv első iniciális változata.	Dpara

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

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 forrása: `ciklus0.c` ; `ciklus1.c` ; `ciklusall.c`

A feladatba kicsit nehézkesen kezdtem bele, mivel végtelen ciklusokkal programozás során nem nagyon találkoztam, de egy kis keresgélés után megértettem, feldolgoztam az anyagot és megoldottam a feladatokat. Az alább használt `while(1) {}` ciklusokat azért választottam, mert szerintem ez az egyik legelterjedtebb és legegyszerűbb módszer. A `while(1) {}` azt jelenti, hogy addig fut a `while` ciklusban lévő kód, amíg teljesül az (1) feltétel, amely helyett boolean típusából kiindulva írhattam volna `while(true)`-t is. Mivel itt nincs kód, ezért a ciklus a végtelenségig fut.

Elsőként az egymagos megoldást próbáltam ki,amely sikerült ezzel az egyszerű kóddal:

```
int main() {

  while(1) {}

return 0;
}
```

Következőnek azt próbáltam ki, hogyan lehetne megoldani, hogy ne foglalja le a CPU-t futás közben. Ehhez elég volt egy `sleep()`-et beraknom a végtelen ciklusba.

```
int main() {

 while (1) {
  sleep();
 }

return 0;
}
```

A végére hagytam azt a feladatot, hogy a CPU minden magját leterhelje a program, ez már kicsit bonyolultabb volt, mivel OpenMP-t kellett hozzá használnom, ezért `#include`-olnom kellett az `"omp.h"` könyvtárat.

```
#include <omp.h>

int main() {

#pragma omp parallel

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

Ezt már fordítanom is máshogy kellett, a megszokott `gcc in -o out` helyett `gcc -o out -fopenmp in` parancsot kellett használnom.

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.

Az eredményünk a következő: Ha a T1000 program megálló program, akkor mégsem áll meg, hanem végtelen ciklusba esik; és ha a T1000 végtelen ciklusba eső gép, akkor megáll a T1000 gép. Így egy szép kis paradoxont kapunk, amit Alan Turing fedezett fel. A felfedezés szerint a megállási probléma nem dönthető el, mivel az egyetlen megoldása ennek az ellentmondásnak az, hogy a T100 program nem létezik, tehát az alábbi algoritmusunk sem.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása:bhax/thematic_tutorials/bhax-textbook/Turing/csere.c

Először is, deklarálnunk kell a két változót, értéket kell adni nekik, aztán kell felcserélnünk őket. Ezután hogy kimutassuk, a változók tényleg felcserélődnek-e, ki kell íratnunk a változók értékeit a `printf()` függvénnyel. Ehhez #include-olnunk kell a `"stdio.h"` könyvtárat, ebben van deklarálva a `printf()` függvényünk.

```
#include <stdio.h>

int main() {

int a = 33;
int b = 42;

printf("a = %d\nb = %d\n",a,b);

b = b - a;
a = a + b;
b = a - b;

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

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:`labda.c`; `labdaif.c`

Az if-es megoldásban a labdának relatív "játszótere" van, ami attól függ, mekkora az ablak mérete; az if nélküli programban viszont a labdánk fix kereteken belül mozoghat, ezt indikálja a két szaggatott(kötőjeles) vonalunk. A kód sikeres fordításához meg kell hívnunk a `"curses.h"` könyvtárat, ami a libncurses package-ben van, így előbb le kellett töltenünk azt. A kód működésének szíve és lelke a `for (;;)` végtelen ciklus, itt "mozog" a labdánk. A lényege, hogy ha a labdánk eléri az egyik falat, akkor elkezdjen menni az ellenkező irányba, a gyorsaságot pedig az `usleep(100000)` függvény adja meg, ez 100ms-t jelent, vagyis a labda 0.1 másodpercenként mozog.

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: `szohossz.c`

A szóhossz megszámolásához szükségünk van a bitshift operátorra, ez található a `while` cikluson belül. Jelen ciklus azt csinálja, hogy az "a" változónkat addig shifteljük, vagyis addig léptetjük balra és adunk 0-t a jobb oldalára, amíg 0 nem lesz az értéke. Mindeközben minden shiftelésnél a "b" változó értékét is növeljük mindig 1-el, amíg le nem zárul a ciklus, így a "b" fogja nekünk kimutatni, hány bites a szó a gépünkön.

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: `pagerank.c`, BN FerSML projekt dia 55-59.

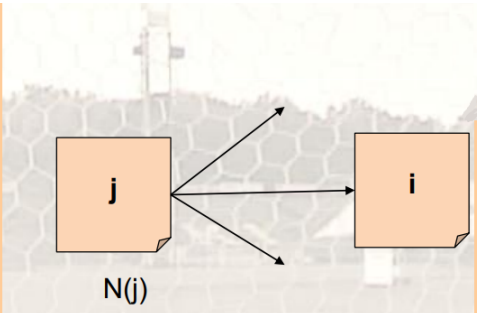
Egy weboldal fontossága teljesen szubjektív; attól függ, mit keres az olvasó. Viszont objektíven is meg tudjuk közelíteni a weboldalakat, mégpedig úgy, hogy a relatív fontosságot nézzük. Erre való a PageRank módszer, ami objektíven méri, az adott oldalt hányan keresik, vagyis más oldalak hányszor mutatnak rá.

A PageRankot linkmátrix megoldással fogjuk megvalósítani, ebben segítenek nekünk az alábbi ábrák:

$$L = (l_{i,j})$$

$$l_{i,j} = \begin{cases} \frac{1}{N(j)}, j \longrightarrow i \\ 0 \end{cases}$$

$$L = \begin{pmatrix} l_{1,1} & \dots & l_{1,j} & \dots & l_{1,n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ l_{i,1} & \dots & l_{i,j} & \dots & l_{i,n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ l_{n,1} & \dots & l_{n,j} & \dots & l_{n,n} \end{pmatrix}$$



$$\begin{pmatrix} PR(h_1) \\ \vdots \\ PR(h_n) \end{pmatrix}$$

$$\begin{pmatrix} PR(h_1) = \sum_{j=1}^{n} l_{1,j} PR(h_j) \\ \vdots \\ PR(h_i) = \sum_{j=1}^{n} l_{i,j} PR(h_j) \\ \vdots \\ PR(h_n) = \sum_{j=1}^{n} l_{n,j} PR(h_j) \end{pmatrix}$$

Itt ha **h** jelöli a PR vektort, akkor **h=Lh**

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

Az ikerprímek olyan prímpárok, melyek különbsége 2.

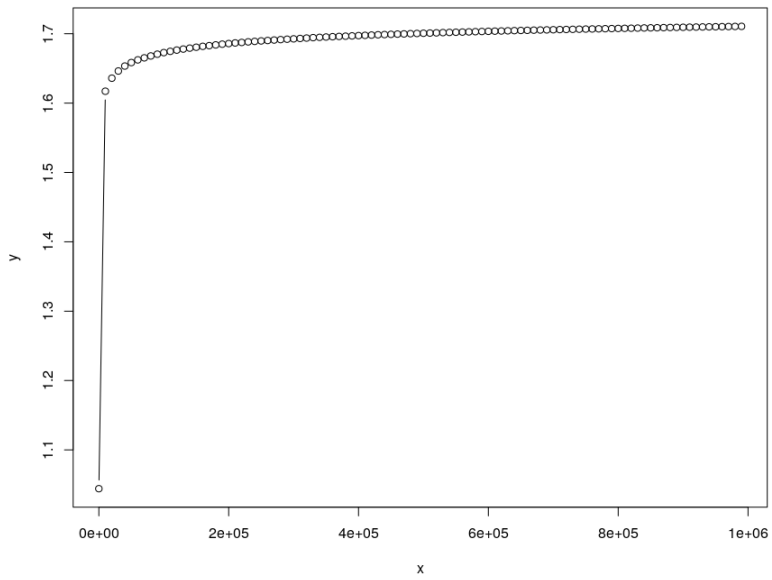
Brun-tétel: Az ikerpríme reciprokösszege egy (Brun-konstans) véges értékhez konvergál. A prímek reciprokösszege divergens, ebből láthatjuk, hogy végtelen sok prímszám van.

Nézzük is ezt az R szimulációban:

```
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))
}
```

Ha még nem tettük meg, töltsük le a matlab könyvtárat, R-en belül **install.packages("matlab")**. Meghívjuk a matlab könyvtárat, majd megírjuk az **stp** függvényt. A függvény első sorában meghívunk egy **primes(x)** függvényt, ami az x prímeit adja vissza. Megírjuk a különbségfüggvényt, majd megnézzük, hogy hol 2 a különbség. A **t1primes** az ikerpár egyik tagja, az idx prímjeit mutatja; a **t2primes** a pár második tagja, szintén az idx prímjeit adja vissza, és hozzáad 2-t. Elosztjuk 1-el a két változót, majd összeadjuk őket, végül pedig visszakapjuk az előbbi függvény összegét. Rajzoljuk is ki ezt a függvényt:

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



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

A Monty Hall-paradoxon egy valószínűségi paradoxon, ami az USA-beli "Kössünk üzletet" című vetélkedő utolsó feladatán alapul. A paradoxon nevét Monty Hallról, a vetélkedő műsorvezetőjéről kapta. A lényege, hogy a műsorvezető bemutat három ajtót, amelyekből 2 ajtó túloldalán egy értéktelen nyeremény van, egy ajtó pedig egy értékes nyereményt rejt. A játékosnak ki kell választania egy ajtót, majd a műsorvezető kinyit egyet, amely egy értéktelen nyereményt tartalmaz. Ezután a játékosnak lehetősége van dönteni, hogy marad az eredetileg kiválasztott ajtajánál, vagy vált a másikra. Alapból a játékosnak 1/3 esélye van, hogy az értékes nyereményt találja el, és 2/3 esélye, hogy az értéktelent. Mivel kinyitották az egyik ajtót, így a játékos igazából 50% hogy az értékes nyeremény ajtaját nyitja ki, szóval a paradoxon nagy kérdése az, hogy érdemes lenne-e változtatni, vagy hogy mindez számít-e egyáltalán.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Tutor: Veress Máté

Megoldás forrása: https://github.com/dankozsolti/prog1/blob/master/decimal.c

Ezen Turing gép lényege, hogy a decimális számokat átváltja unáris számrendszerbe, és ezt a következőképp oldotta meg Turing: Feltételezzük, hogy a gép mindig 0-ból indul, és a szalagunk a 0 fejtől balra üres. Egy szám végét szintén 0-val jelöljük, tehát ha input-ként beírjuk, hogy "6", az alábbit fogja kiírni: "01111110". A szalagot így fel lehet tölteni, például ha beírjuk, hogy "2, 3, 4", akkor a kimenet "0110111011110" lesz. Leprogramozni úgy tudjuk jelen gépünket, hogy bekérjük a decimális számot inputként, majd egy léptetéses ciklussal egyeseket csinálunk a számokból, és kiíratjuk azt.:

```
#include <stdio.h>

int main()
{
  int decimal;
  scanf ("%d", &decimal);
  printf("Unáris:\n0");

for(int i=0; i < decimal; i++)
  printf("1");
  printf("0\n");
  return 0;
}
```

3.2. Az $a^nb^nc^n$ nyelv nem környezetfüggetlen

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

Megoldás videó:

Megoldás forrása:

Noam Chomsky amerikai nyelvész nevéhez többek közt a generatív nyelvtanok csoportosítása fűződik. Négy osztályba sorolta őket (Chomsky hierarchia):

- 0. típus(általános)
- 1. típus (környezetfüggő)
- 2. típus (környezetfüggetlen)
- 3. típus (reguláris)

Nekünk az első, azaz környezetfüggő típussal kell foglalkoznunk ebben a feladatban. Akkor környezetfüggő a grammatika, ha teljesül az adott feltétel: $\mathbf{S \rightarrow \varepsilon}$ **, és az S soha nem jelenik meg a jobb oldalon.** Most pedig nézzük, hogy is néz ki ez gyakorlatban.

```
S, X, Y (változók)
a, b, c (konstansok)
S -> abc, S -> aXbc, Xb -> bX, Xc -> Ybcc, bY -> Yb, aY -> aaX, aY -> aa
```

Itt kiindulunk az S-ből, és a következőt kapjuk:

```
S (S -> aXbc)
aXbc (Xb -> bX)
abXc (Xc -> Ybcc)
abYbcc (bY -> Yb)
aYbbcc (aY - aa)
aabbcc
```

Ezt a példát tovább fejthetjük, és így megkapjuk az alábbi levezetést:

```
S (S -> aXbc)
aXbc (Xb -> bX)
abXc (Xc -> Ybcc)
abYbcc (bY -> Yb)
aYbbcc (aY -> aaX)
aaXbbcc (Xb -> bX)
aabXbcc (Xb -> bX)
aabbXcc (Xc -> Ybcc)
aabbYbccc (bY -> Yb)
aabYbbccc (bY -> Yb)
aaYbbbccc (aY -> aa)
aaabbbccc
```

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:

Tutoráltam: Veress Máté

A legegyszerűbb módon egy `for()` ciklusban lehet bemutatni a két szabvány közötti különbséget. C99-ben, azaz az ISO szabvány szerint a for cikluson belül tudjuk egyszerre deklarálni, majd léptetni a változónkat, az alábbi módon:

```
for(int i, i<5, ++i)
```

C89-ben - vagyis az ANSI szabvány alkalmazásával - ha ugyanezt a kódot próbáljuk fordítani, hibát ír ki. Hogy működjön a kódcsipetünk, a `for()` cikluson kívül kell deklarálnunk az `i`-t.

```
int i;
for(i<5, ++i)
```

Hogy a gépünk felé jelezzük, melyik szabványt akarjuk alkalmazni, fordításkor a végére kell írnunk, hogy `-std=c99` vagy `-std=c89`

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!

Tutor: Veress Máté

Megoldás forrása:

Egy olyan lexert írunk, amely felismeri a bemenetünkből a valós számokat. A feladat megoldásához a Flex-et használjuk (így óriások vállán állunk, nem kispályázunk), ezért a .l kódot át kell konvertálnunk C-be, majd csak ezután tudjuk fordítani és futtatni, az alábbi módon:

```
lex -o valos.c valos.l
gcc valos.c -o valos
./valos
```

Mivel a lexerünk megírja magának a C programot, így csak annyit kell megadnunk, hogy miket tegyen a kódba. a kódunk % jelekkel van elválasztva; az első részben a `"stdio.h"`könyvtárat hívjuk meg és a `realnumbers` változónkat deklaráljuk, a második részben megadjuk, hogy számokkal dolgozunk. A következő részben a fordítási szabályokat adjuk meg a lexernek: Akármennyi számunk lehet (az első digit), majd legyen valamennyi a pont után, de legalább egy (így tizedes tört), de az utóbbi opcionális. Növeljük a `realnumbers` változót, majd kiíratjuk a realnum-ot. Az utolsó (main) rész elejében a `yylex()` függvénnyel hívjuk a lexikális elemzőt, majd kiírjuk a `realnumbers` számát.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Tutoráltam: Szilágyi Csaba

Megoldás forrása: `1337.l`

Ehhez a feladathoz megoldásához szintén a Flex lexikális elemzőt használjuk, ezért az előző feladathoz hasonlóan fordítjuk a kódot:

```
lex -o 1337.c 1337.l
gcc 1337.c -o 1337
./1337
```

A kódunk fő eleme egy cipher, ennek a struktúráját építjük fel elsőnek. A https://simple.wikipedia.org/wiki/Leet oldalon található helyettesítő karaktereket használjuk ebben a programban. A struktúránkban deklarálunk egy karaktert(c) és azt, hogy az adott karaktert mire cserélhetjük. Hagyjuk a programnak, hogy kiválassza, melyik karakterre cserélje a négy megadott lehetőség közül az adott betűket vagy számokat. Hogy kezeljük a kis- és nagybetűket, a `tolower()` függvényt használjuk, ez átalakítja az összes nagybetűnket lowercase-re.

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:sig.c

Megoldás videó:

Először is, mivel jeleket kezelünk, meg kell hívnunk a "signal.h" könyvtárat. Deklarálunk egy jelkezelő változót, ami kiírja, mikor elkapjuk a jelet. Ezután egy while(1) végtelen ciklusba megírjuk a kód azon részét, amely megfogja a SIGINT signalt, ami a Ctrl+C billentyűkombinációnak felel meg. Így hiába küldjük neki ezt a signalt, a programunk megfogja, így nem tudjuk azt bezárni, viszont a Ctrl+Z kombinációval továbbra is le lehet állítani, ráadásul ezt el sem tudjuk kapni.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

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

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

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

- $(\forall x \exists y ((x<y)\wedge(y \text{ prím})))$ - Minden x-re van olyan y, amely nagyobb x-nél és prímszám.
- $(\forall x \exists y ((x<y)\wedge(y \text{ prím})\wedge(SSy \text{ prím})))$ - Minden x-re van olyan y, amely nagyobb x-nél, prímszám és SSy is prím.
- $(\exists y \forall x (x \text{ prím}) \supset (x<y)) $ - Létezik olyan y, amelynél minden x prímszám, és ezért x kisebb mint y.
- $(\exists y \forall x (y<x) \supset \neg (x \text{ prím}))$ - Létezik olyan y, amelynél minden x nagyobb mint y, ezért x nem prímszám.

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?

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

Megoldás videó:

Megoldás forrása:

SKIP

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

Megoldás videó:

Megoldás forrása: tm.c

A programhoz nem kell semmilyen előleges define vagy deklaráció a könyvtárhívásokon kívül, ezért egyből jöhet a `main` függvény. deklarálunk egy `int` típusú változót, aminek értékül 5-öt adunk, ez a sorok számát fogja megadni később. Megadjuk, hogy ha bármilyen hibát észlel a program a `malloc` funkciónál, akkor a mutató a NULL-ba mutasson és kiugorjon a programból. A hibák észlelése között `printf` függvényekkel kiíratjuk a tárcímeket az output elejére. Nézzük meg a mutatókat három ugyanilyen függvénnyel:

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

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

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

    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]);
```

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: titkosito.c

Mivel szöveget akarunk titkosítani, ezért azt bele kell raknunk egy fájlba, a programunk pedig majd ezen fog dolgozni. Mivel kell egy visszafejtési kulcs, a könyvtárhívások után `#define`-oljuk a kulcsunk méretét, majd a buffer méretét is beállítjuk(256 byte). A define után nincs más dolgunk, mint a main függvényt megírni. A kulcsot és a buffert egy-egy `char` változóban tároljuk, majd megadunk két `int` változót a kulcs indexének és az olvasott bájtoknak. A kulcs méretét is egy `int`-ben tároljuk, majd megadjuk, hogy a program futtatásakor be tudjuk írni a kívánt kulcsot argumentumként. Ezután jön egy `while` ciklus, ami a bufferből olvassa a bájtokat, majd a cikluson belül egy `for` ciklus, amely EXOR segítségével kicseréli a szöveg karaktereit a megadott kulcsunk alapján. Mivel a program a standard outputra löki az átírt karaktereket, ezért egy outputot kell megadnunk, amelybe írhatja a titkosított szöveget.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Tutor: Győri Márk Patrik

Megoldás forrása: titkositas.java

```
import java.io.InputStream;
import java.io.OutputStream;

class ExorTitkosito {
    public ExorTitkosito(String text, InputStream input, OutputStream ↩
       output) {
        byte[] key = text.getBytes();
        byte[] buffer = new byte[256];

        int index = 0;
        int readed = 0;
        try {
            while ((readed = input.read(buffer)) != -1) {
                for(int i = 0; i < readed; i++) {
                    buffer[i] = (byte)(buffer[i] ^ key[index]);
                    index = (index + 1) % key.length;
                }
                output.write(buffer, 0, readed);
            }
        } catch (Exception e) {
            System.out.println("Hibas a ciklus *cry* Hiba: " + e.getMessage ↩
               ());
        }
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        if(args.length > 0)
            new ExorTitkosito(args[0], System.in, System.out);
        else
            System.out.println("Nem adtal meg argumentet!");
    }
}
```

Egy külön osztályt hozunk létre a titkosításhoz **(class ExorTitkosito)**, aminek input-ként szolgál a kulcs és egy olvasó-író argumentum. A **while** ciklus addig fut, amíg az olvasott bájtok el nem fogynak. Ezeket a bájtokat eltoljuk a megadott számú kulcs bájttal. Minden bájton végigmegy a ciklus, majd visszaírjuk a kimeneti bufferbe. Ezt a művelet egy **try-catch** blokkba rakjuk, így elfogja a hibákat, ami java kódolásnál lényeges.

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: toro.c

A törő programunk lényege, hogy a titkosításban használt EXOR-ral fejtsük vissza a szöveget, bár ehhez még kelleni fog pár plusz infó a programnak, amiből ki tud indulni. A program folyamatosan vizsgálja az átlagos szóhosszt, valamint feltételezi, hogy a szöveg tartalmazza a gyakori magyar szavakat(hogy, nem, az, ha). Ezekkel a vizsgálatokkal csökkentjük a potenciális töréseket. A másik dolog, amit tudnunk kell az az, hogy a kulcs hány és milyen típusú karakterből áll. A `main` függvényben a titkosításhoz hasonló deklarációkat használjuk, tehát a kulcs méretét és az olvasott bájtokat; majd egy `while` ciklusban olvassuk a titkosított bájtokat, ezután pedig a titkos bufferben lenullázzuk a maradék helyet. Előállítjuk az összes kulcsot `for` ciklusokkal, majd a lehetséges tiszta szövegeket kiírja a kulccsal együtt a standard outputra. Végül újra EXOR-ozunk, így nem kell egy második buffert alkalmaznunk.

4.5. Neurális OR, AND és EXOR kapu

R

A képlet és a vázlat forrása: Tankönyvtár

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

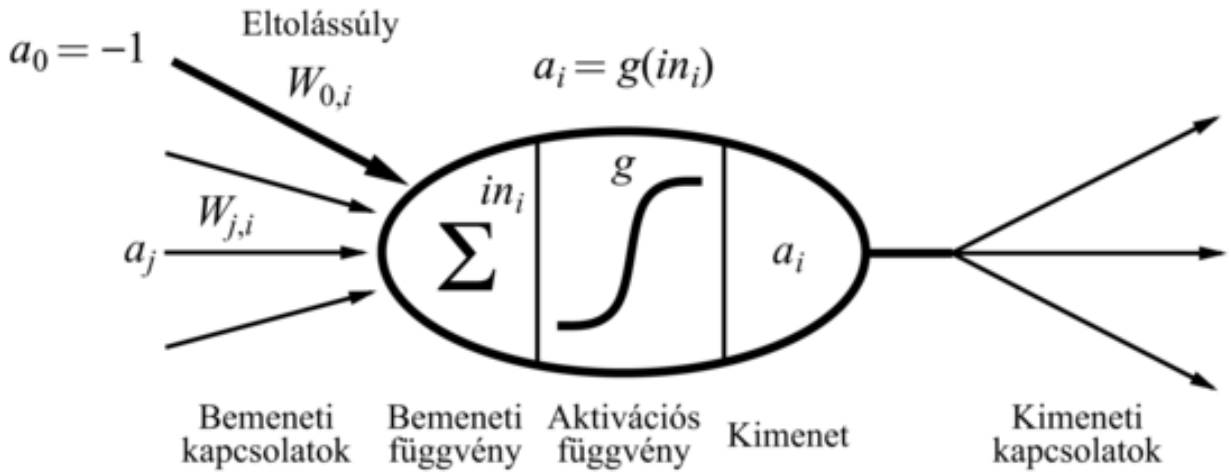
Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/NN_R

Először is beszéljünk magáról a neuronról. A neuron egy olyan agysejt, melynek alapfeladata elektromos jelek összegyűjtése, feldolgozása és szétterjesztése. Elnagyolva az mondható, hogy a neuron akkor „tüzel”, amikor a bemeneti értékek súlyozott összege meghalad egy küszöböt. Az alábbi ábra a neuron egyszerű

matematikai modelljét mutatja, ahogy McCulloch és Pitts megalkották. Az egység kimeneti aktivációja:

$$a_i = g\left(\sum_{j=0}^{n} W_{j,i} a_j\right)$$

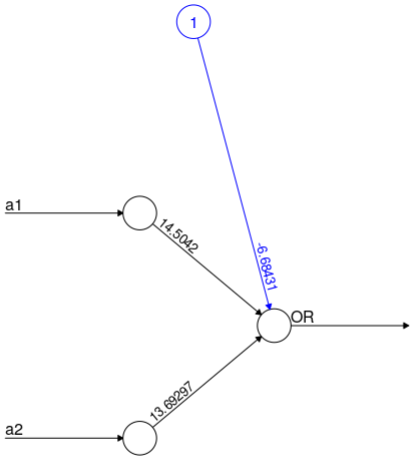
ahol a_j a j-edik egység kimeneti aktivációja és $W_{j,i}$ a j-től i-ig vezető összeköttetés súlya.



Mi főként a kapuval, vagyis az aktivációs függvénnyel foglalkozunk. Mivel ebben az R szimulációban neurális hálókkal fogunk foglalkozni ezért az első dolgunk az lesz, hogy meghívjuk a **library(neuralnet)** könyvtárat, majd a fenti ábra alapján megnézzük a kapukat. Kezdjük is az elsővel:

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

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

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

plot(nn.or)

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

Ha beírjuk a fenti kódot, a következőt fogja kidobni:

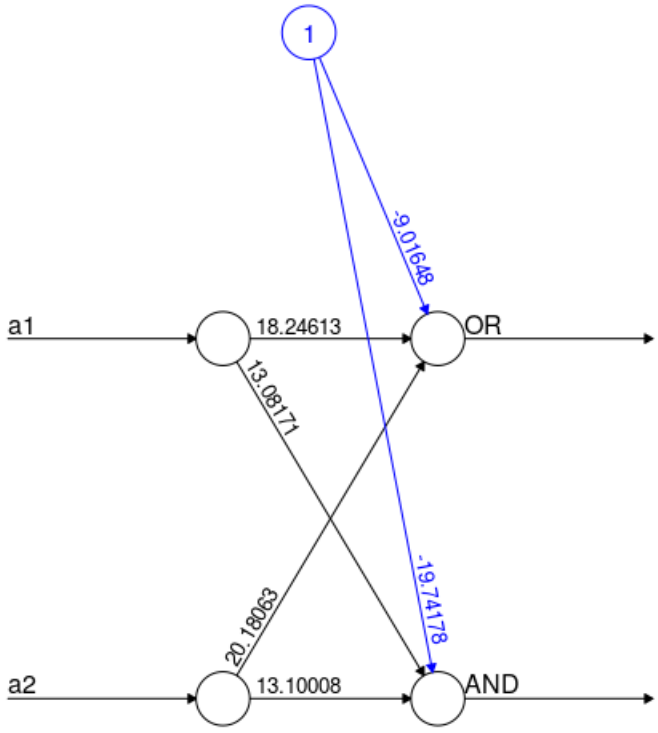


Error: 1e-06 Steps: 145

A fenti kód és ábra alapján a többi kaput is meg lehet oldani, nem kell szinte semmit átírnunk. Elkészítjük a második (ORAND) kaput.

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

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

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

plot(nn.orand)

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

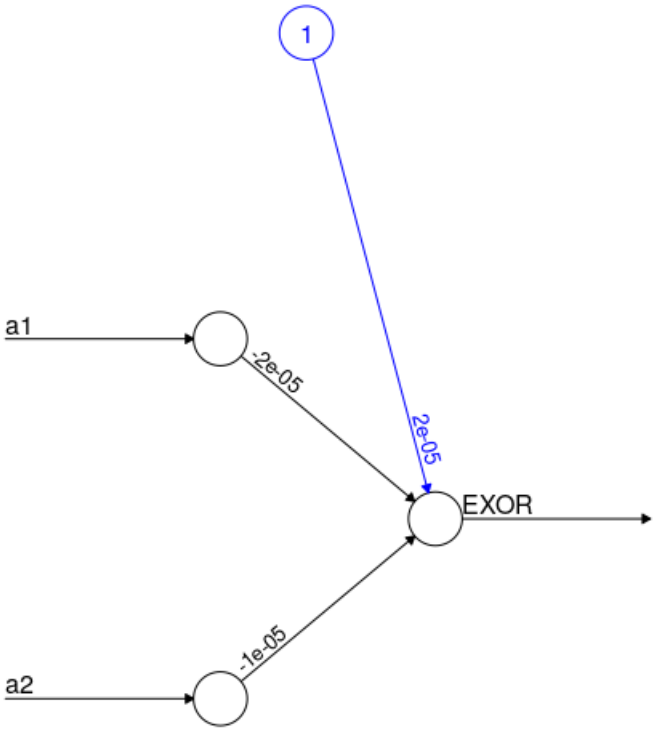
mivel az AND és az OR függetlenek egymástól, ezért ugyanazon hálózatban két kimenete lesz és így nyugodtan beírhatjuk mindkettőt.



Megnézzük az EXOR-os példát, itt sem kell átírnunk szinte semmit, csak az OR-t (vagy ORAND) EXOR-ra.

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

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

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

plot(nn.exor)

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

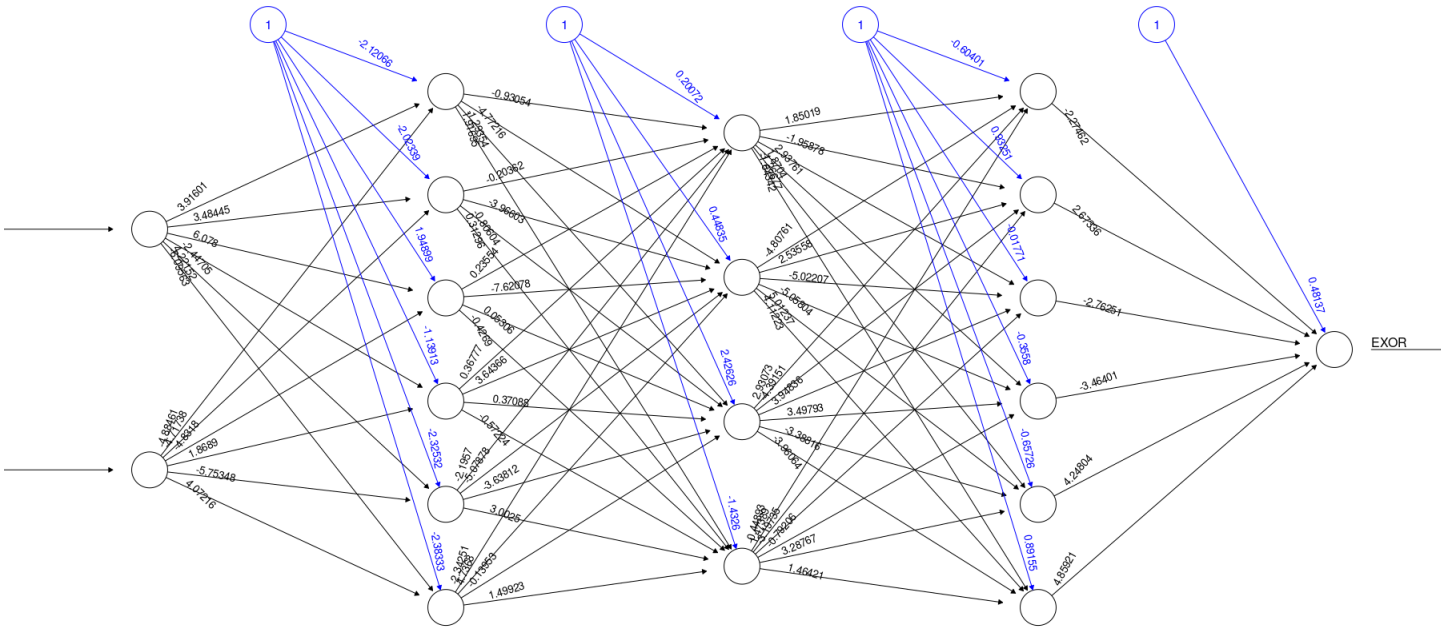
A kizáró vagy példa az alábbi hálót fogja legenerálni:



Végül játszunk egy kicsit a **hidden** részünkkel, mivel eddig csak a háló azon részeit láttuk, amiket lekérdeztünk, tehát nem láttunk rejtett neuronokat. Így viszont beleláthatunk kicsit jobban a rendszerbe, átadunk neki egy listát, hogyan is rajzolja ki a többrétegű hálónkat: **hidden=c(6, 4, 6)**. Nézzük is hogyan néz ki ez a kódon belül:

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

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

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

plot(nn.exor)

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

Így már az ábránk is kicsit érdekesebb lesz:



4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

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

SKIP

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás videó:

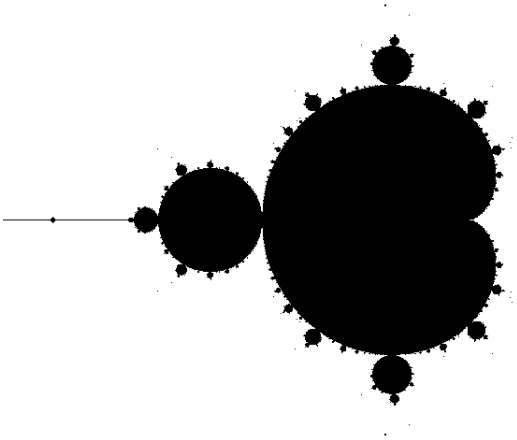
Megoldás forrása: mandelpngt.c++

Mielőtt kifejezetten a Mandelbrot-halmazról beszélnénk, beszéljünk előbb egy kicsit a fraktálokról, mint geometriai alakzatokról. A fraktálok végtelenül komplex geometriai alakzatok, melyek két gyakori, jellemző tulajdonsággal rendelkeznek: Az első, hogy a fraktálok határoló vonalai vagy felületei végtelenül "gyűröttek" vagy "érdesek". A második gyakori jellemzőjük a problematikus, de jól érzékelhető önhasonlóság. A Mandelbrot-halmaz azon c komplex számokból áll, melyekre az alábbi (komplex szám értékű) X_n rekurzív sorozat:

$x_1 := c$

$x_{n+1} := (X_n)^2 + c$

nem tart végtelenbe, azaz abszolút értékben korlátos. A halmazt Benoît Mandelbrot fedezte fel 1980-ban a komplex számsíkon. A Mandelbrot-halmazt úgy tudjuk megjeleníteni, hogy az origóra húzunk egy pl. 800x800-as rácsot, majd a rács minden pontján megnézzük, melyik komplex számnak felelnek meg. Ha jól csináltuk, az alábbi ábrát kell kapnunk:



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

Megoldás videó:

Megoldás forrása: 3.1.2.cpp

Az előző feladat alapján megnézzük, hogyan is néz ki ez az algoritmus C++ nyelven megírva. Include-oljuk az alap "iostream" library-t; a "png++/png.hpp" könyvtárat, mivel képpel foglalkozunk; majd a lényeget, a "complex" könyvtárat. Bekérjük az argumentumokat a felhasználótól (mint például az ablakunk szélességét és magasságát, vagy az iterációs határt), majd készítünk egy üres képet, amin elkezdjük rajzolni a halmazunkat. Ez végigmegy a sorokon és oszlopokon egy léptetéses **for()** ciklussal, és a háló rácspontjaival megegyező komplex számokat kirajzolja.

5.3. Biomorfok

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

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

Mivel még mindig a Mandelbrot-halmazzal foglalkozunk, ezért a programunk forráskódja szerkezetileg hasonlítani fog az előző feladatban látott kódra. Mindezt az iterációk módosításával hozzuk létre. A biomorfokról bővebben **ezen** a linken olvashatunk, és ebből is fogunk dolgozni. Ugyanazokat a könyvtárakat használjuk, mint az előző feladatban, ugyanazokat az argumentumokat használjuk, de ebben a verzióban argumentumként kérjük be a komplex számsíkért felelő adatokat, nem pedig utólag deklaráljuk őket. Ezek után a szerkezet még ugyanaz marad, csinálunk egy üres képet, majd oszloponként és soronként haladva kirajzoljuk a kért képünket; pont úgy, mint az előző feladatban. A képletet változtathatjuk; próbálkozhatunk, hogy kijön-e nekünk is ugyanaz az ábra, mint amit referenciának használtunk a pdf-ből.

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása: mandelpngc_60x60_100.cu

A CUDA megvalósítás C++ alapokon működik, ezért nem csoda, hogy hasonlít a C++ kódunkhoz. A változtatás benne az, hogy nem a CPU-t terheljük a kódunkkal, hanem az nVidia kártyánk CUDA magjait, amely gyorsabb végrehajtást nyújt, mint a processzorunk. Ezt viszont tesztelni is szeretnénk, ezért megírjuk a CUDA-kompatibilis kódot, amit majd nvcc-vel kell fordítanunk. A legfőbb változtatás az alábbi kódcsipet beírásával történik:

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

  int *device_kepadat;
  cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));

  dim3 grid (MERET / 10, MERET / 10);
```

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

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

}
```

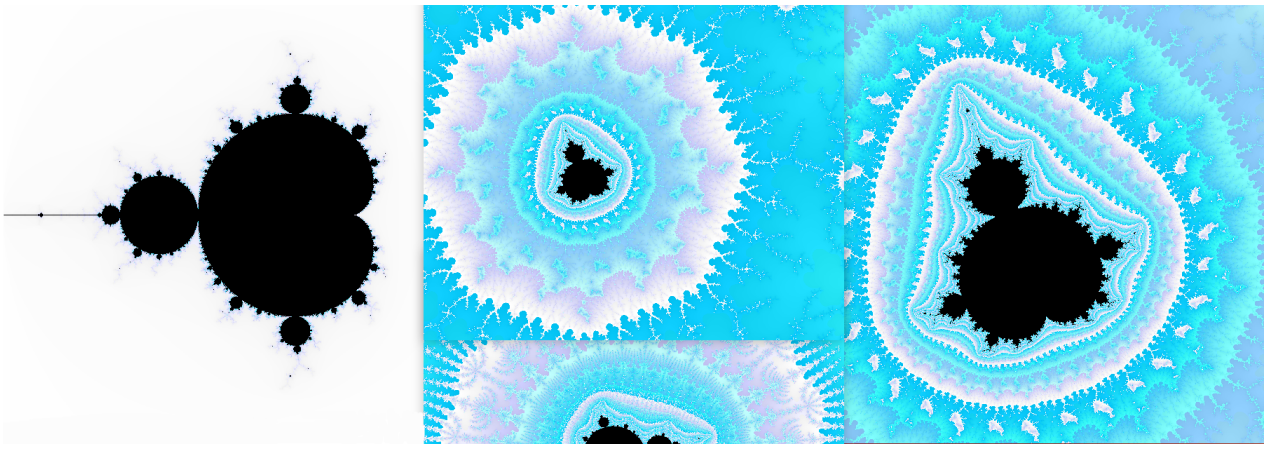
5.5. Mandelbrot nagyító és utazó C++ nyelven

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 forrása: Frak

Megoldás videó:

Hogy a forrásainkból képek legyenek, szükségünk lesz a libqt4-dev csomagra. Telepítés után a mappánkba lépve beírjuk a terminálba, hogy qmake, ez generálni fog nekünk egy Makefile-t, ezután egyszerűen egy make paranccsal le is tudjuk generálni a dolgokat. Futtatás után négy képet dob ki a program, amelyekből az első az alap Mandelbrot-halmaz, a másik három pedig random nagyítások a halmazon belül.



5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása: MandelbrotHalmaz.java MandelbrotHalmazNagyító.java

A kód hasonlóan működik a C++ nyelven írt Mandelbrot halmazhoz, de ezzel a programmal tudunk kicsit szórakozni vele, mert ezzel interaktívan tudunk belenézni a halmazba. Fordítani a **javac MandelbrotHalmazNagyító.java** sorral tudjuk, majd **java MandelbrotHalmazNagyító.java**-val futtatjuk. Ahhoz, hogy a nagyító működjön, előbb a halmazt kell létrehoznunk, amit a **MandelbrotHalmaz.java** programban írtunk meg. A program interaktív része akkor mutatkozik meg, amikor kijelölünk az alap halmazon egy területet, és a program újraszámítja a halmazt, és csak az előzőleg kijelölt területet látjuk. Ezzel eljátszhatunk a végtelenségig, mivel a fraktálok végtelenül komplex alakzatok. Elöször is elindítjuk az egér kattintásainak feldolgozását az alábbi kódcsipettel:

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

A terület kijelölése és feldolgozása úgy történik, hogy bekérjük a kijelölendő terület bal felső sarkát:

```
public void mousePressed(java.awt.event.MouseEvent m) {
    x = m.getX();
    y = m.getY();
    mx = 0;
    my = 0;
    repaint();
}
```

Majd húzással kijelöljük a nagyítandó területet, amit utána újraszámítunk:

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

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: Polargen C++; Polargen Java

Remek OO bevezető példa lehet egy alább itt is bemutatásra kerülő polártranszformációs normális generátor megírása Javaban. A java verzió megvalósításához szükségünk lesz az OpenJDK 6-os verziójára, mindjárt meg is tudjuk, miért. A programunk megírása után a JDK src.zip állományában a java/util/Random.java forrásban láthatjuk, hogy magában a JDK-ban is hasonlóan oldották meg a feladatot a Sun programozói!

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

```
synchronized public double nextGaussian() {
    //See Knutch, ACP, Section 3.4.1 Algorithm C.
    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;
    }
```

Elkészítjük a módosított polármódszeres algoritmust Javaban, majd C++ nyelven. Az algoritmusunk matematikai háttere most számunkra lényegtelen, fontos viszont az eljárás azon jellemzője, hogy egy számítási lépés két normális eloszlású számot állít elő, tehát minden páratlanadik meghíváskor nem kell számolnunk, csupán az előző lépés másik számát visszaadnunk. Ha a programot lefordítjuk és futtatjuk, akkor a következő kimenetet kaphatjuk:

```
-0.7302435745349951
0.3398973333782606
-0.1745186408410782
-0.6733291138289893
-0.7141255333702377
0.8105205642319349
-0.2166963741203095
-0.6100935726625737
-0.0061257158500475665
0.09213084665478943
```

6.2. LZW

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

Megoldás videó:

Megoldás forrása: LZWbinFa

Fa alatt egy olyan rekurzív adatszerkezetet értünk, amely belső és külső csúcsok hierarchikus (szülő-gyermek) elrendezéséből áll. A bináris fa legfőbb jellemzője az, hogy mindegyik csomópontnak csak két utóda lehet. A bináris fák utódjait megkülönböztetjük aszerint, hogy bal illetve jobb részfák. A programunk úgy működik, hogy megadunk neki egy szöveges fájlt, a program pedig lebontja a karaktereket binfa szerkezetre, ám hogy ezt megtekintsük, meg kell adnunk neki egy kimenetet is, hogy később megnézhessük azt.

```
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;
```

```
}
```

A fentebbi kódrészletben definiáltuk a binfa struktúrát, valamint a BINFA_PTR-t, ami a felhasználó által definiált típus lesz. A következőkben megnézzük, hogyan is épül fel a binfa:

```
int
main (int argc, char **argv)
{
  char b;
  int egy_e;
  int i;
  unsigned char c;

  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);
```

Fentebb láthatjuk, hogyan is működik a bináris fa. Létrehozzuk a gyökeret, majd balra és jobbra is elkezdjük létrehozni a gyerekeket. Hogy megtudjuk pontosan mennyi származtatásról is beszélünk, meg kell néznünk az átlagos águhosszat, vagyis a szórás összegét, az átlagot és a mélységet.

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

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

  /* Átlagos águhossz 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;

  /* Águhosszak 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);
}
```

Az **atlag()** függvényt inicializálnunk kell, mert azt a rekurzív bejárás használja:

```
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;

  }

    }

}
```

A szórással is eljátsszuk ugyanezt, tehát inicializálunk:

```
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));

  }

    }
```

```
}
```

A következő kódrészletben megírjuk a **kiir** függvényt, ami az inorder fabejárásért felelős, tehát az outputban a gyökér valahol a gyerekek között lesz.

```
int max_melyseg = 0;

void
kiir (BINFA_PTR elem)
{
  if (elem != NULL)
    {
      ++melyseg;
      _melyseif (melyseg > maxg);
  max_melyseg = melyseg;
      kiir (elem->jobb_egy);
      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;
    }
}
```

Végül felszabadítjuk az elemeket:

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

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: pre; post

Nem kell nagyon belenyúlni a binfa programba, mikor a fabejárást szeretnénk állítani. az inorder részt már az előző részben elmagyaráztuk, ennek alapján már nem lesz nehéz kitalálni, mire való a másik két bejárási módszer. a preorder azt jelenti, hogy a gyökeret az outputban elsőként írja fel, majd utána írja a többi

csomópontot(vagy gyereket). A kódban egyszerűen annyit kell átírnunk, hogy felcseréljük a csomópontok kiírásának sorrendjét. A preorder tehát így néz ki:

```
void
kiir (BINFA_PTR elem)
{
  if (elem != NULL)
    {
      ++melyseg;
      _melyseif (melyseg > maxg);
  max_melyseg = melyseg;
      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);
      kiir (elem->jobb_egy);
      --melyseg;
    }
}
```

A postorder pedig így:

```
void
kiir (BINFA_PTR elem)
{
  if (elem != NULL)
    {
      ++melyseg;
      _melyseif (melyseg > maxg);
  max_melyseg = melyseg;
      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);
      kiir (elem->jobb_egy);
      --melyseg;
    }
}
```

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: z3a7.cpp

Az LZW binfa C++ implementációja azzal kezdődik, hogy az egész fát egy **LZWBinFa** osztályba ágyazzuk. Az osztály definíciójába beágyazzuk a fa egy csomópontjának jellemzését, ez lesz a beágyazott Csomopont osztály. Azért ágyazzuk be, mert külön nem szánunk neki szerepet, csak a fa részeként számolunk vele.

```
class LZWBinFa
{
public:

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

Ebben az osztályban a fa gyökere nem pointer, hanem a '/' betűt tartalmazó objektum; a fa viszont már pointer, mindig az épülő LZW-fa azon csomópontjára mutat, amit az input feldolgozásakor az LZW algoritmus diktál. Ez a konstruktor annyit csinál, hogy a fa mutatót ráállítja a gyökérre. Tagfüggvényként túlterheljük a << operátort, mert ekkor úgy nyomhatjuk a fába az inputot, hogy **binFa << b**; ahol a b egy '0' vagy '1'-es karakter. Mivel ezen operátor tagfüggvény, így van rá "értelmezve" az aktuális (this "rejtett paraméterként" kapott) példány, azaz annak a fának amibe éppen be akarjuk nyomni a b betűt a tagjai ("fa", "gyoker") használhatóak a függvényben.

```
void operator<< (char b)
    {
        // Mit kell betenni éppen, '0'-t?
        if (b == '0')
        {
            /* Van '0'-s gyermeke az aktuális csomópontnak?
           megkérdezzük Tőle, a "fa" mutató éppen reá mutat */
            if (!fa->nullasGyermek ())  // ha nincs, hát akkor csinálunk
            {
                // elkészítjük, azaz példányosítunk a '0' betű akt.  ↩
                   parammal
                Csomopont *uj = new Csomopont ('0');
                // az aktuális csomópontnak, ahol állunk azt üzenjük, hogy
                // jegyezze már be magának, hogy nullás gyereke mostantól  ↩
                   van
                // küldjük is Neki a gyerek címét:
                fa->ujNullasGyermek (uj);
                // és visszaállunk a gyökérre (mert ezt diktálja az alg.)
                fa = &gyoker;
            }
            else      // ha van, arra rálépünk
            {
```

```
                // azaz a "fa" pointer már majd a szóban forgó gyermekre ↩
                   mutat:
                fa = fa->nullasGyermek ();
            }
        }
        // Mit kell betenni éppen, vagy '1'-et?
        else
        {
            if (!fa->egyesGyermek ())
            {
                Csomopont *uj = new Csomopont ('1');
                fa->ujEgyesGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->egyesGyermek ();
            }
        }
    }
```

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: z3a7_pointer

Alapul vesszük az előző C++ kódot, ebben fogjuk átírni a szükséges dolgokat. Először átírjuk a **protected** osztályban a gyökér csomópontot pointerré:

```
Csomopont *gyoker;
```

Majd elkezdhetjük a többi kódrészlet átírását, például a legelején a konstruktorra Kezdjük már is a legelején, a konstruktort írjuk át erre:

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

Szintén a konstruktorban, mivel már mutató a gyökér, ezért a **szabadit()** függvényünkbe is bele kell nyúlnunk kicsit, mutatót állítunk a gyermekekre:

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

Végül a **&gyoker** kifejezéseket kicseréljük simán **gyoker**-re, ugyanis nekünk nem a pointer címére lesz szükségünk, hanem arra a címre amire az mutat.

6.6. Mozgató szemantika

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

Tutor: Győri Márk Patrik

Megoldás forrása: mozgato

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

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

```
void rekurzioInditasa(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 **rekurzioInditasa** 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 node-okat.

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

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

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

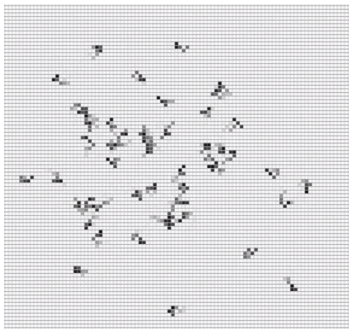
Írj Qt C++-ban egy hangyaszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

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

Tutor: Győri Márk Patrik

Megoldás forrása: ant

A **main** függvény bekéri az argumentumokat, majd deklarálkja az **AntWin** osztályt (w) a bemeneti argumentumokkal, amely felelős a megjelenítésért. Az **AntThread** osztály végzi a számolást és a "hangyák" lehelyezését. Itt az `AntThread::newDir` számolja ki, hogy melyik irányba menjen a hangya. Ennek a számolása a AntThread::moveAnts függvénnyel történik. Az AntThread külön szálon fut, így megállítható, és újra indítható. Erre bindeltük a "P" betűt, amit a keyPressEvent-nél találunk.



7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

SKIP

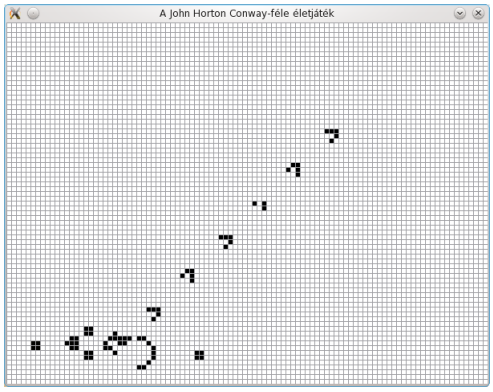
7.3. Qt C++ életjáték

Most Qt C++-ban!

Tutor: Győri Márk Patrik

Megoldás forrása:

Az elvet John Horton Conway dolgozta ki. A lényege, ha egy cella egyedül van, vagy nincs szomszédja, akkor eltűnik. Ha négy, vagy több szomszédja van akkor is eltűnik. Ha kettő vagy három szomszédja van, akkor megmarad. Minden cella, aminek három szomszédja van az tovább osztódik. Ez a műveletsor ismétlődik végtelenszer. Ha például egy 2x2-es cella van, és nincsen szomszédja akkor nem történik semmi, mivel alkalmazzuk rá a fenti szabályokat.

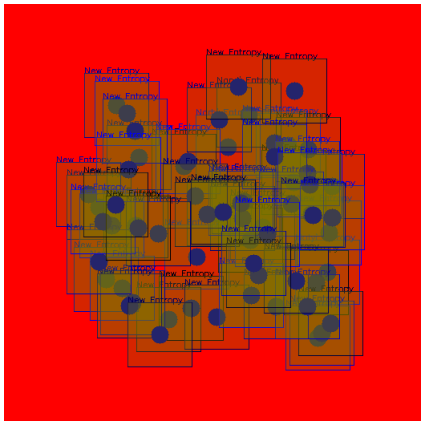


7.4. BrainB Benchmark

Tutor: Győri Márk Patrik

Megoldás forrása: BrainB

A játék lényege, hogy a **Samu Entropy** nevezetű négyztet kell megkeresnünk a többi négyzet közül, és benne kell tartanunk a kurzort a kék körben. A konzol kiírja, ha véletlen hibáznánk, és méri, hogy mennyi bits/sec a reakció időnk. Egyre több hamis entropy jelenik meg a képernyőn, szóval egyre nehezebb lesz a dolgunk



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

Az SMNIST egy gépi tanulásra szóló program, ezért tensorflow-t használunk alapnak. Ezen program azt szolgálja, hogy megszámolja, hány fekete négyzetet lát egy adott képen. Ehhez le kell generálnunk pár képet, legyen ez most 60000 darab kép, hogy minél kevesebb legyen az ismétlődés.

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása: https://github.com/tensorflow/tensorflow/blob/r1.4/tensorflow/examples/tutorials/mnist/-mnist_deep.py

Tanulságok, tapasztalatok, magyarázat...

8.3. Minecraft-MALMÖ

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Megoldás forrása:

A Lisp-hez elegendő egy GIMP-et megnyitnunk, azon belül pedig a Script-fu konzolt kell elindítanunk, majd kezdődhet is a Lisp-es kalandunk. Még mielőtt a faktoriálist írnánk meg, kicsit ismerkedjünk a Lisp nyelvvel. A lényege az, hogy ha azt akarjuk kideríteni, mennyi 2+2+2, akkor csak annyit írunk, hogy **(+ 2 2 2)**. Tudunk függvényt define-olni, például így: **(define (fakt n))**. A lényeg, hogy figyeljünk a zárójelekre, valamint a szóközökre, mert el lehet veszni bennük. Nézzük a várva várt faktoriális megoldást:

```
(define (fakt n) (if (< n 1 ) 1 (* n (fakt(- n 1)))))
```

Ezután ha beírjuk például a **(fakt 5)** függvényt, akkor 120-at kapunk vissza, ami megfelel az **5!**-nak

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

Ahhoz, hogy futtatni tudjuk GIMP-en belül a létrehozott scriptünket, be kell másolni a kódot a GIMP script mappájába. Majd a programon belül: **File/Create/BHAX/bhax_chrome3** és már szórakozhatunk is kedvünkre a kívánt szövegünk átszínezésével. Ha futtatjuk a scriptet, egy alap beállítást is kapunk, amely megadja, hogy 1000x1000 képpontos kép legyen, valamint beállítja a fontot és a font méretét. A színt és a színátmenetet is beállíthatjuk. Nézzük a kód fő lépéseit, ami alapján felépül a króm effekt:

```
   (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)))
```

Az első lépésben megadunk egy fekete színnel kitöltött hátteret, majd jelezzük, hogy a szövegünk fehér színű legyen, a krómosítást pedig később fogjuk elvégezni. A második lépésben az ún. Gaussian módszerrel elmossuk a szöveget:

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

Majd a harmadik lépésben lekanyarítjuk a szöveg éleit: (gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE) Végül ezt is elmossuk: (plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE) Invertáljuk a képet, kitöröljük a fekete hátteret, így kapunk egy átlátszót, majd foglalkozunk a szöveggel. A lényeg a kilencedik lépés, ekkor egy Spline görbével meg tudjuk adni a gradient effektet: (gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))

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

A scripft felépítése hasonló az előző feladatban látott script felépítéséhez, ugyanúgy script-fu-t használunk, tehát GIMP-el foglalkozunk még mindig. A mandala lényege, hogy tükrözgetjük és forgatjuk a szövegünket, amíg egy szép kis képet nem kapunk belőle:

```
(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))
```

Majd ezután két kört rajzolunk, amelyeknek eltérő a vastagságuk (22 és 8):

```
(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)
```

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

Kezdjük az alapfogalmakkal: A programozási nyelveket három szintre tudjuk bontani: gépi nyelv, assembly szintű nyelv és magas szintű nyelv. Mi az utolsó, magas szintű nyelvvel fogunk foglalkozni. Az ilyen szinten írt programokat forrásprogramoknak nevezzük. A magas szintű nyelven írt forrásból gépi nyelvű programot kell alkotnunk; erre létezik interpreteres és fordítóprogramos megoldás is. A **fordítóprogram** egy olyan program, amely a forrásunkból tárgyprogramot állít elő, amit tud kezelni a gép. Négy lépést hajt végre ez a folyamat, a lexikális elemzést, szintaktikai elemzést, szemantikai elemzést, majd legenerálja a kódot. Az interpreteres megoldásnál is ugyanaz az első három lépés, de az interpreter nem csinál belőle tárgyprogramot. Minden utasításnál sorra veszi a forrást, értelmezi majd végrehajtja azt. Itt egyből megkapjuk az eredményt egy gépi rutin szerint. A programnyelveket osztályozni tudjuk: Vannak imperatív nyelvek, deklaratív nyelvek, és másnyelvű nyelvek.

Beszéljünk a kifejezésekről. A kifejezések olyan szintaktikai eszközök, amik arra valók, hogy a program adott pontján ismert értékekből új értéket tudunk meghatározni. Ennek két komponense van, érték és típus komponens. Egy kifejezés három összetevőből áll: operandusok, operátorok és kerek zárójelek. A kifejezéseknek három alakja lehet, attól függ, hol áll az operátor az operandusokhoz képest: **prefix**, amikor az operátor az operandusok előtt van; **infix**, amikor az operátor az operandusok között helyezkedik el; valamint **postfix**, amikor az operátort az operandusok mögé tesszük. Ezek közül az infix alak nem mindig egyértelmű. Az utasításokról: 9 csoportra bonthatjuk őket: Értékadó, üres, ugró, elágaztató, ciklusszervező, hívó, vezérlésátadó, I/O valamint egyéb utasítások. Az elágaztató utasításokat is tovább bonthatjuk: Kétirányú elágaztató utasítás, vagyis feltételes utasítás (IF ... THEN); Többirányú elágaztató utasítás, ennek szintaktikája és szemantikája nyelvenként különböző. A ciklusszervező utasításoknak köszönhetően egy tevékenységet bármennyiszer megismételhetünk. Ezek felépítése a fejből, magból és a végből épül fel. A feltételes ciklusnak is két fajtája van: Kezdőfeltételes ciklus, amikor a feltétel a fejben jelenik meg; valamint végfeltételes ciklus, amikor a feltétel a végben jelenik meg.

A programok szerkezetét nézegetve felmerül a kérdés, hogy egyben kell-e lefordítani a teljes szöveget, vagy fel lehet tördelni azt? Erre három válasz is létezik. Néhány nyelven a program önálló részekből áll, ezek külön fordíthatóak, nem struktúráltak. Más nyelvekben az adott programot egy egységként kell lefordítanunk. Ez a program struktúrálható, az egységei nem függetlenek. Az utolsó esetként a kettő kombinációja is létezik. Ezek független programegységek, de tetszőleges struktúrával rendelkeznek.

Paraméterátadás. Ez a programegységek és az alprogramok közötti kommunikáció eszköze. Itt mindig van egy hívó és egy hívott. Ezek a módok léteznek paraméterátadásra: érték, cím, eredmény, érték-eredmény,

név és szöveg szerinti paraméterátadás. Az aktuális paraméterek lehetnek az alábbiak: érték szerinti: kifejezés -- cím szerinti: változó -- eredmény szerinti: változó -- érték-eredmény szerinti: változó. A következő témánk a blokk. Ez olyan programegység, amely csak és kizárólag más programegység belsejében helyezkedik el, kívül nem állhat. A blokk három részből épül fel; van kezdete, törzse és vége. A kezdetet jelzi egy szó vagy speciális karakterek; A törzsben végrehajtó és deklarációs utasítások is előfordulhatnak. A blokkoknak nincsenek paraméterei, bárhol elhelyezhető, ahol végrehajtó utasítás van. Egy blokkot többféleképp is tudunk aktivizálni.

Nézzük mit érdemes tudni az I/O-ról. Ez a programnyelvek azon területe, ahol a leginkább eltérnek a többi programnyelvtől. Az I/O olyan eszközrendszer, ami a perifériákkal kommunikál. Ennek középpontjában az állomány van. Az adott állomány funkció szerint tekintve lehet input állomány, output állomány valamint mindkettő, tehát input-output állomány. Nézzünk pár példát, hogy egyes nyelvek hogyan kezelik ezeket az I/O eszközöket. Például a FORTRAN csak fix rekordformátumot képes kezelni, a COBOL-nak erős az I/O rendszere, mert mindig konvertál. A PL/I a legjobb állománykezeléshez. Ezzel ellentétben a PASCAL állománykezelése nem túl jó. Ami számomra releváns, az a C, ennek a nyelvaz I/O eszközrendszer nem

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

Elsőként a vezérlési szerkezetekkel foglalkozunk. A vezérlésátadó utasítások azt határozzák meg, milyen sorrendben hajtsa végre a számítógép a számításokat. Foglalkozzunk a különféle utasításokkal, annak típusaival. Az első ilyen típus az **egyszerű kifejezések**, az is lehet utasítás ha egy kifejezést, például egy **x = 1**-et pontosvesszővel zárunk. A kapcsos zárójelekkel behatárolt deklarációkat egy blokknak, összetett utasításnak veszünk. A következő típus az **if-else** és az **else if** utasítások. Az if-else utasítással választást, döntést írunk le, míg az else if megoldással többszörös elágazást készíthetünk. Ha több irányba el akarjuk ágaztatni a programot, ahhoz a **switch** utasítás remek eszköz. A switch kiértékeli azt a kifejezést, amit a zárójelek közé rakunk, majd megnézi, hogy az adott érték megegyezik-e a case-ben tartott értékkel; és ha talál ilyet, végrehajtja a case-en belüli utasításokat. Ezzel megspórolhatunk jópár if-else utasítást. Minden case-en belüli utasítás végére kell egy **break;** függvény, ezzel tudatjuk a programmal, hogy itt ér véget az adott utasításunk. Ha megírtuk az összes case-t amit szerettünk volna, meg kell adnunk egy **default** értéket, amit akkor ír ki a program, ha egyik case-nek sem felel meg az értékünk.

A következő, amivel foglalkozunk, az a **for** és a **while** utasítások. A **while()** ciklusban a gépünk kiértékeli a zárójelben lévő kifejezést, és ha az nem nulla, akkor végrehajtja a kiadott utasítást, majd újra kiértékeli a kifejezést. Ez a ciklus addig megy, amíg az érték 0 nem lesz. A **while** ciklus általános szerkezete:

```
while(valami)
    utasítások
```

A **for** szerkezetre példa:

```
for(x1,x2,x3)
    utasítások
```

ezt át tudnánk alakítani **while** szerkezetre is, az alábbi módon:

```
x1;
while (x2) {
    utasítások
    x3;
}
```

A **for** utasítás mindhárom összetevője (x1,x2,x3) kifejezés. Általában az x1, x3 függvényhívás vagy értékadás, az x2 pedig relációs kifejezés. El lehet hagyni a kifejezések bármelyikét, de a lényeg, hogy a pontosvessző maradjon. Ha a függvényhívás/értékadás elmarad, akkor a pontosvessző elmarad a kifejtésből; ha viszont az x2 marad el, akkor igaznak tekinthetjük az alábbi kódot:

```
for(;;) {
    kódcsipet
}
```

ami egy végtelen ciklus. Ki tudunk belőle ugrani egy **break** vagy **return** függvénnyel.

A következő amit megnézünk az a **do-while** utasítás lesz. Ez abban különbözik a **while** és a **for** utasításoktól, hogy míg ez a kettő a kiugrási feltétel teljesülését a ciklus elején vizsgálja, addig a **do-while** mindezt a ciklus végén teszi meg. nézzük is hogyan épül ez fel:

```
do
            valami
        while(kif);
```

Először az utasítást(valami) hajtja végre, majd a kifejezést értékeli ki(kif). Ha igaz, akkor végrehajtja az utasítást. Ha hamis, akkor a ciklus véget ér. Ezt a megoldást ritkábban szokás használni.

A soron következő utasítás a **break** utasítás. Ezt azért használjuk, mert kényelmesebb néha úgy kilépni a ciklusból, hogy nem függvényvizsgálattal tesszük azt. A **break** utasítástól a vezérlés egyből kilép a legbelső ciklusból vagy switchből.

Az utolsó előtti utasításunk a **continue** utasítás. Ez az utasítás a **break**-hez kapcsolódik, de ritkábban használjuk. A while és a do esetében azonnal vizsgálja a feltételt, a for esetében pedig átlép az újrainicializálás lépésre. A **continue** csak a for, while, do ciklusokra alkalmazható, switch-re nem.

A legutolsó utasításunk a **goto** utasítás. Ezzel címkékre tudunk ugratni. Eredetileg erre soha nincs szükségünk, ezért elég is ennyi róla.

10.3. Programozás

[BMECPP]

Kezdjük a kivételkezelés alapjaival. A kivételkezelés, angolul exception handling, olyan mechanizmus, ami ha hibát talál valahol akkor a program futása egyből a hibakezelő részen folytatódik. Ne higyjük azt, hogy csak hibákat tudunk ezzel kiszűrni, mert **kivétel**kezelésről beszélünk, és nem **hiba**kezelésről. A kivételkezelő bármilyen kivételes helyzetben használható. Próbáljuk ki a kivételkezelést kódban is megnézni. Tegyük fel, hogy osztani akarunk egy olyan számmal, amit a felhasználó ad meg. De mi van, ha a felhasználó 0-t ír? Azt szeretnénk, ha ezt jelezné a felhasználó felé, ezért még mielőtt eljut a felhasználóhoz,

kipróbáljuk magunknak egy **try-catch blokk** alkalmazásával. A lényeg az, hogy a **try** védett blokkba beírjuk úgy a kódot, ahogy szeretnénk, hogy működjön. Ugyanebben a blokkban megpróbálunk dobni neki egy kivételt a **throw** függvénnyel, így szeretnénk egy exception-t dobni a kódnak. Az életben is általában úgy van, ha eldobnak egy labdát, nem árt azt elkapni. Ugyanezt csináljuk a kódunkkal, a dobott kivételt elkapjuk egy **catch** blokkban. Dobáskor a futás egyből arra a catch-re ugrik, amely paraméterei megfelelnek a dobott kivétel paramétereivel. A kivétel elkapásakor az be fog íródni a catch paramétere közé, majd lefut a catch törzse. Ha szeretnénk, meg is tekinthetjük, mit kapott el a kivételkezelő, ezt úgy tudjuk megnézni, ha a catch zárójelei közé létrehozunk egy **const char*** típusú változót. Gyakorlatban így nézne ki maga az elkapás:

```
catch (const char* kivetel)
    {
        std::cout << "Hiba: " << kivetel << std::endl;
    }
```

A kivételkezelés egyik legfontosabb jellemzője az, hogy ha egy try blokkban függvényhívásokra kerül sor, akármilyen mélyen is van a **throw**, a futás egyből a **catch** ágon megy tovább.

Tudunk olyat is készíteni, hogy egy try-catch blokkhoz **két catch** ágat rendelünk. A throw-nak megadunk egy paramétert, ezt a paramétert **kivételobjektumnak** nevezni. Ha egy kivételt nem sikerül elkapni, kezeletlen kivételnek tekintjük. Ebben az esetben egy függvény kilép az alkalmazásból. Olyat is tudunk csinálni hogy egymásba ágyazunk több **try-catch** blokkot. Ez azért hasznos, mert így néhány kivételt a dobott kivételünkhöz közelebb tudjuk kezelni, egy fokkal alacsonyabb szintről.

Foglalkozzunk egy kicsit az operátorokkal. Ezek C nyelven az argumentumaikon végzik el a műveleteket. A C++ nyelvbe bevezettek pár új operátort a C-hez képest. Az egyik ilyen a hatókör operátor (::). Ezt már láttuk sokszor C++ programozás során, csak hogy soroljak párat példának: std::cout, mozgató szemantikánál std::move kipróbálása. Újítás még a pointer-tag operátorok: **.*** és **->*** A következő a túlterhelésről és a függvény szintaxis. C++ nyelven már egyszer használtuk az **operator**-t, ezzel jeleztük a program felé, hogy speciális, saját függvényről van szó. Erre a mozgató szemantikás példát nézzük meg újra:

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

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.