

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert Ács Fűrjes-Benke, Péter	2019. március 19.	

REVISION HISTORY

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

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	9
2.4. Labdapattogás	10
2.5. Szóhossz és a Linus Torvalds féle BogomIPS	12
2.6. Helló, Google!	14
2.7. 100 éves a Brun tétel	16
2.8. A Monty Hall probléma	18
3. Helló, Chomsky!	21
3.1. Decimálisból unárisba átváltó Turing gép	21
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	22
3.3. Hivatkozási nyelv	24
3.4. Saját lexikális elemző	26
3.5. l33t.1	28
3.6. A források olvasása	30
3.7. Logikus	33
3.8. Deklaráció	34

4. Helló, Caesar!	37
4.1. int ** háromszögmátrix	37
4.2. C EXOR titkosító	43
4.3. Java EXOR titkosító	47
4.4. C EXOR törő	50
4.5. Neurális OR, AND és EXOR kapu	54
4.6. Hiba-visszaterjesztéssel perceptron	59
5. Helló, Mandelbrot!	65
5.1. A Mandelbrot halmaz	65
5.2. A Mandelbrot halmaz a std::complex osztállyal	65
5.3. Biomorfok	65
5.4. A Mandelbrot halmaz CUDA megvalósítása	65
5.5. Mandelbrot nagyító és utazó C++ nyelven	65
5.6. Mandelbrot nagyító és utazó Java nyelven	66
6. Helló, Welch!	67
6.1. Első osztályom	67
6.2. LZW	67
6.3. Fabejárás	67
6.4. Tag a gyökér	67
6.5. Mutató a gyökér	68
6.6. Mozgató szemantika	68
7. Helló, Conway!	69
7.1. Hangyaszimulációk	69
7.2. Java életjáték	69
7.3. Qt C++ életjáték	69
7.4. BrainB Benchmark	70
8. Helló, Schwarzenegger!	71
8.1. Szoftmax Py MNIST	71
8.2. Szoftmax R MNIST	71
8.3. Mély MNIST	71
8.4. Deep dream	71
8.5. Robotpszichológia	72

9. Helló, Chaitin!	73
9.1. Iteratív és rekurzív faktoriális Lisp-ben	73
9.2. Weizenbaum Eliza programja	73
9.3. Gimp Scheme Script-fu: króm effekt	73
9.4. Gimp Scheme Script-fu: név mandala	73
9.5. Lambda	74
9.6. Omega	74
10. Helló, Kernighan!	75
10.1. Pici könyv	75
10.2. K&R könyv	78
10.3. BME C++ könyv 1-16. oldal feldolgozás	81
III. Második felvonás	84
11. Helló, Arroway!	86
11.1. A BPP algoritmus Java megvalósítása	86
11.2. Java osztályok a Pi-ben	86
IV. Irodalomjegyzék	87
11.3. Általános	88
11.4. C	88
11.5. C++	88
11.6. Lisp	88

Ábrák jegyzéke

2.1. Megállási probléma	9
2.2. PageRank	15
2.3. Brun-tétel	18
3.1. Decimálisból unárisba	21
3.2. 1. Splint kép	26
3.3. 1. Splint kép	32
3.4. 2. Splint kép	33
4.1. Háromszögmátrix	38
4.2. Pointerek a memóriában	39
4.3. $*(tm + 3)[1] = 43$	42
4.4. $*(tm + 3)[1] = 43$	43
4.5. Titkosítandó szöveg	45
4.6. Fordítás és futtatás	46
4.7. Titkosított szöveg	47
4.8. Titkosított szöveg	49
4.9. Törés	53
4.10. Törés	54
4.11. Neuron	55
4.12. OR	56
4.13. AND	57
4.14. EXOR első próba	58
4.15. EXOR második próba	59
4.16. Perceptron bemenet	60
4.17. mandelpng.cpp fordítása és futtatása	61
4.18. Mandelbrot-halmaz	62
4.19. Fordítás és futtatás	64

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 (hogyan lássuk mást is) példával.

Hogyan nyomjuk?

Ránts 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 dlatex 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
dlatex 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 'dlatex' 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 találsz 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.

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Végtelen ciklusnak olyan ciklusokat hívunk, melyek soha nem érnek véget, általában valamilyen logikai hiba miatt. Pl:

Program pelda

```
{
  int main()
  {
    int i = 0;
    while (i<=0)
    {
      i = i-1;
    }
  }
}
```

Ebben jól láthatod, hogy a while cilusban lévő feltétel folyamatosan teljesül, theát a program végtelen ciklusba kerül. Viszont, ha kifejezetten végtelen ciklust szeretnél írni, ennek a legelegánsabb módja a következő:

Program vegtelen.c

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

Ez a végtelen ciklus csak egy magot dolgoztat, de azt 100%. A lényege annyi, hogy a for ciklus nem kap semmilyen argumentumot, ennek következtében a ciklus előtti teszt folyamatosan igazat fog adni, tehát a ciklus nem fejeződik be.

De nem elégszünk meg az egy maggal, hiszen ma már a legtöbb számítógép legalább 4 maggal rendelkezik. Tehát találni kell egy megoldást, hogy az összes mag dolgozzon 100%-on. Ezt oldja meg az OpenMP.(Bővebben [itt](#) olvashatsz erről.) A lényege annyi, hogy program több szálon dolgozhat, így kihasználva a rendelkezésre álló erőforrásokat. Ráadásul ez könnyen implementálható az előző kódunkba:

```
Program vegtelen_all.c
{
    int main()
    {
        #pragma omp parallel
        for(;;);
    }
}
```

Amint látod, csak a `#pragma omp parallel` sort kellett hozzáadni. Természetesen ezt bármelyik programnál használhatod, sőt javasolt is, köszönhetően a hardverek gyors fejlődésének.

Még egy dologgal adós vagyok. Már láttad, hogy hogyan lehet megoldani, hogy egy végtelen ciklus 100%-ban használjon egy magot, majd azt is, hogyan használjon 100%-ban a processzort. Itt az idő megnézni, hogyan lehet elérni, hogy egy végtelen ciklus egyáltalán ne használjon (vagyis nagyon keveset) a CPU által biztosított erőforrásból. Ehhez a `sleep` függvényre lesz szükségünk. A kód a következő:

```
Program vegtelen_s.c
{
    int main()
    {
        for(;;)
        {
            sleep(1);
        }
    }
}
```

A `sleep` függvény lényegében minden meghívásánál "aludni" küldi azt a szálát, amit program használna, jelen esetben 1 másodpercig altatja.

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

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

Megoldás videó:

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


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

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

A program futtatása, például akár az előző v. c ilyen 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 építő Lefagy2 már nem tartalmaz feltételezett, 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?

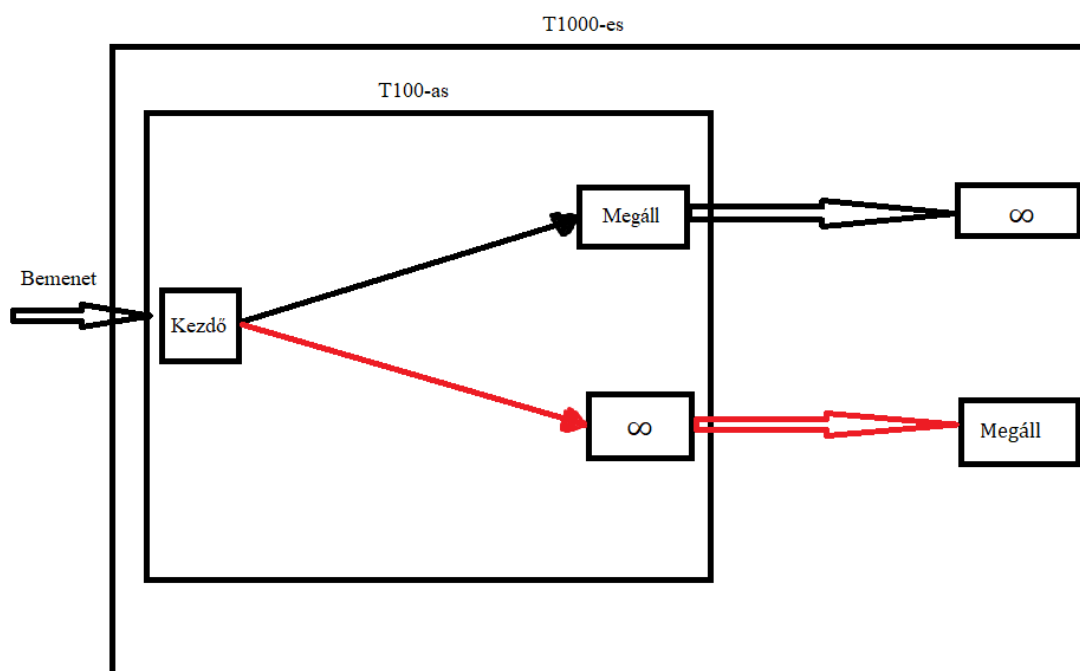
- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true
- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

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

Tanulságok, tapasztalatok, magyarázat...

Alan Turing a XX.század egyik legjelentősebb brit matematikusa, a modern számítógép-tudomány atyja. Az ő nevéhez fűződik a Turing-gép, mint fogalom, ezt 1936-ban dolgozta ki. De egy másik fontos eredménye is van, hiszen az ő segítségével sikerült a szövetségeseknek feltörni a Náci Németország titkosító berendezését, az Enigmát. Enélkül talán soha nem sikerült volna véget vetni a II. világháborúnak, de az biztos, hogy sokkal több embervesztéssel járt volna. Ezt a történetet dolgozza a fel a [Kódjátzsma](#) című film.

De visszatérve a Turing-géphez, ez egy 3 főbb fizikai egységből áll: egy cellákra osztott papírszalagból, egy vezérlőegységből és egy író-olvasó fejből. A működése nagyjából abból áll, hogy az író-olvasó fej a szalagon beolvas egy cellát, módosítja, majd tovább mozog. Ez folytatódik minden iterációban. Ennek két-féle kimenetele lehet, vagy megáll a "program", vagy végtelen ciklusba kerül. Egy másik fontos fogalom az Univerzális Turing-gép, melynek lényege az, hogy egy bemenetre ugyan azt az eredményt adja mindegyik Turing-gépen. Ezzel eljutottunk a megállási problémához, mely mind a mai napig megoldhatatlan probléma elé állította a számítógép-tudományt. A megállási probléma azt mondja ki, hogy nem tudunk olyan programot írni, amely meg tudja mondani egy másiktól, hogy az le fog-e fagyni, azaz végtelen ciklusba kerül-e, vagy sem. Ez az ábra szemlélteti a forrásban leírtakat:



2.1. ábra. Megállási probléma

Tehát a T100-as program kap egy programot bemenetként, és arról eldönti, hogy az megáll-e vagy sem. Ebben az esetben a bemeneti programot úgy kell elképzelni, mint egy szöveges fájlt. A program beolvassa és eldönti, hogy van-e benne végtelen ciklus. A T100-as visszaad egy igaz/hamis értéket. Ezt átadjuk a T1000-es programnak, mely ha a bemenet igaz, akkor megáll, ha az érték hamis, akkor pedig végtelen ciklusba kerül. Itt azt hihetnén, hogy minden rendben, hiszen a program működik, de mi történik akkor, ha a T1000-es program bemenete önmaga. Ha úgy érzékeli, hogy nincs önmagában végtelen ciklus, akkor végtelenciklusba kerül, ha pedig van benne végtelen ciklus, akkor pedig megáll. Itt jól látható az ellentmondás. Ez az oka annak, hogy mind a mai napig nem tudott senki ilyen programot létrehozni.

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: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ha valaki két változónak az értékét fel akarja cserélni, akkor a legegyszerűbb megoldásnak egy segédváltozó bevezetése tűnhet. De természetesen ennél sokkal kifinomultabb eszközök is vannak erre a célra. Az egyik ilyen megoldás, hogy valamilyen matematikai műveletet használunk. Egyik megoldás az, hogy a két

változó értékét összeadjuk, majd ebből az összegből kivonjuk a változók régi értékét úgy, hogy a értékük felcserélődjön. Tehát:

```
int a = 4;
int b = 5;
a = a+b;
b = a-b;
a = a-b;
```

Ez szín tiszta matematika, viszont ennél egy sokkal érdekesebb dolog ugyan ennek a feladatnak EXOR-ral való megvalósítása. A lényeg annyi, hogy a számítógép a változó értékét 2-es számrendszerben tárolja ennek következtében a szám 0-kból és 1-kből áll. A XOR (kizáró vagy) minden esetben 1-et ad vissza, azaz igaz értéket, kivéve ha a művelet jobb és bal oldalán azonos érték van, mert ilyenkor 0-t ad vissza. Ezt használjuk most ki a következő példában:

```
int a = 4; //2-es számrendszerben: 0100
int b = 5; //2-es számrendszerben: 0101
a = a^b; // 0100 ^ 0101 = 0001
b = a^b; //0001 ^ 0101 = 0100
a = a^b //0001 ^ 0100 = 0101
```

A komment szekcióban láthatjátok, hogy mi is történik a háttérben.

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés használata 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:[itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ahogy a videóban láthattátok, a labdapattogás annyiból áll, hogy a terminálon belül egy karakter pattog az az ablak teljes méretében. Fontos, hogy az ablak méretét állíthatjuk, és a program ezt érzékeli.

```
WINDOW *ablak;
ablak = initscr ();
```

Az initscr() beolvassa az ablak adatait, melynek segítségével megtudjuk az ablak méretét. Ezután létrehozunk változókat, melyekben a lépésközt, a pozíciót, és az ablak méretét fogjuk eltárolni.

```
int x = 0; //aktuális pozíció x-tengelyen
int y = 0; //aktuális pozíció y-tengelyen

int xnov = 1; //lépésköz az x-tengelyen
int ynov = 1; //lépésköz az y-tengelyen
```

```
int mx; //ebben lesz eltárolva az ablak szélessége
int my; //ebben pedig az ablak magassága
```

Ezután létrehozunk egy végtelen ciklust a már megszokott módon:

```
for (;;)
{
}
```

És ebbe a végtelen ciklusba fogjuk "pattogtatni" a labdát. Ehhez elsőnek meghívjuk a `getmaxyx()` függvényt melynek átadjuk paraméterként a az ablakban eltárolt értékeket, és azt, hogy melyik változóba tárolja el az ablak hosszúságát és szélességét. És az `mvprintw()` függvény fogja az általunk megadott koordinátákba a karaktert "mozgatni".

```
getmaxyx ( ablak, my , mx );
mvprintw ( y, x, "O" );
```

Mostmár tudjuk az ablak méretét. Az `x` és az `y` változót folyamatosan 1-el növelve a karakter el kezd mozogni a terminálban. Azt, hogy ez milyen gyorsan történjen, azt a `usleep()` függvénnyel tudjuk beállítani. A `usleep` mikroszekundumba számol, tehát az másodperc egymiliomod részében. Ha azt akarjuk, hogy a labda másodpercenként menjen 1-et, akkor 1000000-et kell beírunk a `usleep`-be. Így:

```
usleep(1000000);
```

Most, hogy a labda már mozog, már csak meg kéne állnia az ablak határainál. Ezt pedig `if`-el fogjuk elsősorban megoldani.

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

Tehát, ha a labda eléri valamelyik szélét az ablaknak, akkor a lépésközt megszorozzuk -1-el, így elérve, hogy visszapattanjon.

Egy másik megoldás is létezik ehhez, mégpedig az, ahol nem használunk `if`-et. Ennél a `for`-cikluson belül a következőt írjuk:

```
getmaxyx (ablak, my, mx);
xj = (xj - 1) % mx;
xk = (xk + 1) % mx;

yj = (yj - 1) % my;
yk = (yk + 1) % my;

//clear ();

mvprintw (abs (yj + (my - yk)),
          abs (xj + (mx - xk)), "X");

refresh ();
usleep (150000);
```

Ennél a maradékos osztást használjuk ahhoz, hogy a labda egy bizonyos érték után "visszapattanjon". A lényeg annyi, hogy a modulóval való osztás, mindig annak a számnak az értékét adja vissza, amit elosztunk egészen addig, ameddig egyenlő nem lesz az ablak szélességével/hosszával, mert akkor újra visszaáll 1-re (vagy -1-re), ebben a pillanatban a labda elindul visszafelé, és ez folytatódik végtelen ciklusban. A program futását Ctrl+c-vel tudjátok megállítani. Jelenleg ehhez további magyarázatot nem tudok fűzni, mivel én nekem eszembe se jutott volna ez a megoldás módszer.

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: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ahogy láthattad a forrásban, a gépeden egy szó 32 bites. Hogy ezt kiszámold, arra rengeteg megoldás létezik. Az egyik ilyen a bitshiftelés, melynek lényege az, hogy addig léptetjük a számot, ameddig nulla nem lesz.

```
int a = 1; //kettes számrendszerben: 00000000 00000000 ←
          00000000 00000001
while (a != 0) {
    a <<= 1; //itt léptetjük eggyel: 00000000 00000000 ←
          00000000 00000010
          // újra: 00000000 00000000 00000000 00000100
          //...
}
```

Ha ezt megismételjük 32-ször, akkor végén csak nullából fog állni, mivel ez nem körkörös folyamat, ha az egy elér az elejére, akkor nem fogvisszaugrani a végére. A linkelt forrásban az 1 hexadecimálisan lett megadva, de nyugodtan használhatod az általam írt példát, mivel ugyan azt az eredményt adja.

Ennek a résznek a másik fontos témája a `BogoMIPS`. Ez lényegében egy `Cpu` tesztelő program, melyet Linus Torvalds írt, és a linux kernel része mind a mai napig. A lényege az, hogy a program méri, hogy mennyi idő alatt fut le, ezzel megmondva, hogy a CPU-d milyen gyors. Persze, ha CPU vásárlás előtt állsz, ne pont ez alapján dönts egyik-másik processzor mellett. Számunkra azért érdekes ez a program, mert ennek a `while` fejléce ugyan azt a megoldást használja, amit mi a szóhossz számításához.

```
while (loops_per_sec <= 1)
{
    ;
}
```

Most, hogy láttátok, hogy mi a kapcsolat a mi kis programunk és a Linus Torvalds féle `BogoMIPS` között, akkor lássuk is, hogy hogyan működik pontosan. Elsőnek deklarálnunk kell 2 változót, az első a `loops_per_sec`, melynek definiálása során az értékét egyre állítjuk. A bitshiftelésnek köszönhetően ebbe 2 hatványokat fogunk tárolni. A `ticks` pedig a CPU időt fogja tárolni.

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

    ...
}
```

A `while` ciklus addig tart, ameddig a `loops_per_sec` le nem nullázódik. A ciklusba lépve, minden iterációban, lekérdezzük az aktuális CPU időt, és eltároljuk a `ticks` változóban. Ezután pedig meghívjuk a `delay` függvényt.

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

Ez a függvény egy hosszú egész számot kér paraméteréül, és amint látod, csak egy `for` ciklus megy végig 0-tól paraméter-1-ig. Ezután a `while` cikluson belül újra lekérjük az aktuális processzor időt és kivonjuk belőle a kezdeti értéket. Így megkapjuk, hogy mennyi ideig tartott a `cpu`-nak befejeznie a `delay` függvényben lévő `for` ciklus befejezéséig. Ezt egészen addig iteráljuk, ameddig nem teljesül az `if`-ben lévő feltétel.

```
while (loops_per_sec <= 1 )
{
    ...
}
```

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

            printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec/500000,
                    (loops_per_sec/5000) % 100);
            return 0;
        }
    }
```

A CLOCKS_PER_SEC a POSIX szabvány szerinti értéke 1.000.000, tehát akkor teljesül a feltétel, ha a processzor idő ezzel egyenlő, vagy nagyobb. Ezután pedig kiszámoljuk, hogy CLOCKS_PER_SEC idő alatt milyen hosszú ciklust képes végrehajtani a gép. Ennek az eredménye egy viszonylag nagy szám lesz, de a végeredmény megadásához használhatjuk a `log` függvényt, mely sokkal lassabban növekszik, így egy jóval kisebb számot kapunk eredményül. Ehhez csak egy kicsit kell módosítani az előbbi forrást:

```
printf("ok - %lld %f BogoMIPS\n", loops_per_sec, log( ←
        loops_per_sec));
```

2.6. Helló, Google!

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

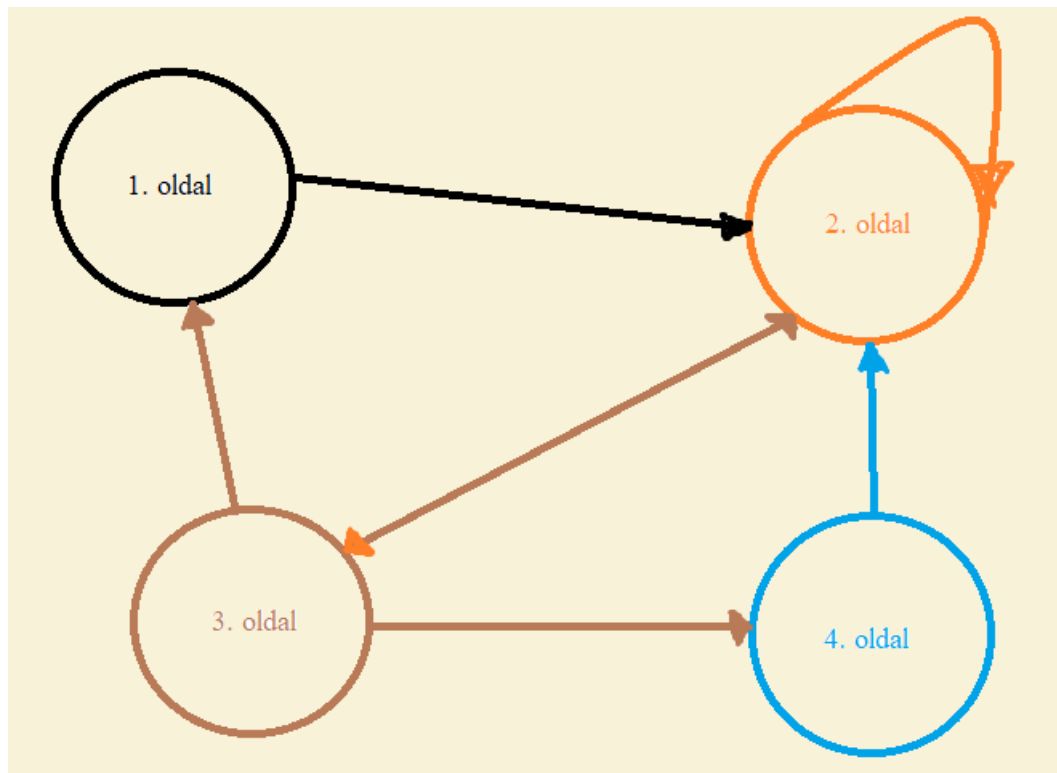
Megoldás videó:

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

A PageRank-et Larry Page és Sergey Brin fejlesztette ki 1998-ban. Az algoritmus az alapja mind a mai napig a Google keresőmotorjának. A lényege az, hogy osztályozza az oldalakat az alapján, hogy hány link mutat rájuk és a rájuk mutató oldalakra hány oldal mutat. Tehát általában az az oldal lesz az első találat, amelyikre a legtöbb oldal hivatkozik. Ez az ötlet megjelenésekor hatalmas forradalmat hozott az internetes keresők világában, hiszen a pontos találatok száma jelentősen megnőtt. Az előtt csak álm volt, hogy már az első találat az lesz, amit éppen keres, de a PageRank-kel ez valósággá vált. Ezt a forradalmi algoritmusnak láthatod a forrásban is, egy 4 honlapos hálózatra leszűkítve.

Az első iterációban mindegyik oldal PageRank-je 1/4, hiszen 4 oldalunk van. Ezután kezdjük el vizsgálni, hogy ezek között milyen kapcsolatvan.



2.2. ábra. PageRank

A kapcsolatokat egy mátrixban tároljuk:

```
double L[4][4] = {
    /*1.*/* /*2.*/* /*3.*/* /*4.*/*
    /*1.*/{0.0, 0.0, 1.0/3.0, 0.0},
    /*2.*/{1.0, 1.0/2.0, 1.0/3.0, 1.0},
    /*3.*/{0.0, 1.0/2.0, 0.0, 0.0},
    /*4.*/{0.0, 0.0, 1.0/3.0, 0.0}
};
```

Az sorok és oszlopok metszetében láthatod, hogy milyen kapcsolatvan van az oldalak között. Ezt a mátrixot adjuk át argumentumként a `pagerank()` függvénynek, mely így néz ki:

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

    int i, j;

    for(;;){
```

```
// ide jön a mátrix művelet

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

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

// ide meg az átpakolás PR-ből PRv-be

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

kiir (PR, 4);
}
```

A `PRv[]` tömbben tároljuk az oldalak első iterációbeli értékét, és a `PR` tároljuk el a mátrixszorzás eredményét. A mátrixszorzást a `L` és `PRv` tömb között hajtjuk végre, pontosan ebben a sorrendben, mivel ez a művelet nem kommutatív, csak bizonyos esetekben. A szorzás eredménye lesz egy 4x1-es oszlopvektor lesz, mely a mi "kis" hálózatunkban lévő 4 oldal PageRank-jét tartalmazza. A `kiir()` függvénnyel pedig kiíratjuk az egyes oldalakhoz tartozó eredményeket, mely a következő képpen zajlik:

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

    int i;

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

Tehát egy `for` ciklussal bejárjuk a tömböt, és egyenként kiírjuk az értékeket.

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

Tanulságok, tapasztalatok, magyarázat...

Viggo Brun, XX. századi norvég matematikushoz kapcsolódik a Brun tétel kidolgozása, melyet 1919-ben bizonyított be. A tétel azt mondja ki, hogy **az ikerprímek reciprokösszege egy véges értékhez, úgynevezett Brun-konstanshoz konvergál**. Ikerprímeknek nevezzük azokat a prímpárokat, melyek különbsége 2. A konvergálás pedig azt jelenti, hogy az reciprokösszeg soha nem halad meg egy bizonyos értéket, csak tetőleges mértékben megközelíti. Ezt bizonyítottuk be egy R programmal, amit az előbbi linken láthatsz. Az R nyelv kifejezetten matematikai számításokhoz, statisztikák és grafikonok készítéséhez lett kifejlesztve. Ezért ideális ennek a tételnek a bizonyításához is. Magát az ehhez szükséges programot [itt](#) tudod letölteni. A feladat megoldásához a matlab kiegészítőre lesz szükséged, melyet ezzel a paranccsal tudsz telepíteni:

```
install.packages("matlab")
```

Ha megnézed a `stp.r` forráskódot, láthatod, hogy az R nyelv szintaktikája eltér a C nyelvétől, de könnyen tanulható. Mivel az R nyelv iteratív nyelv, ezért nem kell lefordítanod az egész fájlt gépi kóddá, hanem sorról sorra írod be, és folyamatosan lefordul, majd végrehajtodik. A program futtatásához elsőnek be kell tölteni a matlab függvénykönyvtárat, majd létrehozuk a `stp` függvényt.

```
library(matlab)

stp <- function(x) {

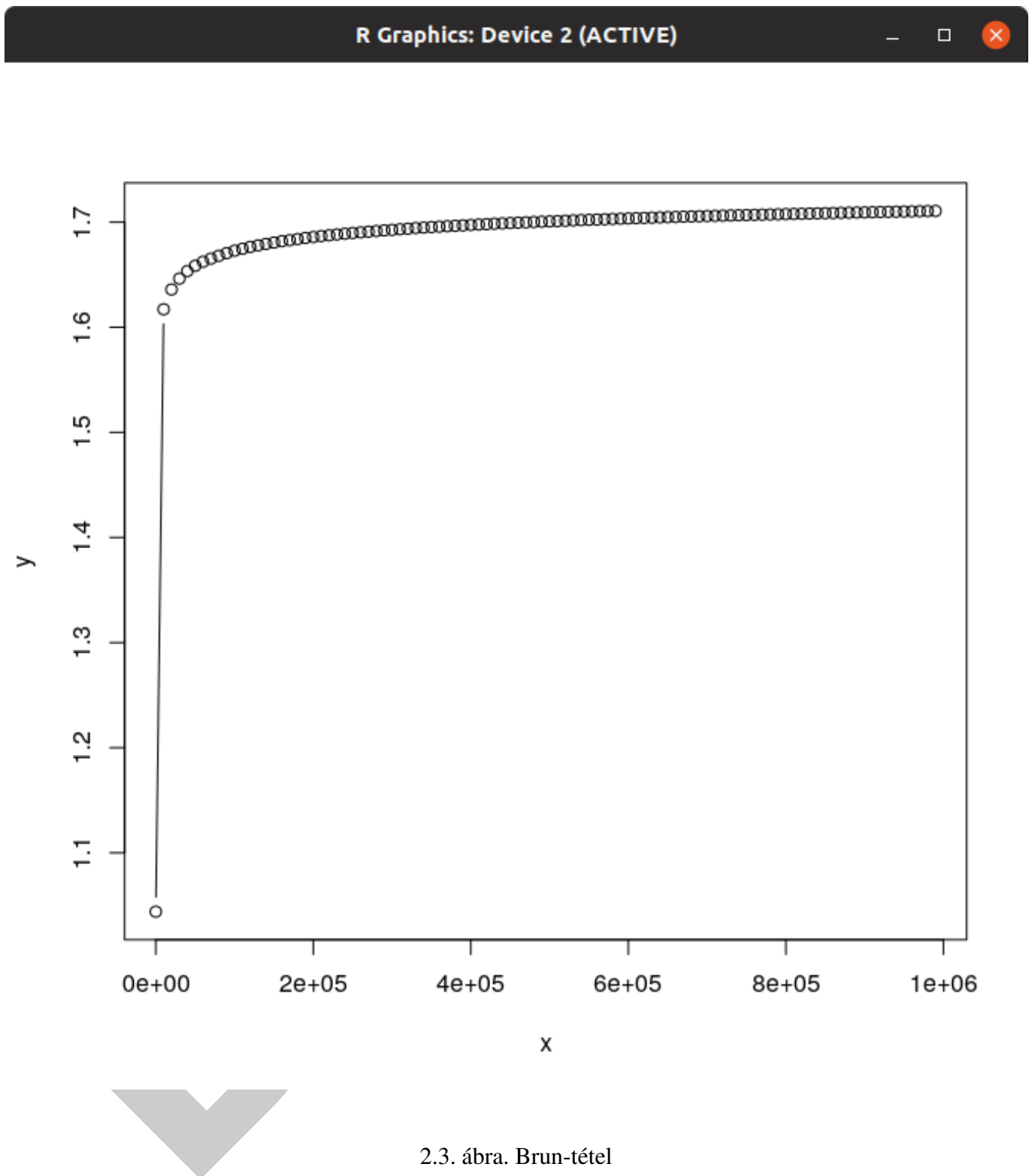
  primes = primes(x)
  diff = primes[2:length(primes)]-primes[1:length( ←
    primes)-1]
  idx = which(diff==2)
  t1primes = primes[idx]
  t2primes = primes[idx]+2
  rt1plust2 = 1/t1primes+1/t2primes
  return(sum(rt1plust2))

}
```

Ez a függvény 1 paramétert kér, és ezt adja tovább a `primes` beépített matlab függvénynek. Ez a függvény a megadott `x`-ig kiír minden prím számot egy vektorba. A `diff` változóban eltároljuk a `primes` vektorban lévő egymás melletti prímek különbségét. Ezután az `idx` vektorban pedig `diff` vektor elemeinek indexét tároljuk el, ahol amelyek értéke 2. Ezután index alapján megnézzük, hogy melyek ezek párok, ahol a különbség kettő. Majd felhasználjuk a Brun-tételt, tehát vesszük a prímeknek a reciprokát, és azt adjuk össze. Mivel egy függvényt akarunk kirajzolni, ezért meg kell adnunk egy `x` és egy `y` értéket, hogy ki tudjuk rajzolni az ábrát.

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

A `seq()` függvénnyel megadjuk, hogy `x`-tengekyen mettől-meddig haladjunk, és milyen lépésközzel. A következő sort, pedig így kell érteni: `y = stp(x)`. Tehát minden `y`-hoz hozzárendeljük az `stp(x)` értékét. A `plot()` függvény pedig kirajzolja az `(x,y)` értékeket egy grafikonon. Ha majd MATLAB-ot kell használnod, akkor jól megfogod ismerni a `plot()`, `plot3()` és a hasonló kiírató függvényeket. Tehát, ha ezt lefutattod, akkor a következő ábrát fogod látni:



2.3. ábra. Brun-tétel

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

[paradoxon_kapcsan](#)

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

Tanulságok, tapasztalatok, magyarázat...

A Monty HALL probléma lényege a következő. Van 3 ajtó, az egyik mögött van a nyeremény, a többi mögött semmi. A játékos választ egyet a 3 ajtó közül, majd a játékmester kinyit egy olyan ajtót, amit a játékos nem választott, és nincs mögötte semmi. Ezután megkérdezi a játékost, hogy szeretne-e változtatni a döntésén, vagy marad az általa elsőnek kiszemelt ajtónál. Kérdés az, hogy megéri-e váltania? Ahogy a belinkelt videóban is láhattad ez még a legjobb matematikusoknak is fejfájást okozott. Tehát a válsz, a mindenkit foglalkoztató kérdésre az, hogy igen, megéri váltani.

Ennek oka nagyon egyszerű. Az első tippednél $1/3$ az esélye annak, hogy eltaláltad a helyes ajtót. Ekkor ha váltasz, akkor buksz, ha maradsz a döntésednél, akkor viszont nyersz. De tegyük fel, hogy nem találtad el a megfelelő ajtót. Ebben az esetben te rámutatsz egy üres ajtóra, a játékmester kinyitja neked a másik üres ajtót. Tehát, ha ebben az esetben váltasz, akkor nyersz. Ennek az esély pedig $2/3$ -hoz. Hiszen $2/3$ eséllyel tippelsz rosszul az elején. Egy szemléltető ábra, a fent linkelt oldalról: <https://m.blog.hu/bh/bhaxor/image/-montyhall2.png>.

Ezt próbáljuk szimulálni egy R programban.

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

Első lépésben megadjuk, hogy mennyi legyen a kísérletek száma. Ha ez meg volt, akkor véletlenszerűséget szimuláljuk a `sample()` függvénnyel. Ahogy látjátok ennek 3 argumentuma van, az első megadja, hogy mettől-meddig generáljon random számokat. Majd megadjuk, hogy hányszor tegye meg ezt, és a `replace=T(rue)` pedig azt engedi meg, hogy lehessen ismétlődés a számok között. Lényegében itt egy tömböt adunk a `sample()`-nek át, amit átrendez. Tehát a kísérletek tömbben tároljuk, hogy az egyes esetekben hol van a nyeremény, a `jatekos` tömbben a játékos tippjeit. A `műsorvezető` változó pedig szintén egy tömbvektor, melynek jelenleg csak a méretét adjuk meg, mivel az ő döntése függ a játékosétól és a nyeremény helyétől is.

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

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

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

  }else{

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

  }

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

}
```

A for ciklusban az *i* változót futtadjuk 1-től a kísérletek számáig. Két esetet kell szétbontani itt, a játékos elatlálta a megfelelő ajtót, vagy sem. Ezt vizsgáljuk az if-ben. Tehát, ha az a kísérlet tömb első elem és a játékos tömb első eleme megegyezik, akkor a játékvezető csak azt az egy ajtót nem választhatja. A miből tömbben, pont ezt tároljuk el. Ehhez a `setdiff()` függvényt használjuk, mely a megadott tömbből kiveszi a kísérlet tömb első elemével megegyező értéket. Ugyan ez történik az else ágon is, csak ott mind a kísérlet, mind a játékos tömb első elemét ki kell vonni a megadott tömbből. Ez alapján pedig fel tudjuk tölteni a musorvezeto vektort a megfelelő értékekkel, ehhez megint a `sample()` függvényt használjuk.

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

A `nemvaltoztatesnyer` vektorba a `which()` függvénnyel betöltjük azon elemek indexét melyek a a kísérlet és a játékos tömbben azonos pozícióban vannak, és megegyeznek. Ezután létrehozuk a `valtoztat` vektort, melynek mérete megegyezik a kísérletek számával.

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

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

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

A for cikluson belül feltöltjük a `valtoztat` vektort minden esetben azzal az ajtóval, amire a játékos mutat, ha újra választ. Ezután a `valtoztatesnyer` vektorba betöltjük annak az elemeknek az indexét, melyeknek az értéke megegyezik mind a kísérlet, mind a `valtoztat` tömbben. Ha ezzel megvagyunk, már csak ki kell print-elni a `valtoztatesnyer` és a `nemvaltoztatesnyer` vektorok hosszát, és ezzel megtudjuk, hogy melyik lesz a nagyobb. Természetesen az előbbi, ahogy már azt említettem.

3. fejezet

Helló, Chomsky!

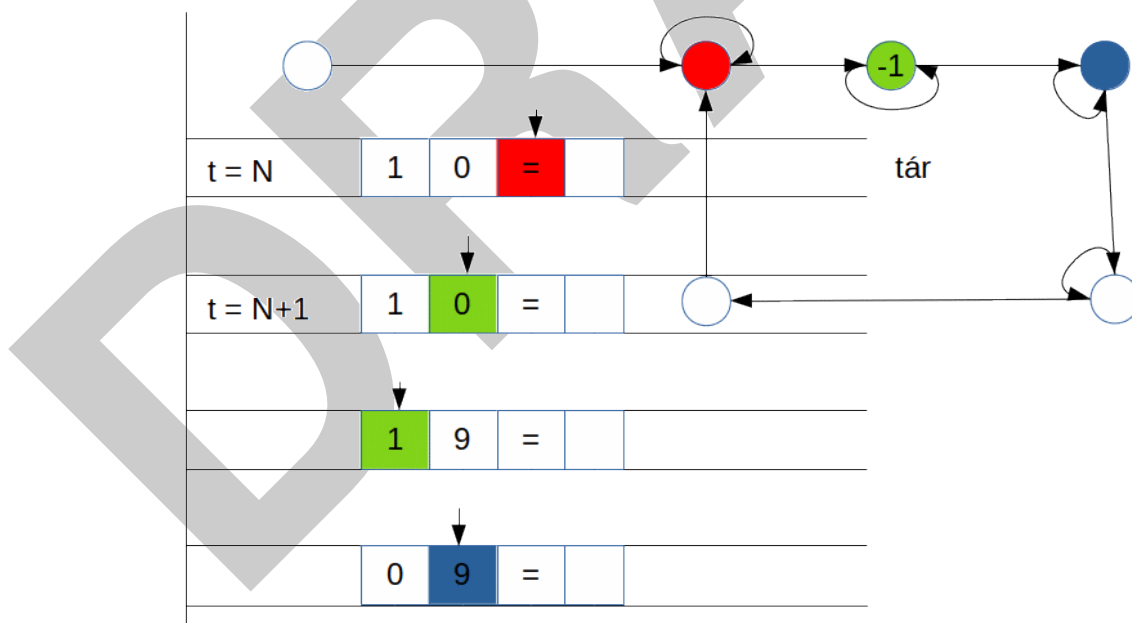
3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfiával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...



3.1. ábra. Decimálisból unárisba

A unáris számrendszer a létező legegyszerűbb az összes számrendszer közül. Lényegében megfelel annak, amikor az ujjaink segítségével számolunk, tehát a számokat vonalakkal reprezentáljuk. A szám ábrázolása pont annyi vonalból áll, amennyi a szám. A könnyebb olvashatóság érdekében minden ötödik után rakhatunk helyközt, vagy az ötödik vonalat lényegében rakhatjuk keresztbe az öt megelőző 4-re.

Az ábrán látható Turing gép az egyes számrendszerbe való átváltást végzi, de itt a 1 helyett 1-eseket írunk. A gép működése abból áll, hogy beolvassa a szalag celláiban tárolt számokat, ha a talál egyenlőség jelet, akkor az előtte lévő számból kivon 1-et. Ezt egészen addig teszi, ameddig az előtte lévő szám le nem nullázódik. Jelen esetben ez pont 0, de mivel előtte áll egy 1-es, ezért itt a 0-ból 9 lesz, és az 1-esből 0. Miközben folyamatosan vonja ki a egyeseket a számból, a kivont egyeseket kiírja a tárolóra. Így a végén egy 1-esekből álló sorozatot kapunk, melyek száma megegyezik a kiindulási szám értékével.

A forrásként megadott program lényegében egy átváltó, mely függőleges vonalakat ír ki a bemenettől függően. Mivel ez egy C++ program, ezért ennek a fordításához a g++-t érdemes használni, a szintaxisa teljesen megegyezik a gcc-nél megszokottakkal. Ha lefuttatod, akkor a következőt fogod látni:

```
$ g++ unaris.cpp -o unaris
$ ./unaris
Adj meg egy számot decimálisan!
10
Unárisan:
||||| |||||
```

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Generatív nyelvtan jelsorozatokat átalakítási szabályait tartalmazza. A nyelvnek megfelelő szavak létrehozásához szükség van egy kezdő értékre, és ezután már csak a szabályokat kell alkalmazni az átalakítások elvégzéséhez. Tehát a kezdő szimbólumot más szimbólumokkal helyettesítjük a nyelvtani szabályok figyelembevételével.

Noam Chomsky, XX. századi amerikai nyelvész volt az, aki elsőnek javasolta a generatív nyelvtanok formalizálását. Az 1950-es években publikálta munkásságát, mely alapján egy nyelvtannak a következő elemekből kell állnia:

- i. nem-terminális szimbólumokból
- ii. terminális szimbólumokból
- iii. produkciós szabályokból
- iv. előállítási szabályokból
- v. kezdő szimbólumból

Chomsky nevéhez fűződik még a generatív nyelvtanok csoportosítása. Négy osztályba sorolta őket ezek a következők:

- 0. típus (rekurzíve felsorolható nyelvtanok)
- 1. típus (környezetfüggő nyelvtanok)
- 2. típus (környezetfüggetlen nyelvtanok)
- 3. típus (reguláris nyelvtanok)

Ebben a feladatban az 1. típussal foglalkozunk. Az környezetfüggő nyelvtanok szabályait kétféle képpen lehet felírni. Itt a képzési szabály mind a két oldalán szerepelhetnek terminális szimbólumok, ellentétben a környezetfüggetlen nyelvtanokkal ahol a produkciós szabályok bal oldalán csak nemterminális szimbólum állhat.

Most hogy tudjuk, miből épül fel egy generatív nyelvtan lássuk a feladat megoldását.

S, X, Y „változók” – nemterminálisok
 a, b, c „konstansok” – terminálisok
 $S \rightarrow abc, S \rightarrow aXbc, Xb \rightarrow bX, Xc \rightarrow Ybcc, bY \rightarrow Yb, aY \rightarrow aa$ – képzési szabályok

Jelen esetben a kezdő szimbólumunk az S lesz.

A képzési szabályokat alkalmazva a következőket kapjuk:

```
S (S → aXbc)
aXbc (Xb → bX)
abXc (Xc → Ybcc)
abYbcc (bY → Yb)
aYbbcc (aY → aaX)
aaXbbcc (Xb → bX)
aabXbcc (Xb → bX)
aabbXcc (Xc → Ybcc)
aabbYbcc (bY → Yb)
aabbYbcc (bY → Yb)
aaYbbbcc (aY → aa)
aaabbbcc
```

Tehát meg kell határozni változókat, terminálisokat és szabályokat. Amint látod a képzési szabályt formailag a nyíl operátorral jelöljük, tehát miből mi lesz. Ugyanennek a nyelvnek egy rövidebb reprezentációja a következő:

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

Fentebb említettem, hogy mi is teszi környezetfüggővé a nyelvtant, erre kiváló példa több is a képzési szabályokból. Például $bY \rightarrow Yb$, ahol tisztán látszik, hogy a nyíl bal oldalán is van terminális szimbólum, ez egy környezetfüggetlen nyelvben nem lenne lehetséges.

Egy másik nyelvtan, amely a feladatban szereplő nyelvet generálja a következőképpen néz ki:

A, B, C „változók”
 a, b, c „konstansok”
 $A \rightarrow aAB$, $A \rightarrow aC$, $CB \rightarrow bCc$, $cB \rightarrow Bc$, $C \rightarrow bc$
 A-ból indulunk ki, ez lesz a kezdőszimbólum.
 A szabályokat követve:

```

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

```

Amint láthatod nem sokat változtattunk, elég volt csak a változókat átírni, és a szabályokat, ezzel meg is kaptunk egy másik generatív nyelvtant, mely szintén az $a^n b^n c^n$ nyelvet generálja.

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: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

A Backus-Naur-forma(vagyis BNF) segítségével környezetfüggetlen grammatikákat tudunk leírni. Széles körben alkalmazzák a programozási nyelvek szintaxisának leírására. Ahogy a neve is mutatja, a jelölés rendszer John Backus nevéhez fűződik, aki 1959-ben a párizsi WCC-n mutatta be. Majd Peter Naur egyszerűsített a jelöléseken, csökkentette a felhasznált karakterek számát. Tevékenységéért kiérdemelte, hogy az ő neve is szerepeljen a forma nevében.

Ahhoz, hogy lássuk hogyan is néz ki egy BNF leírás, vegyünk egy egyszerű példát:

```

<postai_cím> ::= <név_rész> " ," <irányítószám_rész> " " <cím_rész>
<név_rész> ::= <személyi_rész> <keresztnev> | <név_rész> <keresztnev>
<személyi_rész> ::= <titulus> " ." " <vezetéknév> | <vezetéknév> " "
<cím_rész> ::= <kerület> <elnevezés> <közterület_típus> <szám> <EOL>
<közterület_típus> ::= "utca" | "tér" | "körút" | "lépcső" | "u." | "krt."
<irányítószám_rész> ::= <ország_kód> "-" <irányítószám_belső> | <←
    irányítószám_belső>
<irányítószám_belső> ::= <irányítószám> " " <városnév> " ,"

```

Forrás: [wiki](#).

Nézzük például az első sort. Itt a postai címet határozzuk meg, ami áll egy név részből, irányítószámból és címből. Közéjük vesszőt rakunk, vagy csak szóközt. A második rész már egy kicsit érdekesebb, mert ebből kiderül, hogy a BNF jelölések rekurzívak is lehetnek. A `név_részben` hivatkozunk a `személyi_részre`, és önmagára is. A `|` jel vagy-ot fejez ki, az értékadás pedig a `::=` operátorral történik.

Mostmár van fogalmunk arról, mi is az a BNF, hát akkor próbáljuk meg átültetni ezt a tudást a C utasítás leírására. Mielőtt tovább haladnál a könyv olvasásával érdemes elolvasni a feladatban említett részletet a K&R könyvből.

```
<utasítás> ::= <kifejezés_utasítás>|<összetett_utasítás>|
               <feltételes_utasítás>|<while_utasítás>|
               <do_utasítás>|<for_utasítás>|<switch_utasítás>|
               <break_utasítás>|<continue_utasítás>|<return_utasítás>|
               <goto_utasítás>|<címkézett_utasítás>|<>nulla_utasítás>

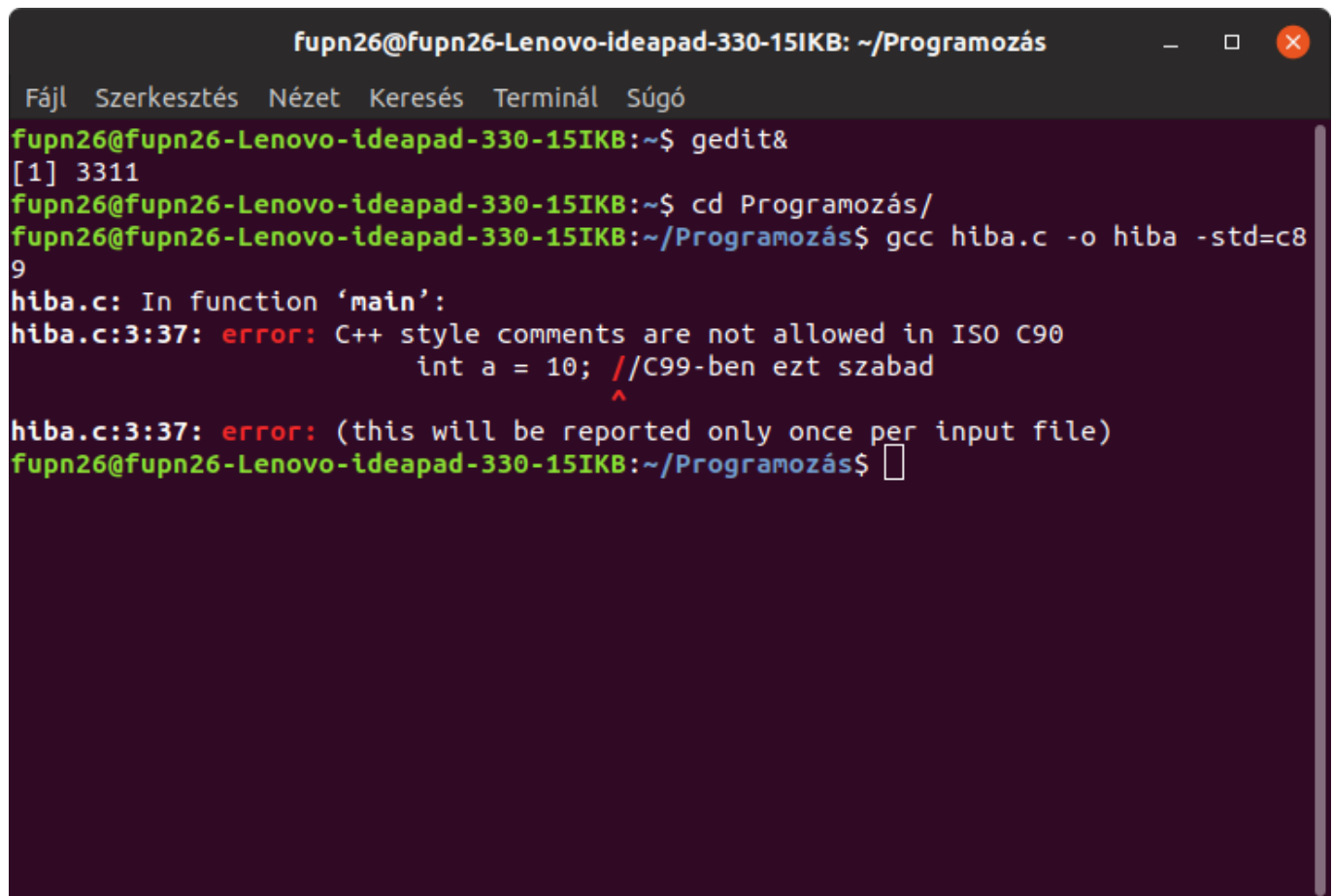
<kifejezés_utasítás> ::= <kifejezés>;
<összetett_utasítás> ::= {<deklarációlistaopc> <utasításlistaopc>}
<feltételes_utasítás> ::= if (<kifejezés>) <utasítás>|<feltételes_utasítás> ←
    else <utasítás>
<while_utasítás> ::= while (<kifejezés>) <utasítás>
<do_utasítás> ::= do <utasítás> while (<kifejezés>)
<for_utasítás> ::= for (<1._kifejopc>;<2._kifejopc>;<3._kifejopc>) < ←
    utasítás>
<switch_utasítás> ::= switch (<kifejezés>) <utasítás>| <switch_utasítás> ←
    case
        <állandó_kifejezés>:|<switch_utasítás> default:
<break_utasítás> ::= break;
<continue_utasítás> ::= continue;
<return_utasítás> ::= return; | <return_utasítás> <kifejezés>
<goto_utasítás> ::= goto <azonosító>;
<címkézett_utasítás> ::= <azonosító>:
<>nulla_utasítás> ::= ";"
```

A C programnyelvet az 1970-es években alkották meg, mely azóta 3 főbb verziót ért meg. Az első volt a C89, ez a klasszikus C nyelv, amit a K&R könyvből ismerhetünk meg. Ezután jött a C99, mely újdonságai között szerepelt többek között az inline függvények támogatása, új adattípusok jelentek meg, mint a long long int, vagy a complex. Egészen eddig nem volt támogatott az egysoros komment `//` jelölése, és a dinamikus tömb sem. Egy újabb verziója pedig 2011-ben jelent meg C11 néven, mind a mai napig ez a legfrissebb sztenderd. Ez főleg a C99-es sztenderd továbbfejlesztésének tekinthető.

Fontos megjegyezni, hogy a gcc lap esetben a C89-es szabványt használja, tehát, ha ezen módosítanál, akkor a `-std=c99` kapcsolót kell használnod.

```
int main()
{
    int a = 10; //C99-ben ezt szabad
    return 0;
}
```

Ránézésre semmi probléma nem lehetne vele, de a gcc-t a `-std=c89` kapcsolóval használva hibát fog kiírni.

A screenshot of a terminal window titled "fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás". The window has a menu bar with "Fájl", "Szerkesztés", "Nézet", "Keresés", "Terminál", and "Súgó". The terminal shows the following commands and output:

```
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~$ gedit&
[1] 3311
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~$ cd Programozás/
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ gcc hiba.c -o hiba -std=c89
hiba.c: In function 'main':
hiba.c:3:37: error: C++ style comments are not allowed in ISO C90
      int a = 10; //C99-ben ezt szabad
                  ^
hiba.c:3:37: error: (this will be reported only once per input file)
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$
```

3.2. ábra. 1. Splint kép

Igen, jól látod, az egysoros komment jelölésére használt `//` a C99 előtt, csak a C++ támogatta. Ennek a támogatásának bevezetése a C nyelvbe, a C99-es sztenderd egy nagy újdonsága volt. Ha C89-es szabvány szerint szeretnél kommentelni, akkor az előző példa, így módosul:

```
int main()
{
    int a = 10 /*ez már lefordul -std=c89-el*/
    return 0;
}
```

Ennek a kommentelésnek az előnye, hogy lehetőséget biztosít a többsoros komment létrehozására.

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:[itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ehhez a feladathoz a lex programot kell használni, melyel egy lexikális lemezőt lehet készíteni. Szövegfájlokból olvassa be a lexikális szabályokat, és egy C forráskódot készít, melyet a gcc-vel tudunk fordítani. A lex forráskód 3 részből áll, az első a definíciós rész, amely lényegében bármilyen C forrást tartalmazhat, itt lehet include-álni a header fájlokat. A második rész a szabályoknak van fenntartva. Ez 2 részből áll, reguláris kifejezésekből, és az azokhoz tartozó C utasításokból. Tehát, ha a program futása során a bemenetként kapott string illeszkedik valamelyik reguláris kifejezésre, akkor végrehajtja a hozzá tartozó utasítást. A harmadik rész pedig egy C-kód, amely lényegében a lexikális elemzőt hívja meg. Ez a rész, és az első, teljes mértékben átmásolódik a lex által generált C forrásba. A részeket %%-jellel különítjük el.

Most hogy már tudod mi is az a lexer, itt az idő végig futni a belinkelt forráson. Az első rész a következő:

```
%{
#include <stdio.h>
int realnumbers = 0;
%}
```

Ahogy említettem, itt importáljuk a header fájlokat, és a szükséges változókat is itt deklaráljuk. Ezután következik a második rész:

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

Itt a bal oldalon vannak a reguláris kifejezések, jobb oldalon pedig az, amit a C program végrehajt. Ez a regex azokra a bemenetekre illeszkedik, amelyek számmal kezdődnek, és a * jelöli, hogy többször is előfordulhat. Majd a zárójelben lévő csoportból valamelyik tag 0-szor, vagy 1-szer fordul elő. Ezt jelöli a ?-jel. Az atof() függvény pedig az argumentumként kapott stringet double-lé alakítja. Vessünk egy pillantást a harmadik részre:

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

Ez tartalmazza a yylex() lexikális elemző függvény hívását, és itt írjuk ki az eredményt is.

Most, hogy láttad, hogyan épül fel a lexer, akkor már csak ki kéne próbálni.

```
$ lex -o lexikalis.c lexikalis.l
$ gcc lexikalis.c -o lexikalis
$ ./lexikalis
12 34 54 12
The number of real numbers is 4
```

A bevittelt a Ctrl+D paranccsal lehet megállítani. Szöveges fájlt beleírányítani pedig a

```
./lexikalis <fajl_nev
```

paranccsal lehet.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó: [itt](#)

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Elsőnek nézzük meg, hogy mi is az a leet nyelv. Ennek a nyelvnek az a lényege, hogy a szavakban lévő bizonyos betűket, számokkal, vagy egyéb más karakterekkel helyettesítjük. Az egyes betűket közmegállapodás szerinti karakterekre cserélhetjük. Erről a teljes listát [itt](#) találsz. Természetesen a linkelt programunk is ebből indul ki, de nem minden opciót vettünk bele.

Az előző feladatban már ismertettem veled a lexer felépítését, és ugyan ezt használjuk itt is. Természetesen az első részben most is importálva vannak különböző header fájlok, de ami érdekesebb az utána jön.

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

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

    {'a', {"4", "4", "@", "/-\\"}},
    {'b', {"b", "8", "|3", "|"}},
    {'c', {"c", "(", "<", "{"}},
    ...
};
```

A `#define` segítségével lényegében egy makróhelyettesítést hajtunk végre, ezt úgy kell elképzelni, hogy ha a programban valahol hivatkozunk az `L337SIZE`-ra akkor a mellette található értékkel fogja helyettesíteni. Ezután létrehozuk a `cipher` struktúrát, mely egy `char c`-ből, és egy 4 elemű karakterekből álló tömbre mutató `char *` típusú mutatóból áll. Ez alapján a struktúra alapján létrehozuk az `l337d1c7 []` tömböt. Ez a tömb tartalmazza az egyes betűket, és a hozzájuk tartozó lehetséges helyettesítő karaktereket. Tehát a tömb minden eleme áll egy `char c`-ből, és egy `char *leet[4]`-ből.

A második részben ezúttal nem használunk különféle regex-eket, tehát minden karakter esetén végrehajtjuk a C utasítást.

```
. {

    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)); //random szám ←  
            generálás  
  
        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);  
  
}
```

Tehát a `.t` használjuk, ami minden karakterre illeszkedik. A `for` ciklus fejében lekérjük a `L337SIZE` konstans értékét, ami a `l337d1c7[]` tömbnek és a cipher struktúra méretének a hányadosa. Mivel a `l337d1c7[]` egy struktúrált tömb, ezért a `l337d1c7[i].c` tudunk hivatkozni a tömb *i*-dik elemének a `char c` részére. A `tolower()` függvény segítségével az esetleges nagybetűket kicsivé változtatjuk. Majd ezután képzünk egy random számot 1-100 között. Ha 91-nél kisebb számot "dob" a gép, akkor a `char *leet[4]` tömb első elemét printeljük, és így tovább, ahogy a forrásban látod. A `int found` változót azért vezettük be, hogy jelezzük, benne van-e tömbünkbe a beolvasott karakter. Ha nincs, akkor visszadjuk az eredeti karaktert, módosítás nélkül.

Ezután megint a C forrás jön, melyben elindítjuk a lexelést.

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

Itt található egy érdekesség, az `srand()` függvény adja meg a kiindulási értéket a `rand()` függvény számára. Jelen esetben az aktuális időt és a szülő folyamat PID-jának összegét adja át.

3.6. A források olvasása

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

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

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



Bugok

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

i.

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

ii.

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

iii.

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

iv.

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

v.

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

vi.

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

vii.

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

viii.

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

Megoldás forrása:

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat...

Programozó pályafutásunk során rengetegszer kell mások kódjában tájékozódni, ez a feladat ebben nyújt segítséget, hogyan értelmezzük helyesen a kódokat. Lássuk is az elsőt. A bevezetőben már láthattad, hogy melyik manuál oldalakat kell átnézni ehhez a kódcsipethez, szóval ezzel most nem rabolnám az időt.

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


Ez azt jelenti, hogyha eddig nem volt figyelmen kívül hagyva a SIGINT jel, akkor a jelkezelő függvény kezelje. Ellenkező esetben hagyjuk figyelmen kívül.

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

Ez egy for ciklus, amelynél az első iterációban az *i* nulla, majd megnézzük, hogy kisebb-e, mint 5, és minden iterációban növeljük 1-el.

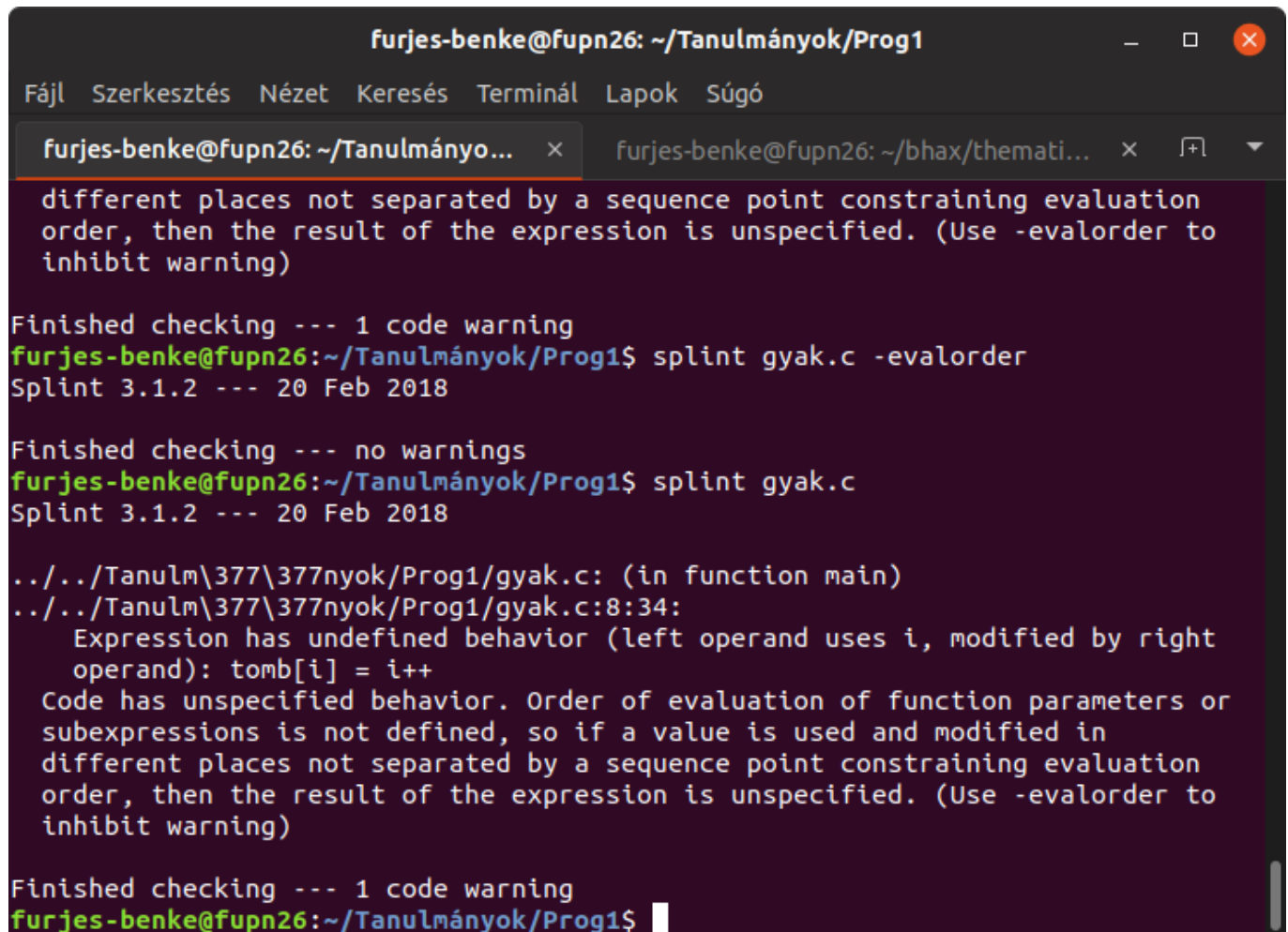
```
for(i=0; i<5; i++)
```

Ez ebben az esetben megegyezik az előzővel, ugyan úgy 0,1,2,3 majd 4 lesz az *i* értéke. Viszont ez nem mindig van így.

```
int a = 5;
int b = a++; //itt a b értéke 5 lesz, majd növeljük az a értékét 1-el
int c = ++a; //c értéke már nem 6 lesz, hanem 7, mivel itt előbb ↵
           növelünk
```

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

Ez már viszont egy bugos program, mivel egyszerre inkrementáljuk az *i*-t, és hivatkozunk a *tomb* *i*. elemére. Az a baj, hogy nem ismerjük a végrehajtás sorrendjét, emiatt nem kiszámítható az eredmény.



```
furjes-benke@fupn26: ~/Tanulmányok/Prog1
Fájl Szerkesztés Nézet Keresés Terminál Lapok Súgó
furjes-benke@fupn26: ~/Tanulmányo... x furjes-benke@fupn26: ~/bhax/themati... x [+]
```

different places not separated by a sequence point constraining evaluation order, then the result of the expression is unspecified. (Use -evalorder to inhibit warning)

Finished checking --- 1 code warning

```
furjes-benke@fupn26:~/Tanulmányok/Prog1$ splint gyak.c -evalorder
Splint 3.1.2 --- 20 Feb 2018
```

Finished checking --- no warnings

```
furjes-benke@fupn26:~/Tanulmányok/Prog1$ splint gyak.c
Splint 3.1.2 --- 20 Feb 2018
```

../Tanulm\377\377nyok/Prog1/gyak.c: (in function main)
../Tanulm\377\377nyok/Prog1/gyak.c:8:34:
Expression has undefined behavior (left operand uses i, modified by right operand): tomb[i] = i++
Code has unspecified behavior. Order of evaluation of function parameters or subexpressions is not defined, so if a value is used and modified in different places not separated by a sequence point constraining evaluation order, then the result of the expression is unspecified. (Use -evalorder to inhibit warning)

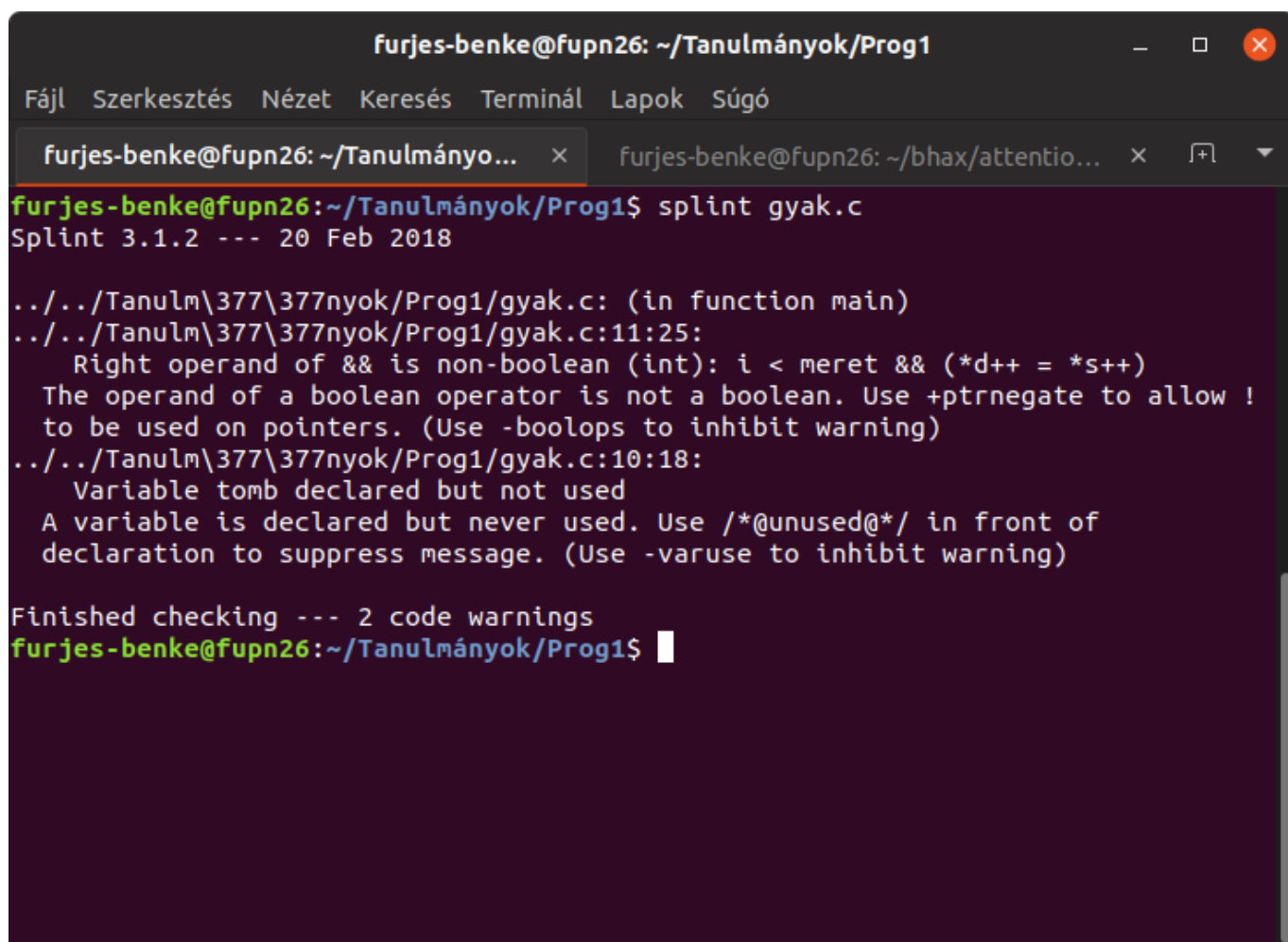
Finished checking --- 1 code warning

```
furjes-benke@fupn26:~/Tanulmányok/Prog1$
```

3.3. ábra. 1. Splint kép

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

Ez a kódcsipet szintén bugos, a probléma az, hogy az értékadó operátort használjuk, az összehasonlító operátor helyett, ennek következtében a && operátor jobb oldalán nem egy logikai operandus áll.



```
furjes-benke@fupn26: ~/Tanulmányok/Prog1
Fájl Szerkesztés Nézet Keresés Terminál Lapok Súgó
furjes-benke@fupn26: ~/Tanulmányo... x furjes-benke@fupn26: ~/bhax/attentio... x [+]
```

```
furjes-benke@fupn26:~/Tanulmányok/Prog1$ splint gyak.c
Splint 3.1.2 --- 20 Feb 2018

../../Tanulm\377\377nyok/Prog1/gyak.c: (in function main)
../../Tanulm\377\377nyok/Prog1/gyak.c:11:25:
    Right operand of && is non-boolean (int): i < meret && (*d++ = *s++)
    The operand of a boolean operator is not a boolean. Use +ptrnegate to allow !
    to be used on pointers. (Use -boolops to inhibit warning)
../../Tanulm\377\377nyok/Prog1/gyak.c:10:18:
    Variable tomb declared but not used
    A variable is declared but never used. Use /*@unused@*/ in front of
    declaration to suppress message. (Use -varuse to inhibit warning)

Finished checking --- 2 code warnings
furjes-benke@fupn26:~/Tanulmányok/Prog1$
```

3.4. ábra. 2. Splint kép

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

Ez is hibás kód, mivel az `f` függvény két `int`-et kap, de azok kiértékelésének sorrendje nincs meghatározva.

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

Ennél a kódcsipetnél nincs probléma, kiírjuk az `a` értékét és az `a` függvény által módosított értékét.

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

Ennek a kódnak szintén a kiértékelés sorrendjével van baja, mivel az `f()` függvény módosítja az `a` értékét, emiatt nem tudhatjuk, hogy az önmagában kiprintelt `a` az eredeti értékét, vagy a módosított értékét fogja kiírni.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

```

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

$(\forall x \exists y ((x < y) \wedge (y \text{ \textit{prím}})) \wedge (S y \text{ \textit{prím}})) \leftrightarrow$
  )$

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

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

```

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 aritmetika nyelve a logikai nyelvek közül az elsőrendűek közé tartozik. A legalapabb logikai nyelv a nulladrendű logika, ez csak ítéletváltozókat tartalmaz, és 3 műveletet: a konjunkciót, diszjunkciót és implikációt. Erre épül rá az elsőrendű logika, kiegészülve függvényszimbólumokkal, változókkal, kvantorokkal. Ha részletesebb tájékozódni, akkor a forrásban több könyv is meg van nevezve, ami a feladathoz szükséges ismereteket tartalmazza.

Akkor térjünk rá a feladatra. Elsőnek elmondom, hogy melyik kifejezés mit jelent. A `forall` fogja jelölni az univerzális kvantort, és az `exist` pedig az egzisztenciális kvantort. A `wedge` alatt az implikációt értjük, és a `supset` pedig a konjunkciót. Ehhez a feladathoz érdemes még ismerni az Ar nyelv rákövetkező függvényét, amit `S`-el jelölünk. Most, hogy ezeket tudjuk, már csak le kell fordítani a emberi nyelvre a fentebb látható formulákat.

1. Minden x -hez létezik olyan y , amelynél ha x kisebb, akkor y prím.
2. Minden x -hez létezik olyan y , amelynél ha x kisebb, akkor y prím, és ha y prím, akkor annak második rákövetkezője is prím.
3. Létezik olyan y , amelyhez minden x esetén az x prím, és x kisebb, mint y .
4. Létezik olyan y , amelyhez minden x esetén az x nagyobb, és 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?

- `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: [itt](#) ,

Tanulságok, tapasztalatok, magyarázat...

```
int a;
```

egy egész vezet be a programba.

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

egy egészre mutató mutatót vezet be. A mutató, vagy elterjedtebb nevén pointer lényegében egy olyan változó, ami egy változó, egy tömbre esetleg egy függvényre memóriacímére mutat. Érdekes megfigyelni, hogy a `&a` vagyunk képesek átadni az `a` változó memóriacímét.

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

egy egésznek a referenciáját vezeti be. Láthatod, hogy a mutatót a * operátor segítségével deklarálunk, míg a referenciát a & jellel. A kettő közti különbség az, a referencia, lényegében egy alias egy másik változóhoz. Nem foglal külön helyet a memóriában, mint a pointer, hanem osztozik a területen a referált változóval.

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

egy egészekből álló, 5 elemű tömböt deklarál.

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

az egészek tömbjének referenciáját vezeti be a programba. Fontos látni, hogy ez nem az első elemnek a referenciája, hanem az összes elemnek.

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

egészre mutató mutatók tömbje.

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

egészre mutató mutatót visszaadó függvény.

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

egészre mutató mutatót visszaadó függvényre mutató mutató.

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

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

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

függvényt mutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

4. fejezet

Helló, Caesar!

4.1. `int **` háromszögmátrix

Megoldás videó:

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban a pointerek használatával fogunk egy kicsit jobban megismerkedni. De mielőtt rátérnénk a forrásra, előtte tisztázzuk mi is az a háromszögmátrix. Ez 2 tulajdonsággal rendelkezik, az első, hogy négyzetes, tehát sorai és oszlopai száma megegyezik, a másik pedig az, hogy a főátlója alatt(a mi programunkban felett) csupa nulla szerepel. A program ezt a mátrixot fogja elkészíteni:

Főátló

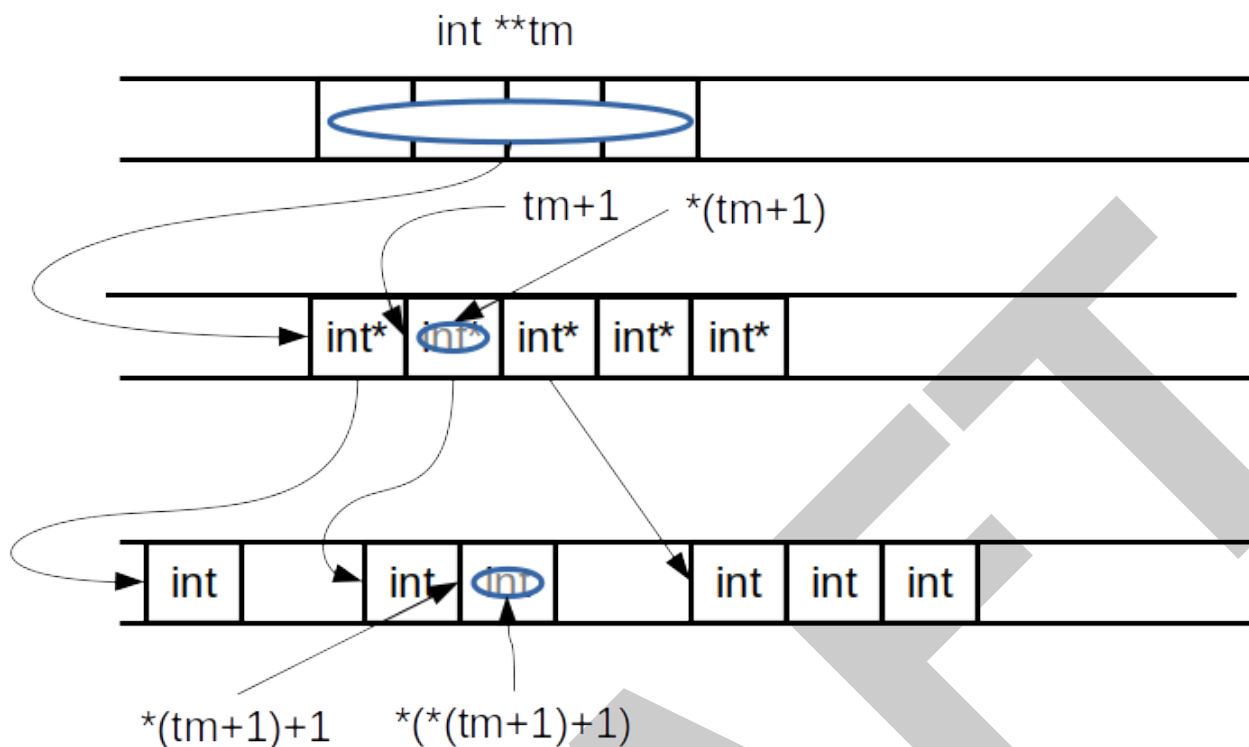
Oszlopok

Sorok

	0	0	0	0	0
1	2	0	0	0	
3	4	5	0	0	
6	7	8	9	0	
10	11	12	13	14	

4.1. ábra. Háromszögmátrix

Most, hogy tudjuk, mit várunk el a programtól, nézzük meg, hogyan lehet ezt megvalósítani. Ahogy a feladat kiírásában is láthatjátok, ezt egy egésze mutató mutatónak a mutatójával fogjuk létrehozni. Gondolom ez most egy kicsit bonyolultnak hangzik, de itt egy ábra a program egyszerű megértéséhez:



4.2. ábra. Pointerek a memóriában

A könnyebb megértés érdekében vegyük sorra, hogy mi is van az ábrán. Legfelül látjuk az `int **tm` mutatót, ez a deklaráció. Azért áll 4 négyzetből, mert az `int` típusú változók 4 bájtosak. És ez a mutató egy `int*`-ra mutat, vagyis annak a memóriacímére, a memóriacím kerül be a kék oválisba. Ezután láthatod a `tm+1`-et, ez azt jelenti, hogy a `tm` mutató "eggyel" arrébb mutat, vagyis 4 bájttal arrébb. A `* (tm+1)`-el pedig a benne lévő értéket káojuk meg. Ezzel ekvivalens jelölés, mely talán jobban érthető, ha úgy képzeljük el ezt, mintha egy tömb lenne. Tehát a `tm+1` értelmezhető így: `&tm[1]`, vagyis a `tm` tömb második elemének memóriacímeként. Ebből következik, hogy a `* (tm+1)` pedig a `tm[1]`, mely nem más, mint a `tm` tömb második elemének az értéke. Mivel még mindig a pointereknél tartunk, ezért ez az érték szintén egy memóriacím lesz, mégpedig egy `int` típusú változójé. És ezzel megérkeztünk a változók szintjére, ahol már nincsenek pointerek. Ha azt akarod tudni, hogy mi a `* (tm+1)` által mutatott `int` értéke, akkor egyszerű a dolgunk, csak elé rakunk még egy csillagot, tehát `* (* (tm+1))`. Értelem szerűen, itt úgy tudsz a következő elemre mutatni, hogy `+1`-et hozzáadsz, vagyis `* (tm+1)+1`, ennek az értékét pedig `* (* (tm+1)+1)`. Ebben az esetben is használhatod a tömbös analógiát. Mivel a mátrix lényegében egy két dimenziós tömb, ezért ábrázolható így is `&tm[1][1]` a memóriacímet, és `tm[1][1]`-el az értékét. Összességében rajtad áll, hogy melyiket szeretnéd használni, kezdetben talán a tömbös megoldás érthetőbb, de érdemes hozzászokni a másikkhoz, mert az az elterjedtebb.

Most elemezzük a programot sorról sorra.

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

int
main ()
```

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

Az elején, ahogy már megszoktad includálj a megfelelő header fájlokat, az `stdio.h` ismerős lehet, ez kell a `printf` használatához, és most megismerkedünk az `stdlib.h`-val, mely a `malloc` utasítás használatához lesz szükséges. Az `nr` tartalmazza az oszlopok számát, ls itt deklaráljuk a `**tm` pointert is.

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

Ebben a részben a `printf` kiírja a `tm` memóriacímét, majd az `if` feltételén belül, a `tm`-et ráállítjuk a `malloc` által foglalt `5*8` bájtnyi táterületre. A `malloc` egy pointert ad vissza, ami a lefoglalt tárra mutat, `void *` típusút, tehát bármely típust vissza tud adni típuskényszerítéssel. Jelen esetben ez `**int` visszaadására kényszerítjük. Majd az `if` feltételeként megvizsgáljuk, hogy tudott-e lefoglalni területet a `malloc`, ha nem, akkor visszatérünk hibával. Ha a tárfoglalás sikerült, akkor kiírjuk a lefoglalt tár címét. Ha az ábrát visszanézed, most tartunk a második sorban.

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

A `for` cikluson belül "létrehozzuk" az ábra szerinti második sor elemeit, melyek `int *` típusúak. A ciklusban 0-tól megyünk 4-ig, egyesével lépkedve. A `tm` mutatót itt úgy kezeljük, mint egy tömböt, és a `tm` által mutatott mutatókhoz foglalunk tárt területet, és ráállítjuk őket. Érdemes megfigyelni, hogy mindegyikhez `i+1`-szer 4 bájtot foglalunk le, és a `malloc int *`-ot ad vissza. Itt is megvizsgáljuk, hogy sikerült-e a foglalás, hanem hibával térünk vissza. Most kész a második sor, és mindegyik `int *` egy harmadik sorban lévő `int`-ek csopoortjának első elemére mutat, mindegyik más csoportra.

```
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)
```

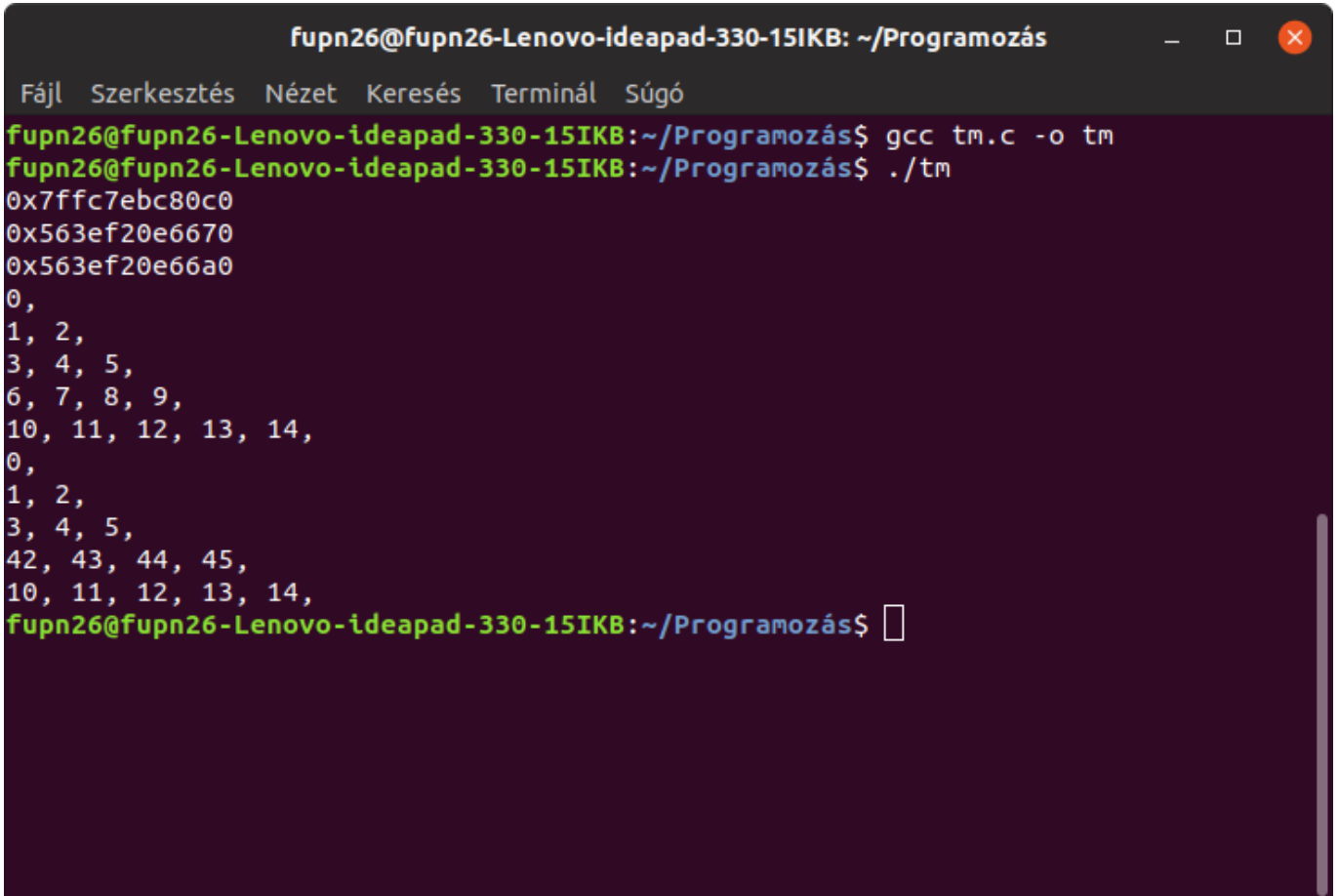
```
        printf ("%d, ", tm[i][j]);  
        printf ("\n");  
    }
```

Kiíratjuk a harmadik sor első int csoportjának első elemének a memóriacímét. Majd a for cikluson belül értéket adunk a harmadik sori int-eknek. Az *i*-vel megyünk a 4-ig, vagyis *nr-1*-ig, *j*-vel pedig mindig 0-tól *i*-ig. Az *i* jelöli a sorok számát, a *j* pedig az oszlopokét. Mátrix minden eleméhez a sor-szám*(szorszám+1)/2+oszlopszám, és ezzel megkapjuk a feladat legelején felvázolt mátrixot, amit a következő `for`-ban már csak elemenként kiíratunk.

Hogy egy kicsit szokja a szemed a többféle jelölést, nézzük meg az előbb megadott mátrix néhány elemének módosítását.

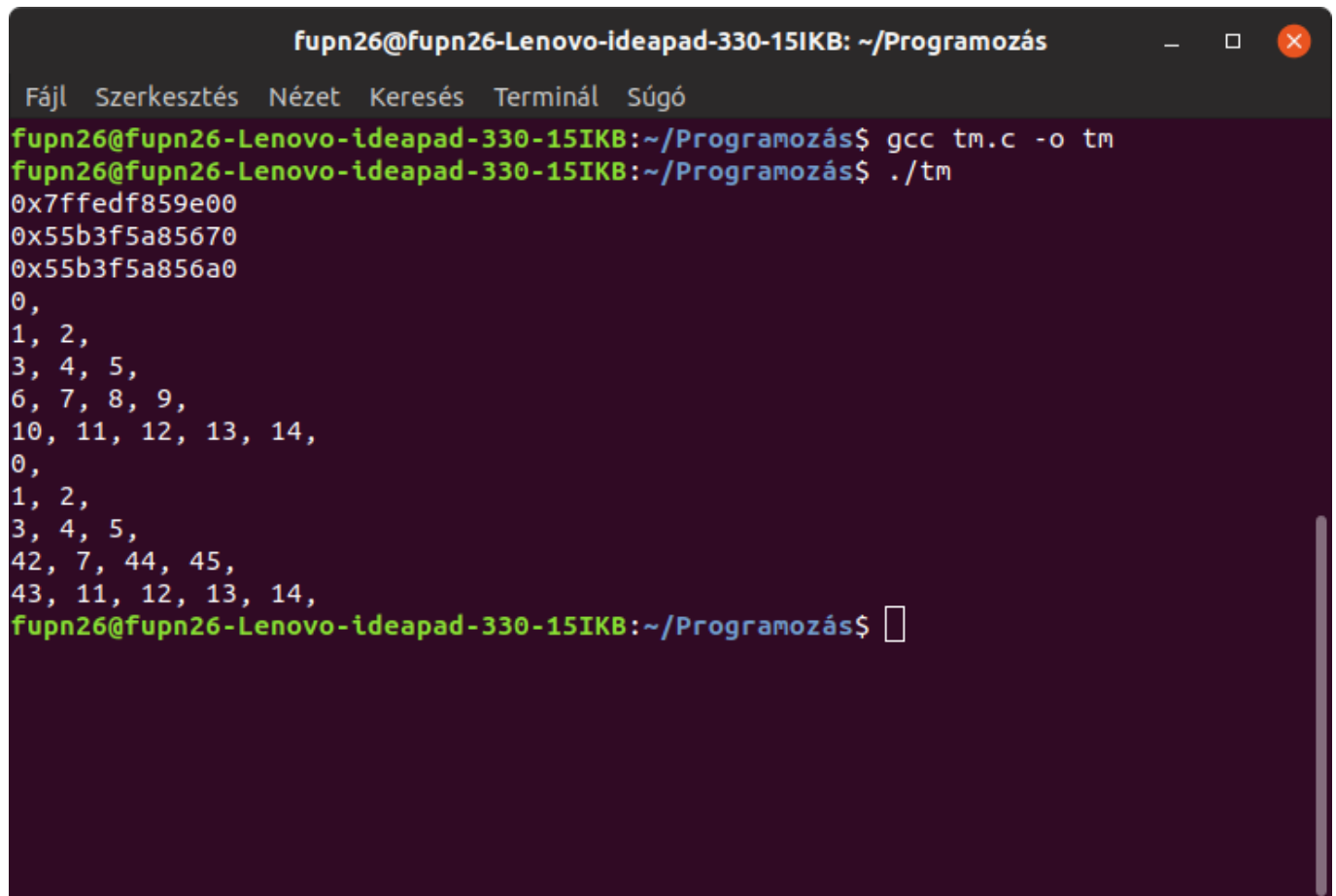
```
tm[3][0] = 42;  
(* (tm + 3)) [1] = 43; // mi van, ha itt hiányzik a külső ↵  
    (  
    * (tm[3] + 2) = 44;  
    * (* (tm + 3) + 3) = 45;
```

Az elsőt már láttad a gyakorlatban, hogy működik, mivel a programban eddig ezt használtuk, tehát a `tm` 3. sorának első elemének értékét 42-re módosítjuk. Utána a harmadik sor második elemének az értékét változtatjuk, majd a harmadik sor harmadik elemét, végül pedig a harmadik sor negyedik elemét. Itt is láthatod, hogy az első verzió sokkal egyszerűbb a többinél, főleg azoknak, akik már sokat használtak tömböket C-ben. A második lehetőségnél felmerül a kérdés, hogy elhagyható-e a külső zárójel. Elhagyható viszont, így nem a harmadik sorba lesz a módosítás, hanem a 4. sor első eleménél, mivel `* (tm + 3) [1]` azzal ekvivalens, hogy `* (tm+4)`.



```
fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás
Fájl Szerkesztés Nézet Keresés Terminál Súgó
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ gcc tm.c -o tm
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ ./tm
0x7ffc7ebc80c0
0x563ef20e6670
0x563ef20e66a0
0,
1, 2,
3, 4, 5,
6, 7, 8, 9,
10, 11, 12, 13, 14,
0,
1, 2,
3, 4, 5,
42, 43, 44, 45,
10, 11, 12, 13, 14,
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$
```

4.3. ábra. $(*(tm + 3))[1] = 43$



```
fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás
Fájl Szerkesztés Nézet Keresés Terminál Súgó
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ gcc tm.c -o tm
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ ./tm
0x7ffedf859e00
0x55b3f5a85670
0x55b3f5a856a0
0,
1, 2,
3, 4, 5,
6, 7, 8, 9,
10, 11, 12, 13, 14,
0,
1, 2,
3, 4, 5,
42, 7, 44, 45,
43, 11, 12, 13, 14,
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$
```

4.4. ábra. $*(tm + 3)[1] = 43$

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása:[itt](#)

Tanulságok, tapasztalatok, magyarázat...

Az EXOR titkosító lényegében a logikai vagyra, azaz a XOR műveletre utal, mely bitenként összehasonlítja a két operandust, és mindig 1-et ad vissza, kivéve, amikor az összehasonlított 2 Bit megegyezik, mert akkor nullát. Tehát 2 operandusra van szükségünk, ez jelen esetben a titkosítandó bemenet, és a titkosításhoz használt kulcs. Ideális esetben a kettő mérete megegyezik, így garantálható, hogy szinte feltörhetetlen kódot kapunk, mivel túl sokáig tart annak megfejtése. A mi példánkban természetesen nem lesz ilyen hosszú a kulcs, mivel ki is szeretnénk próbálni a programot. Viszont ha a kulcs rövidebb, mint a titkosítandó szöveg, akkor a kulcs elkezd ismétlődni, ami biztonsági kockázatot rejt magában.

Nézzük is meg ennek a titkosító algoritmusnak a C-beli implementációját, melynek majd a törő verzióját is elkészítjük a későbbiekben.

```
#define MAX_KULCS 100
#define BUFFER_MERET 256
```

Elsőnek a kulcs méret és a buffer méretének maximumát konstansban tároljuk el, ezek nem módosíthatóak. A szintaxisa is másabb, mint egy változó definiálásánál, itt lényegében azt adjuk meg, hogy mivel helyettesítse a megadott nevet a program a forrásban. Az előre definiált konstansok nevét általában nagy betűvel írjuk, ezzel is elkülönítve a változóktól. Nem csak számokat használhatunk konstansként, hanem stringeket, és kifejezéseket is. Érdekessége még, hogy nem program futtatásakor történik meg a behelyettesítés, hanem a már a fordítás alatt, tehát a gépi kód már behelyettesített értékeket tartalmazza.

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

Újabb érdekesség, hogy a `main()` definiálása itt egy kicsit másképp zajlik, mivel jelen esetben argumentumokat adunk át neki, az argumentumokat általában a terminálon keresztül adjuk át, amikor futtatjuk. Az `argc`-vel adjuk át az argumentumok számát, és az argumentumokra mutató mutatókat pedig az `argv` tömbben tároljuk el.

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

A `main()` belül deklarálunk két tömböt, egyikbe a kulcsot tároljuk, a másikban pedig a beolvasott karaktereket, mind a kettőnek a mérete korlátozott, melyet még a `#define` segítségével adtunk meg.

```
int kulcs_index = 0;
int olvasott_bajtok = 0;

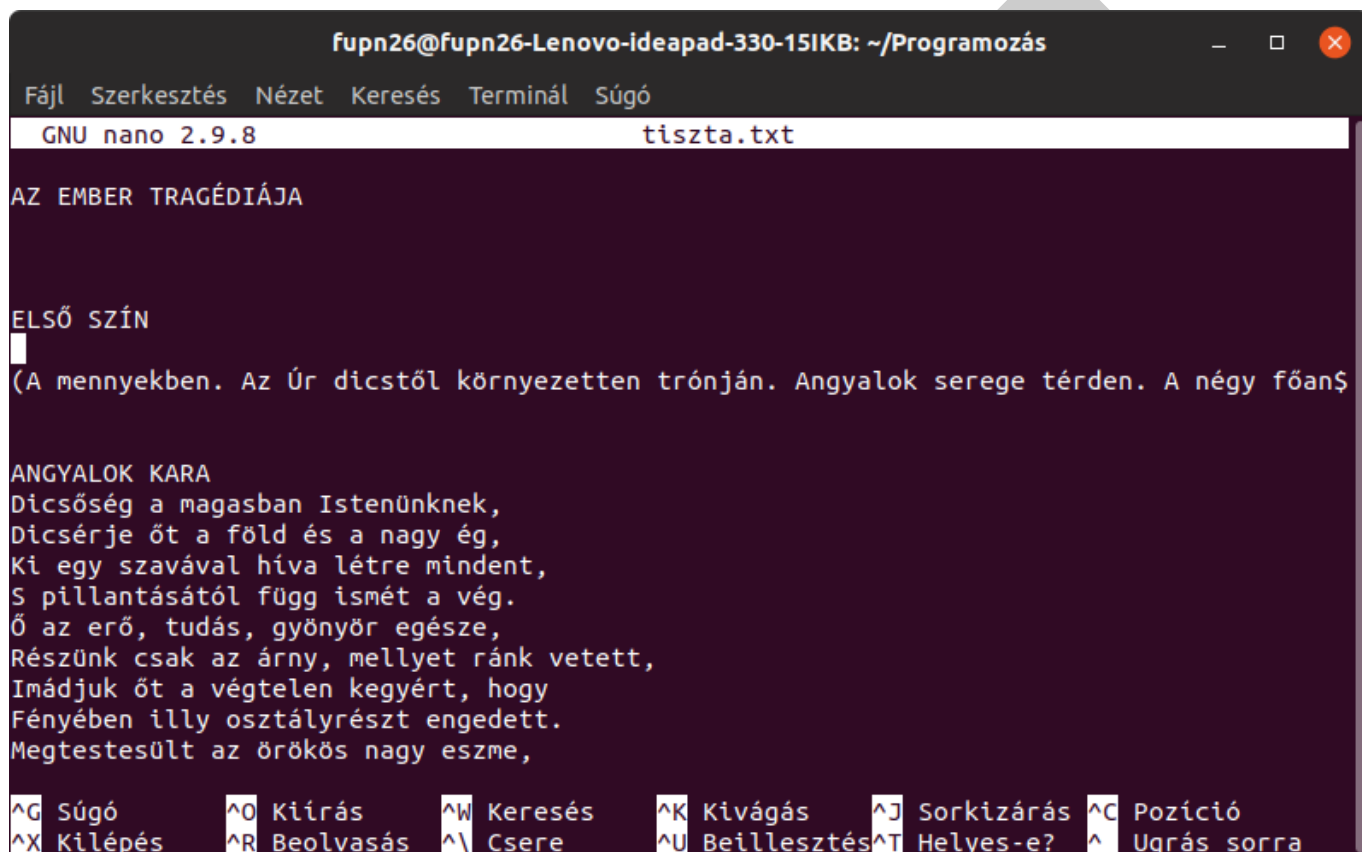
int kulcs_meret = strlen (argv[1]);
strncpy (kulcs, argv[1], MAX_KULCS);
```

Definiálunk számlálókat, melyek segítségével bejárjuk majd a kulcs tömböt, és számoljuk a beolvasott bajtokat. A kulcs méretét a `strlen()` függvénnyel kapjuk meg, mely jelen esetben visszatér a második parancssori argumentum hosszát. Ezután a `strncpy()` függvénnyel átmásoljuk az `argv[1]`-ben tárolt sztringet karakterenként a kulcs tömbbe "másolja", lényegében mindegyikhez visszaad egy pointert. A `MAX_KULCS`-sal pedig meghatározzuk, hogy mennyi karaktert másoljon át.

```
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` cikluson feltétele addig lesz igaz, ameddig a `read` parancs beolvassa a megadott mennyiségű bájtokat. A `read` 3 argumentumot kap, az egyik a file descriptor, ami megadná, hogy honna szeretnénk beolvasni a bájtokat, jelen esetben a standard inputról olvasunk, a bájtokat a buffer-ben tároljuk egészen addig, ameddig el nem érjük a megadott mennyiséget, amit `BUFFER_MERET` definiál. A beolvasott bájtok számát adja vissza. Ezután pedig végigmegyünk elemenként a bufferben eltárolt karaktereken és össze EXOR-ozzuk a kulcs tömb megfelelő elemével, majd inkrementáljuk a `kulcs_index`-et 1-el, mely egészen addig nő, ameddig el nem érjük a `kulcs_meret`-et, ekkor lenullázódik. Végezetül pedig kiírjuk a buffer tartalmát a standard outputra.



```
fupn26@fupn26-Lenovo-ideapad-330-151KB: ~/Programozás
Fájl Szerkesztés Nézet Keresés Terminál Súgó
GNU nano 2.9.8 tisza.txt

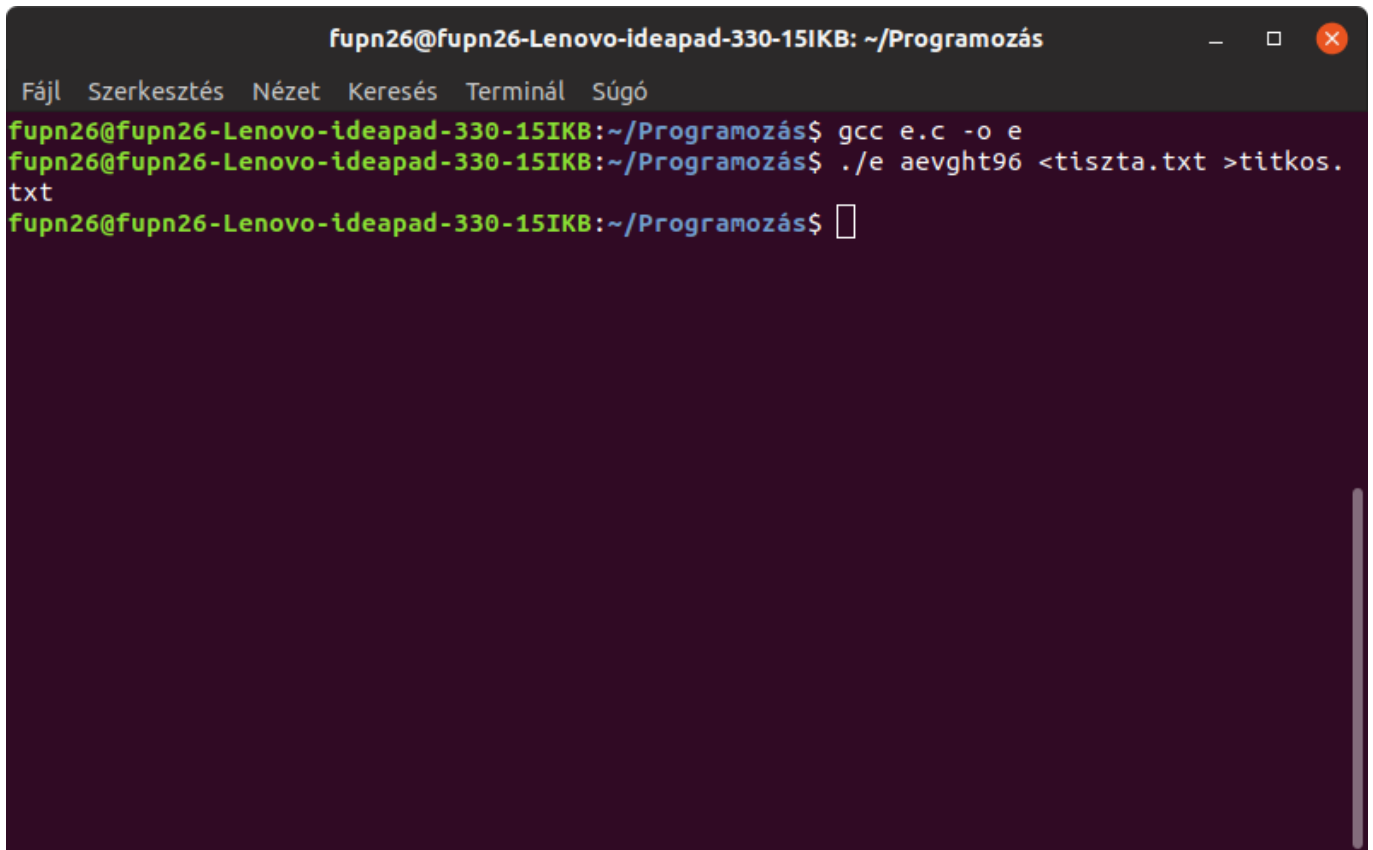
AZ EMBER TRAGÉDIÁJA

ELSŐ SZÍN
█
(A mennyekben. Az Úr dicstől környezetten trónján. Angyalok serege térden. A négy főan$

ANGYALOK KARA
Dicsőség a magasban Istenünknek,
Dicsérje őt a föld és a nagy ég,
Ki egy szavával híva létre mindent,
S pillantásától függ ismét a vég.
Ő az erő, tudás, gyönyör egésze,
Részünk csak az árny, mellyet ránk vetett,
Imádjuk őt a végtelen kegyért, hogy
Fényében illy osztályrészt engedett.
Megtestesült az örökös nagy eszme,
```

^G Súgó	^O Kiírás	^W Keresés	^K Kivágás	^J Sorkizárás	^C Pozíció
^X Kilépés	^R Beolvasás	^_\ Cseré	^U Beillesztés	^T Helyes-e?	^_ Ugrás sorra

4.5. ábra. Titkosítandó szöveg



```
fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás
Fájl Szerkesztés Nézet Keresés Terminál Súgó
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ gcc e.c -o e
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ ./e aevght96 <tiszta.txt >titkos.
txt
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$
```

4.6. ábra. Fordítás és futtatás


```
fupn26@fupn26-Lenovo-ideapad-330-151KB: ~/Programozás
GNU nano 2.9.8 titkos.txt

[]?V"%6|dA1$&/r(+-)~3^Vko3+;;V2?&~3^^ E^[^B^F^Z@S
^G^S   FTxLA^UH^PPU^R^Q^DTR^W^X^^M^N\B^U^@^XG^_\F^O^O^FZ^Yw^O^B^O^F^D^[R^V$
IVm,^]ZE^D^M^MT^UE^WG^N^Z^EE^[TX^V^O^D^Q^^H^QME|,^AT\Q^XE^E^] ^B^W^D^ZG^@^$
^QW^V^H ^Z^^H^[JL^U^K^Q^F^R^_ ^BG^M^Z^S^E^@^B^SFT3{^D^B^B^B^[^@\E^Z^SH^UC^V^D^$

^B^TI_ ^@^XIH~^W^H^_ ^K^D^]
^L^QG^M^XS^WV^S^M^Z^S^M^_\^FX^Y<,^L^QG^M^S^@^V
^@^D^_ ^Y^F^SH^AS^Q^X^NH^_ \Z^MKVm.^QU^V   ^SDTO_ ^M^@^A^Y^Y^@^R^J^F^QTB^_KH$
^C^B^QT^V^L^@H^Q^O^R^_ ^S^UH^V\X^O^@^B^B^CX^Y< ^H^_ ^\T\Z^[^@^@^\^[R^V^M^E $

[ 9 sor beolvasva ]
^G Súgó      ^O Kiírás    ^W Keresés    ^K Kivágás    ^J Sorkizárás  ^C Pozíció
^X Kilépés    ^R Beolvasás ^\ Cseré      ^U Beillesztés ^T Helyes-e?  ^_ Ugrás sorra
```

4.7. ábra. Titkosított szöveg

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása:



Megjegyzés

Ebben a feladatban tutoráltként részt vett: Molnár András Imre, Pócsi Máté.

Tanulságok, tapasztalatok, magyarázat...

Az előző feladatát kell most java-ban megoldani. Mivel a Java a C és a C++ alapján készült, ezért az előző kód implementációja nem nehéz feladata, de természetesen vannak különbségek, melyeket érdemes megemlíteni.

Az első és legfontosabb dolog, hogy a Java a C-vel ellentétben Objektum-orientált programozási nyelv, azaz létretudunk hozni objektumokat, úgynevezett class-okat melyekkel bizonyos utasításokat tudunk végrehajtani. A C++-ban kicsit arra hasonlít, mintha egy változót hoznánk létre, csak nem egy típust írunk elé,

hanem a class nevét. De ott a class elkülönül a `main()` függvénytől, míg a Java-ban szerves részét képezi a classnak, tehát az egész program egy nagy class-ból áll.

```
public class ExorTitkosító {  
    ...  
    ...  
}
```

Tehát ezen belül kell mindent deklarálni és definiálni. A class-nak lehet `public` és `private` elemei, ezt a függvény deklaráció elé írjuk, jelen esetben nem alkalmazunk `private` elemeket, de a lényeg az, hogy azokhoz nem férünk hozzá közvetlenül a classon kívül, csak akkor ha a class tartalmaz egy `public` függvényt, amivel ki tudjuk nyerni az értéket.

Elsőnek nézzük meg a `main` tartalmát:

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

Az `main` fejléce egy kicsit furán néz ki, mivel egy kicsit bonyolultabb, mint azt C-ben megszoktuk. A `public` annyit tesz, hogy elérhető a class-on kívül is. A `static` azt jelenti, hogy a `main` része a class-nak, de egy külön álló objektum, nem része egyik beágyazott objektumnak sem. A `void`-ot már ismerjük, tehát ennek a `main`-nek nincs visszatérési értéke. Ráadásul itt is képesek vagyunk parancssori argumentumokat átadni a programnak a `String[] args` segítségével. Egy másik újdonság a `try` és a `catch` használata, mely lényegében a Java-ban és a C++-ban a kivételkezeléshez nélkülözhetetlen. A `try` blokk tartalmazza az utasításokat, ha valami hiba történik, akkor dobtunk egy hibát, a `catch` "elkapja" és visszaad egy hibaüzenetet a terminálba. Jelen esetben ha nem adunk meg kulcsot, akkor kapunk hibát. A `try`-ban tárhelyet foglalunk az `ExorTitkosító` függvénynek, melynek átadjuk a kulcsot, a bemenetet és a kimenetet.

```
public ExorTitkosító(String kulcsSzöveg,  
    java.io.InputStream bejövőCsatorna,  
    java.io.OutputStream kimenőCsatorna)  
    throws java.io.IOException {  
  
    byte [] kulcs = kulcsSzöveg.getBytes();  
    byte [] buffer = new byte[256];  
    int kulcsIndex = 0;  
    int olvasottBájtok = 0;  
  
    while((olvasottBájtok =  
        bejövőCsatorna.read(buffer)) != -1) {
```


4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...



Megjegyzés

Ebben a feladatban tutorként és tutoráltként részt vett: Imre Dalma, György Dóra.

Az előbbi feladatokban láthattad, hogy hogyan lehet titkosított szövegeket készíteni az EXOR titkosítás segítségével. Most ennek az ellentétét kell megcsinálnunk, ami egy kicsit trükkösebb, és talán nem is tökéletes, de az előző feladatban generált titkos szöveg feltörésére alkalmas lesz.

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

Ezúttal is meghatározzunk bizonyos konstansokat, ebből a kulcs_meret érdekes, mert feltételezzük, hogy a kulcs 8 elemből áll, már itt látni, hogy ez nem lenne túl hatékony a való életben.

```
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;
}
```

Az atlagos_szohossz függvénnyel kiszámítjuk a bemenet átlagos szóhosszát, argumentumként átadjuk egy tömböt, és annak a méretét. Majd egy for ciklussal bejárjuk, és minden elem után hozzáadunk 1-et az sz változóhoz. Visszatérési értéként pedig a tömb méretének és a számlálónka a hányadosát adjuk.

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

```
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");  
  
}
```

A `tiszta_lehet` függvény az átlagos szóhossz segítségével vizsgálja, hogy a fejtésben lévő kód tiszta-e már. Itt meg kell felelni az átlagos magyar szóhossznak, és a leggyakoribb szavakat tartalmaznia kell. Felmerül a kérdés, hogy mi történik akkor, ha ezeknek nem felel meg a törni kívánt szöveg? Sajnos akkor nem tudjuk feltörni, tehát ez egy újabb gyengesége a programunknak.

```
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;  
  
    }  
  
}
```

Az `exor` függvény ugyan azt csinálja, mint az EXOR titkosító, mivel ha valamit kétszer EXOR-ozunk, akkor visszakapjuk önmagát. Lényegében ezzel állítjuk vissza a tiszta szöveget. Ez argumentumként megkap egy lehetséges kulcsot, annak méretét, és magát a titkosított szöveget, annak méretével együtt.

```
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);  
  
}
```

Az `exor_tores` hívja a korábban definiált függvényeket, és 0-át vagy 1-et ad vissza, attól függően, hogy tiszta-e már a szöveg.

```
char kulcs[KULCS_MERET];
char titkos[MAX_TITKOS];
char *p = titkos;
int olvasott_bajtok;
```

Mostmár áttérhetünk a main belüli deklarációkra, definíciókra. Elsőnek deklarálnunk egy kulcs[] tömböt, és egy titkos[] tömböt. Ezek mérete a fentebb már rögzített értékekkel lesz azonos. Majd definiálunk egy mutatót, mely a titkos[] tömbre mutat, és deklaráljuk a beolvasott bajtok számlálóját.

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

A while ciklussal addig olvassuk a bajtokat, ameddig a buffer be nem telik, vagy a bemenet végére nem érünk.

```
for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
    titkos[p - titkos + i] = '\\0';
```

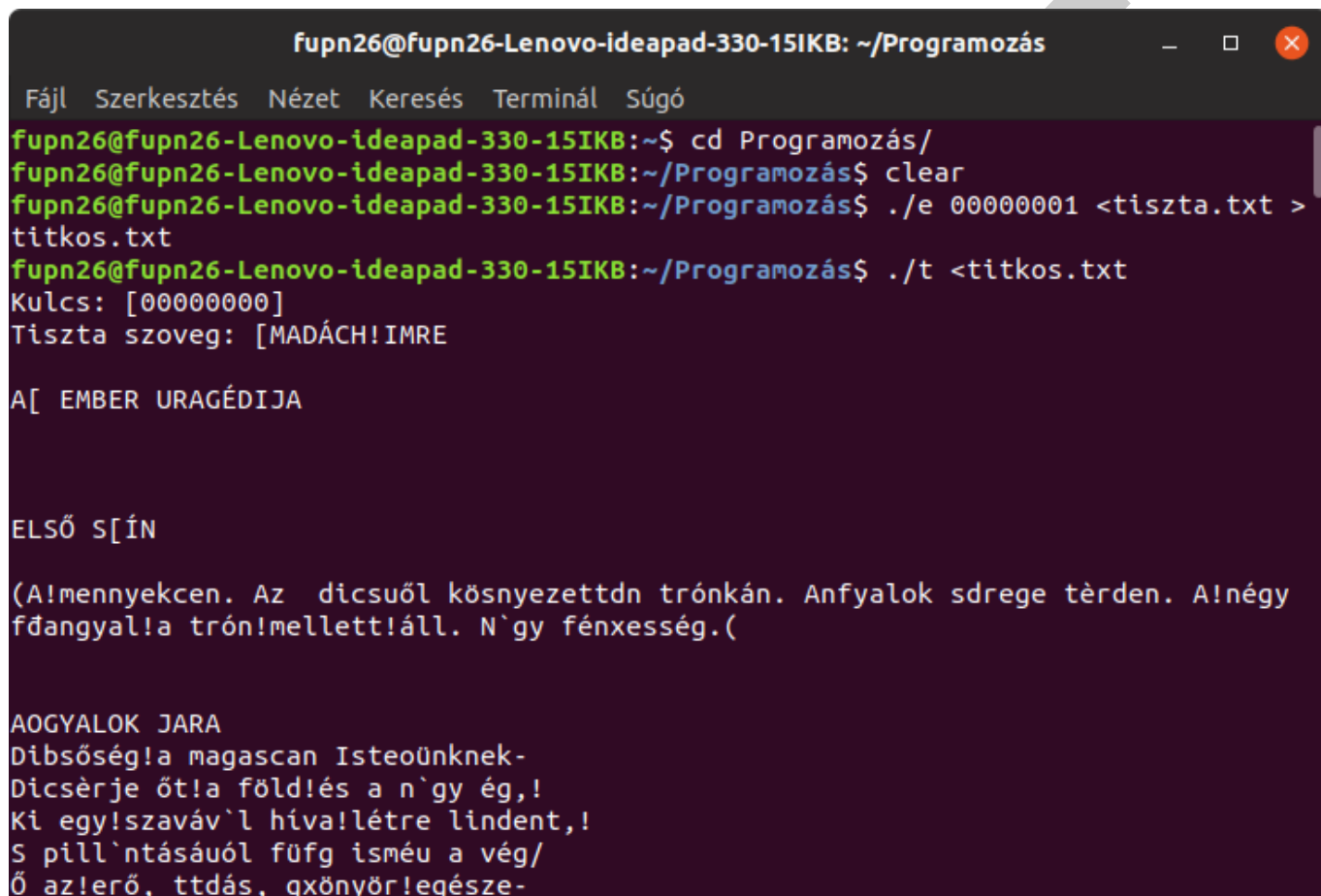
Ezzel a for ciklussal kinullázzuk a buffer megmaradt helyeit, és utána pedig előállítjuk az összes lehetséges kulcsot:

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

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

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

Végig futatjuk az összes lehetőségen a `for` ciklust, majd meghívjuk az `exor_tores` függvényt. Ha ez igazat ad, tehát a visszatérési érték 1, akkor kiíratjuk az aktuális kulcsot és a feltört szöveget. Ahogy látod ez csak olyan kódokat tud feltörni, amit számokkal kódoltunk. Ezután pedig újra meghívjuk az `exor` függvényt, ezzel elkerülve a második buffer létrehozását. Az előző feladatban én a betűket és számokat is használtam, ezt is ki lehetne bővíteni, hogy fel tudjuk törni azt a kódot, de az a baj, hogy nagyon sokáig tartana. Tehát ennél a megoldásnál maradván újratíkosítottam az `tiszta.txt`-t



```
fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás
Fájl Szerkesztés Nézet Keresés Terminál Súlyó
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~$ cd Programozás/
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ clear
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ ./e 00000001 <tiszta.txt >
titkos.txt
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ ./t <titkos.txt
Kulcs: [00000000]
Tiszta szoveg: [MADÁCH!IMRE
A[ EMBER URAGÉDIJA
ELSŐ S[ÍN
(A!menyekcen. Az dicsuöl kösnyezettdn trónkán. Anfyalok sdrege tēden. A!négy
fđangyal!a trón!mellett!áll. N`gy fénxesség.(
AOGYALOK JARA
Dibsőség!a magascan Isteoünknek-
Dicsérje őt!a föld!és a n`gy ég,!
Ki egy!szaváv`l híva!létre lindent,!
S pill`ntásáuól fűfg isméu a vég/
Ő az!erő, ttdás, gxönyör!egésze-
```

4.9. ábra. Törés

A `t.c` fordítva és futtatva a képen látható módon folyamatosan kapjuk a lehetséges megfejtéseket, fontos hogy nem kell végig várni a folyamatot, mert az sokáig tart, `Ctrl+c`-vel meg tudod állítani. Jelen esetben láthatod, hogy sikerült megtalálnia a megfelelő kódot.

4.10. ábra. Törés

4.5. Neurális OR, AND és EXOR kapu

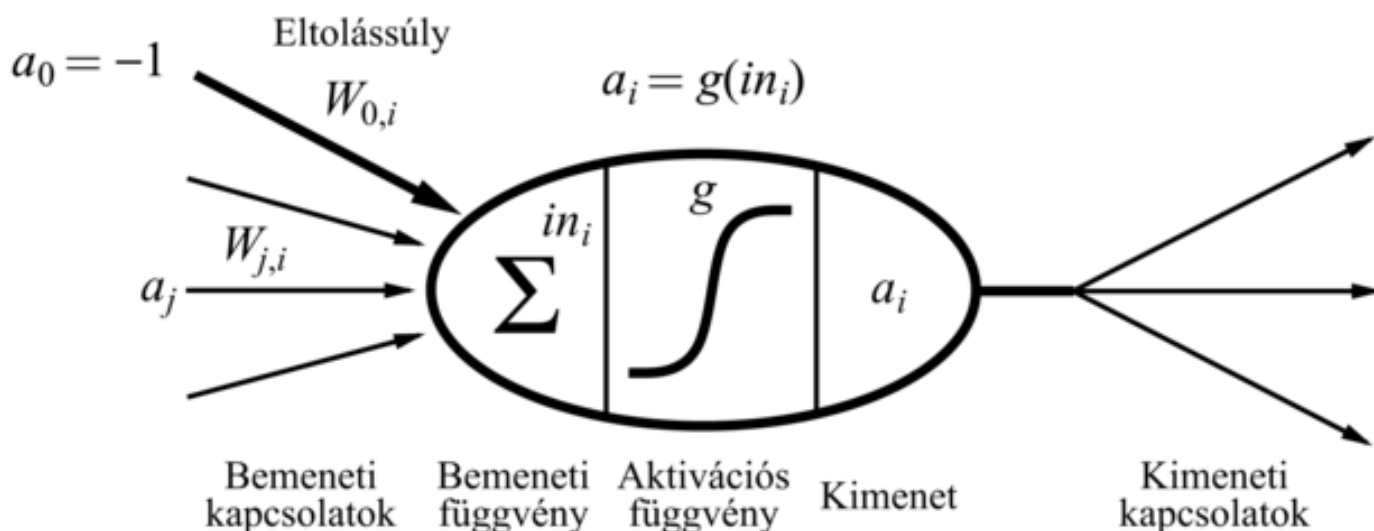
R

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

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

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban újra visszatérünk a Monty Hall problémánál megismert R nyelvhez. Segítségével neurális hálózatot fogunk létrehozni, mely képes "tanulni", és megközelíteni az általunk megadott megfelelő értékeket. A hálózat a nevét a neuronról kapta, mely gyunk egy sejtje. Feladata az elektromos jelek összegyűjtése, feldolgozás és szétterjesztése. Az a feltételezés, hogy az agyunk információfeldolgozási képességét ezen sejtek hálózata adja. Éppen emiatt a mesterséges intelligencia kutatások során ennek a szimulálást tűzték ki célul. A neuron matematikai modeljét McCulloch és Pitts alkotta meg 1943. Ezt mutatja a következő ábra:



4.11. ábra. Neuron

A lényeg, hogy a neuron akkor fog tüzelni, ha a bemenetek súlyozott összege meghaladnak egy küszöböt. Az aktivációs függvény adja meg a kimenet értékét.

Ezt a modellt fogjuk implementálni egy R programba. Az első részben a logikai vagyot tanítjuk meg a neurális hálózatnak, mely 1-et ad vissza, kivéve, ha mind a két operandusa 0, mert akkor 0-át.

```
library(neuralnet)

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

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

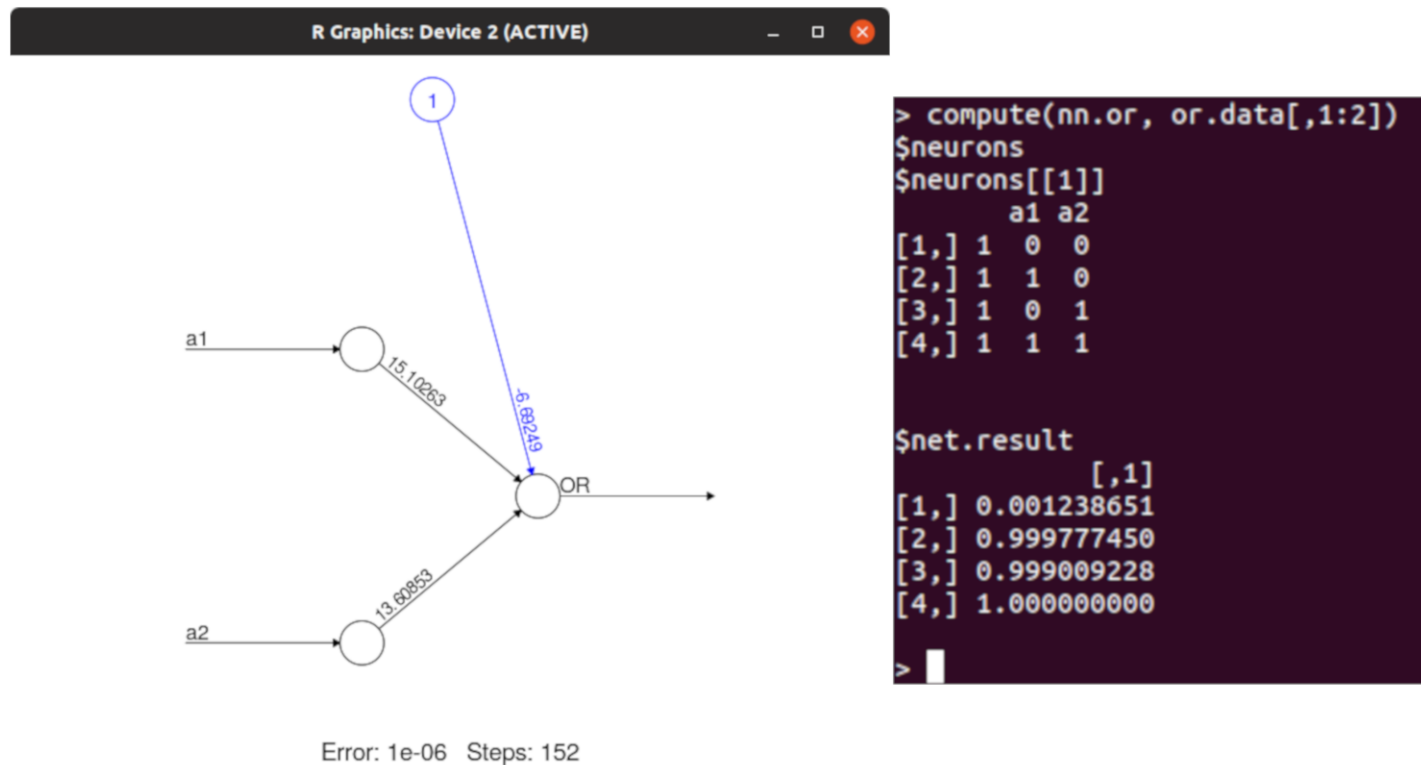
nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE, ←
  stepmax = 1e+07, threshold = 0.000001)

plot(nn.or)

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

A neurarnet könyvtárat kell használni a feladat megoldásához. Az a1, a2, OR változókhoz hozzárendeljük a megfelelő értékeket, amit szeretnénk betanítani a programnak. Majd az or.data-ban tároljuk el ezeket a változókat, melyek a betanítás alapjául fognak szolgálni. Az nn.or változó értékéül pedig a neuralnet függvény visszatérési értékét adjuk. Ennek a függvénynek több argumentuma van, ezeket vegyük gyorsan sorra. Az első elem maga a formula, amit meg kell tanulni a programnak, majd jön egy minta, mely alapján a tanulást végzi, a harmadik argumentum a rejtett neuronok számát adja meg. A linear.output egy logikai változó, melynek értékét TRUE-ra kell állítani, ha azt szeretnénk, hogy az aktivációs függvény ne fusson le a kimeneti neuronokra. A stepmax adja meg a maximum lépésszámát a neuron háló tanulásának, mely befejeződik, ha elérjük ezt az értéket. A threshold pedig számérték, mely meghatározza a hiba részleges deriváltjainak küszöbértékét, a tanulás megállási kritériumaként funkcionál. Ezután pedig plot

kirajzoltatjuk a neurális háló tanulási folyamatának egy állapotát. A neuralnet minden bementhez kiszámol egy súlyozást, amellyel megszorozza a bemenet értékét. Majd pedig kiszámítjuk a logikai művelet neurális háló szerinti értékét, és összevetjük a referencia értékekkel.



4.12. ábra. OR

A jobb oldali kis ablakban láthatod, hogy egészen jó közelítéssel sikerült visszaadnia a megfelelő értékeket a programnak. Ugyan ezt megismételjük az AND logikai operátorral is, mely akkor lesz igaz, azaz 1, ha mind a 2 operandusa 1, amúgy hamis.

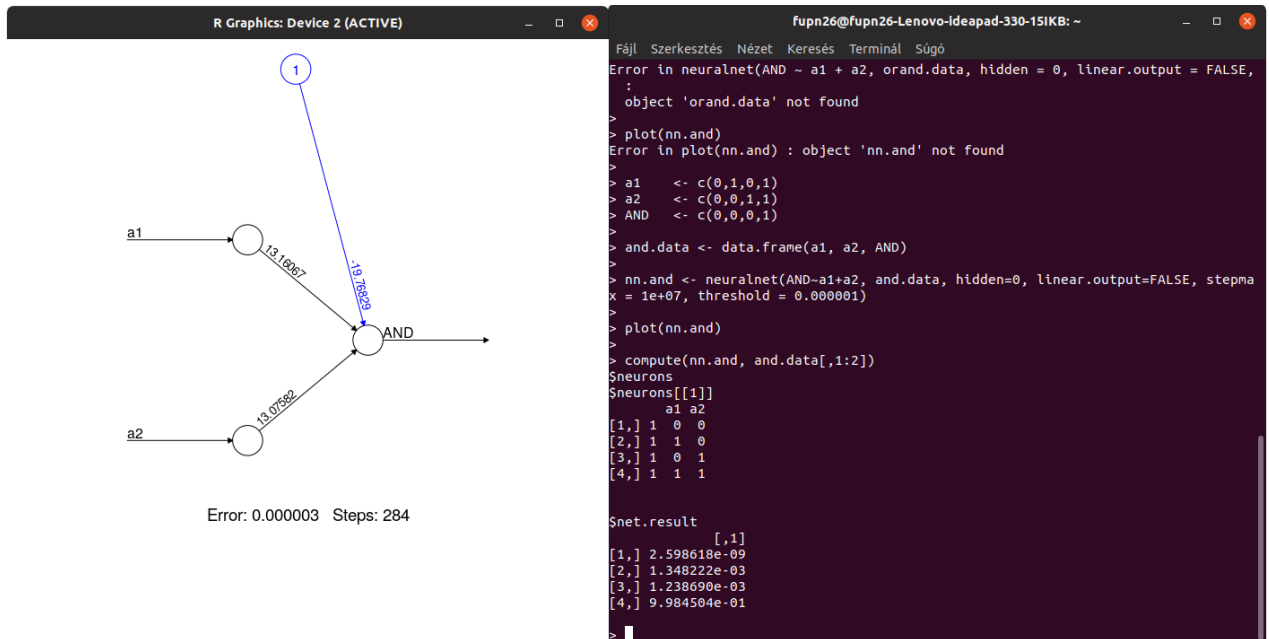
```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
AND <- c(0,0,0,1)

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

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

plot(nn.and)

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



4.13. ábra. AND

Ezután pedig jönne az EXOR, viszont ez már nem annyira egyszerű. Amikor régen ezt a technológiát kitalálták, és az EXOR nem működött, sokan elpártoltak tőle. Majd a kor nagy matematikusai megfejtették, hogy nem lehetetlen feladat, csak egy apróságra van szükség, létre kell hozni a rejtett neuronokat, melyek segítik a tanulást.

```

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

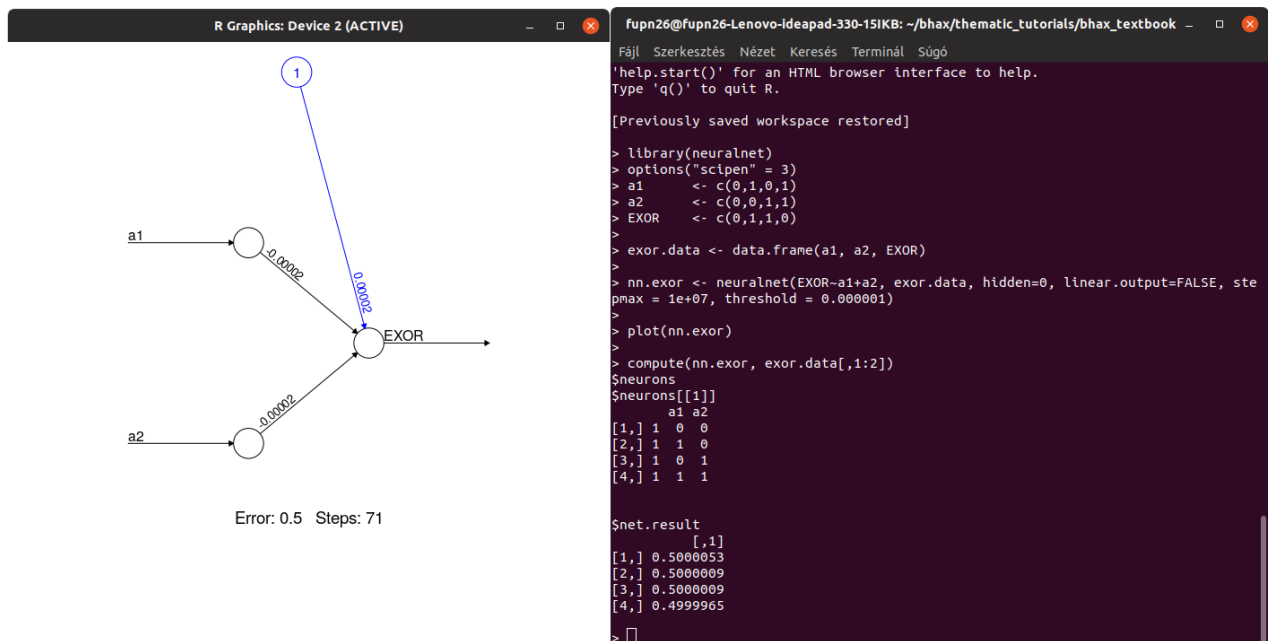
exor.data <- data.frame(a1, a2, EXOR)

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])

```



4.14. ábra. EXOR első próba

Látható, hogy nagyon nagy a hibaarány, és még az eredmények is tévesek lettek, mindegyik 0,5 körüli lett, az 1 és a 0 helyett. A `neuralnet` hidden argumentuma 0-ra volt állítva ebben az esetben, tehát nem használtunk rejtett neuronokat. Ha ezt átállítjuk, akkor a következőt kapjuk:

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

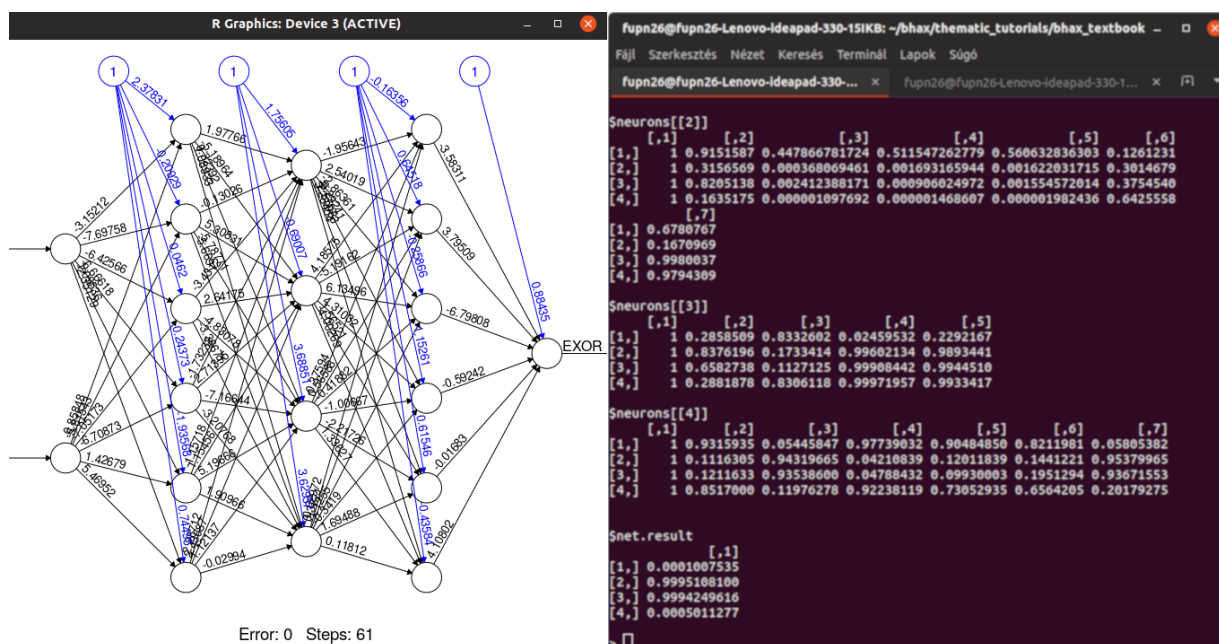
exor.data <- data.frame(a1, a2, EXOR)

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])
```

Módosítottuk a rejtett neuronok számát, elsőnek 6, majd 4 és végül megint 6 neuront hozunk létre. Lássuk az eredményt:



4.15. ábra. EXOR második próba

Láthatod, hogy sikerült, megkaptuk a helyes eredményeket, bár az ábra jóval bonyolultabb lett a rejtett neuronok miatt.

4.6. Hiba-visszaterjesztéses perceptron

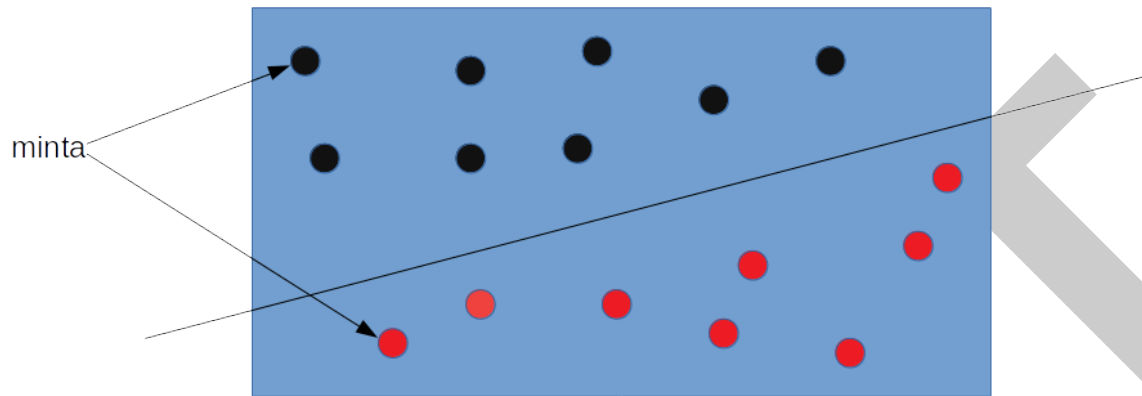
C++

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

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

Tanulságok, tapasztalatok, magyarázat...

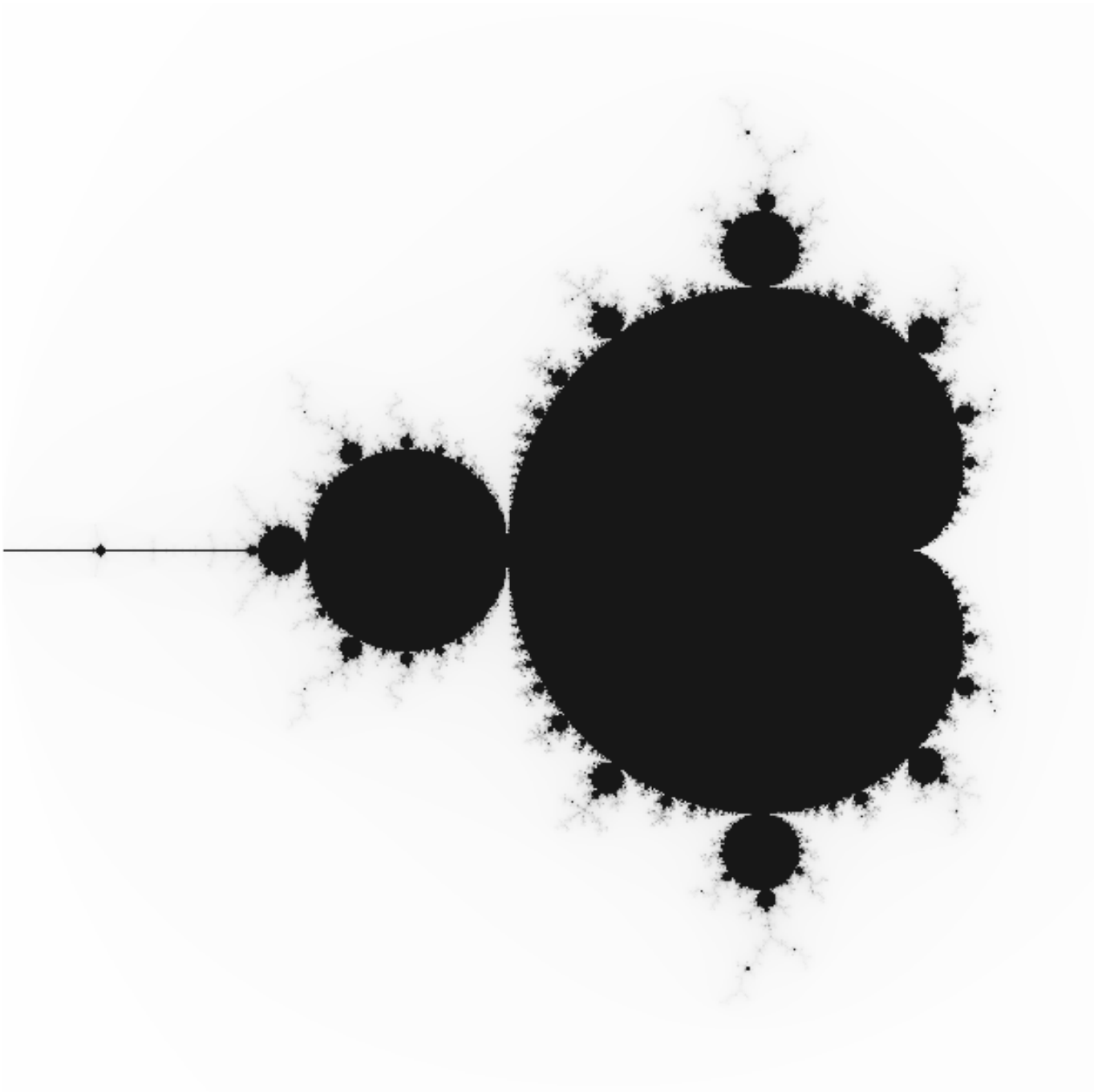
Ebben a feladatban folytatjuk a elmélkedést a neuron hálózatokra, ezen belül perceptronokról lesz szó. Ez egy algoritmus amely a számítógépnek "megtanítja" a bináris osztályozást. Ide is berakhatnám az előző fejezetben lévő képet a neuronról, ami egy bemenetet kap, és egy bizonyos pontot elérve "tüzel", ad egy kimenetet. Itt is hasonló dologról van szó:



4.16. ábra. Perceptron bemenet

Tehát van egy halmaz amiben vannak piros és fekete pontok, a fekete pontok a vonal felett vannak, a pirosak pedig alatta. Adok a perceptronnak bemenetként egyet-egyet mind a kettőből, és a képes lesz megmondani a többite, hogy a vonal felett van-e, vagy alatta. Ezért nevezzük bináris osztályozásnak, mert van a vonal feletti osztálya és az alattiaké, ez a kapcsolat könnyen reprezentálható eggyel és nullával.

Akkor lássuk magát a programot. Most C++-ban fogunk dolgozni, melyről tettem már említést a Java-s feladatban, szóval annyit már tudsz, hogy ez is egy objektum-orientált nyelv, szóval a class-okat képtelenség lesz elkerülni. 3 fájlra lesz szükség, abból az egyikkel, `mandelpng.cpp`-vel most nem foglalkoznék, hiszen a következő fejezetben pont erről lesz szó. Egyelőre legyen elég annyi, hogy ezzel tudunk készíteni egy képet, ami a Mandelbrot halmazt ábrázolja.



4.18. ábra. Mandelbrot-halmaz

Ez a kép lesz a bemenete a perceptron algoritmust szimuláló programunknak. Az első dolog amit nagyon fontos, hogy most 2 fájlból áll a programunk. A `ml.hpp` és `main.cpp`-re lesz szükségünk. Az `ml.hpp` tartalmazza a Perceptron class-t, ezzel a `main.cpp` sokkal átláthatóbb. Ezt technikát gyakran használják, szóval érdemes megtanulni. Magát a class-t most nem vesézzük ki, helyette vessünk egy pillantást a `main`-re.

```
#include <iostream>
#include "ml.hpp"
```



```
#include <png++/png.hpp>

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

Itt láthatod, hogyan kell az `ml.hpp`-t includálni, maga a `main` fejrésze pedig az EXOR-nál már jól megszokott felépítést követi.

```
png::image <png::rgb_pixel> png_image (argv[1]);
```

Ezzel létrehozunk egy üres png-t, melynek mérete megegyezik a bemenetként kapott fájl méretével. Ehhez van szükség a `png.hpp` headerre. Egy fontos dolog még, hogy miért van szükség a dupla kettőpontra. A C++-ban létezik egy olyan fogalom, hogy névterek. Erről még fogunk szót ejteni, de gyelőre annyi elég lesz róla, hogy a `png.hpp`-ben használt függvények, változók elé oda kell rakni a `png::`-ot. Ugyan így van ez az `iostream`-ben lévő `cout`-tal is, mely a standard kimenetre írja ki azt amit szeretnénk. Előtte az `std::` prefixet használjuk. Lényegében ez azért hasznos, mert ha lenne a `png.hpp`-ben és az `iostream`-ben is `cout`, akkor ezzel meg tudjuk őket különböztetni. Persze kicsit hosszú mindegyik elé odaírni, és lesz is rá majd megoldás, de ennek használatát egyelőre kerüljük.

```
int size = png_image.get_width() * png_image.get_height();

Perceptron* p = new Perceptron (3, size, 256, 1);

double* image = new double[size];
```

A kép méretét eltároljuk egy változóban, majd létrehozunk felhasználó által definiált típust, ez a `Perceptron`, melyet a `ml.hpp` `Perceptron` osztályában találunk. Lényegében itt adjuk meg a rétegek számát, jelen esetben ez 3, majd azt adjuk meg, hogy hány darab neuront szeretnénk az egyes rétegekben. Az utolsóba csak 1-et rakunk, mely az eredményt adja majd. Definiálunk még egy `double*` pointert, melyet `size`-zal megegyező memóriaterületre állítunk rá.

```
for (int i = 0; i<png_image.get_width(); ++i)
    for (int j = 0; j<png_image.get_height(); ++j)
        image[i*png_image.get_width() + j] = png_image[i][j].red;
```

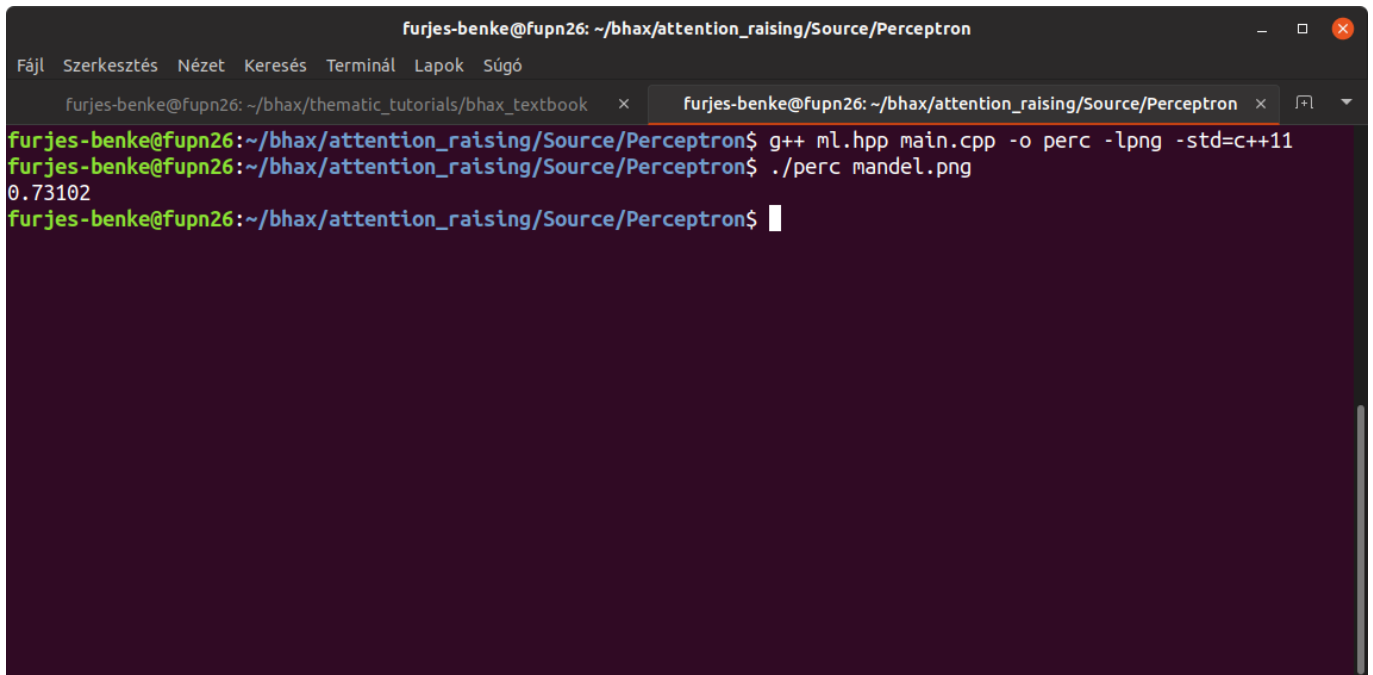
Az egymásba ágyazott `for` ciklusok segítségével az újonan lefoglalt tárbba másoljuk bele a beolvasott kép pixeleinek piros komponensét.

```
double value = (*p) (image);
```

Itt meghívjuk a `Perceptron` class `()` operátorát, mely vissza fogja adni az nekünk szükséges eredményt. Végezetül ezt már csak kiírjuk a `cout`-tal.

```
std::cout << value << std::endl;
```

Futassuk:



```
furjes-benke@fupn26: ~/bhax/attention_raising/Source/Perceptron
Fájl Szerkesztés Nézet Keresés Terminál Lapok Súgó
furjes-benke@fupn26: ~/bhax/thematic_tutorials/bhax_textbook x furjes-benke@fupn26: ~/bhax/attention_raising/Source/Perceptron x
furjes-benke@fupn26:~/bhax/attention_raising/Source/Perceptron$ g++ ml.hpp main.cpp -o perc -lpng -std=c++11
furjes-benke@fupn26:~/bhax/attention_raising/Source/Perceptron$ ./perc mandel.png
0.73102
furjes-benke@fupn26:~/bhax/attention_raising/Source/Perceptron$
```

4.19. ábra. Fordítás és futtatás

Fontos figyelembe venni, hogy nem mindig fogja ugyan azt az értéket adni, szóval nem kell kétségbe esni, ha nem ugyan az jön ki, mint a képen.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása:

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

Megoldás videó:

Megoldás forrása:

5.3. Biomorfok

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

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

Tanulságok, tapasztalatok, magyarázat...

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása:

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

Megoldás forrása:

Megoldás videó:

Megoldás forrása:

5.6. Mandelbrot nagyító és utazó Java nyelven

DRAFT

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás videó:

Megoldás forrása:

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!

6.2. LZW

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

Megoldás videó:

Megoldás forrása:

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása:

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása:

6.5. Mutató a gyökér

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

Megoldás videó:

Megoldás forrása:

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása:

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:

Tanulságok, tapasztalatok, magyarázat...

7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.2. Szoftmax R MNIST

R

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.4. Deep dream

Keras

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.5. Robotpszichológia

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Megoldás forrása:

9.2. Weizenbaum Eliza programja

Éleszd fel Weizenbaum Eliza programját!

Megoldás videó:

Megoldás forrása:

9.3. Gimp Scheme Script-fu: króm effekt

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

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

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

Tanulságok, tapasztalatok, magyarázat...

9.4. 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...

9.5. Lambda

Hasonlítsd össze a következő programokat!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9.6. Omega

Megoldás videó:

Megoldás forrása:

DRAFT

10. fejezet

Helló, Kernighan!

10.1. Pici könyv

Alapfogalmak

Ha a számítógépek programozási nyelveiről beszélünk, akkor fontos említést tenni róla, hogy milyen szintekre tudjuk őket osztani. Az első a gépi nyelv, mely lényegében 0-kból és 1-kből álló bináris kód. Ezt követi az assembly szintű nyelv, amely már egy kicsit közelebb van az emberi nyelvekhez, de még alacsony szintű programozási nyelvnek minősül. Legvégül pedig tegyünk említést a könyv fő témájáról a magas szintű programozási nyelvekről. Ezek már közel állnak az emberek által is értelmezhető nyelvekhez, főleg az angolra épülnek.

A magas szintű programozási nyelven írt programot forrásprogramnak nevezzük. Ennek az előállításához be kell tartani bizonyos az adott nyelvre jellemző formai, szintaktikai szabályokat és a tartalmi, szemantikai szabályokat.

A processzorok saját gépi nyelvvel rendelkeznek, és csak az ezen a nyelven írt programokat képesek végrehajtani. Tehát a forráskódokat át kell alakítani a gép által értelmezhető gépi kódra. Erre két analógia létezik, az egyik a fordítóprogramos megoldás, a másik az interpreteres.

A fordítóprogram egyetlen egységként kezeli a forrást, és lexikai, szintaktikai, szemantikai elemzést hajt végre, majd legenerálja a gépi kódot. Ez még nem futtatható, ebből a kapcsolatszerkesztő állít elő futtatható programot, melyet a betöltő behelyez a tárbá, és a futtató rendszer felügyeli a futását. Bizonyos esetekben lehetőség van arra, hogy nem nyelvi elemeket használjunk egy forrásprogramban, de ilyenkor szükség van egy előfordítóra is.

Az interpreteres megoldás nem készít tárgykódot, viszont a fentebb említett 3 elemzést végrehajtja. Utasításonként sorra veszi a forrásprogramot, értelmezi, és végrehajtja. tehát rögtön kapjuk meg eredményt. Bizonyos nyelvek esetén mind az interpreteres, mind a fordítóprogramos megoldást alkalmazzák.

Minden programozási nyelvhez tartozik egy hivatkozási nyelv, mely a szemantikai és szintaktikai szabályokat határozza meg. Emellett léteznek még implementációk. Az egyes rendszereken több fordítóprogram és interpreter létezik, és ennek következtében az implementációk nem kompatibilisek egymással, ez pedig meggátolja a programok tökéletes hordozását a platformok között.

A programozó dolgának megkönnyítése érdekében létrejöttek az integrált grafikus felületek (IDE), melyek egy csomagban tartalmazznak minden szükséges eszközt a programok megírásához, és futtatásához.

A programnyelvek osztályozása

Összeségében 2 fő csoportra oszthatjuk a programozási nyelveket: az imperatív nyelvekre és a deklaratív nyelvekre.

Az imperatív nyelvek algoritmikus nyelvek, tehát a programozó maga kódolja le az algoritmust, amit a processzor majd végrehajt. A program utasításokból épül fel. Legfőbb eszköz a változó, mely segítségével el tudunk érni tárterületet, és tudjuk módosítani annak tartalmát. Jelen esetben az algoritmusok teszik ezt. Az imperatív nyelveknek 2 alcsoportja van: az Eljárásorientált nyelvek és az Objektumorientált nyelvek.

Ezzel szemben a deklaratív nyelvek nem algoritmikusak, a programozó csak felvázolja a problémát, és a nyelvi implementációkban be van építve a megoldás megkeresésének módja. Memóriaműveletekre nincs lehetőség, vagy nagyon korlátozott módon. Ennek is 2 alcsoportja van: a Funkcionális nyelvek és a Logikai nyelvek.

Léteznek olyan nyelvek is, amelyek nem sorolhatóak be egyik osztályba sem, nincs egységes jellemzőjük. Általában az imperatív nyelv valamelyik tulajdonságát tagadják.

Karakterkészlet

A programok legkisebb építőelemei a karakterek. A forrásszöveg írásakor alapvető a karakterkészlet ismerete, mivel csak ezek jelenhetnek meg a kódban, és belőlük állítható elő összetett nyelvi elem. Az eljárásorientált nyelvek lexikális, szintaktikai egységek, utasítások, programegységek, fordítási egységek és végül maga a program épül fel a nyelvi elemekből.

Minden nyelv definiálja a saját karakterkészletét, de a legtöbb programnyelv a karaktereket 3 kategóriába sorolja: betűk, számok, egyéb karakterek. A legtöbb nyelvben a betűk az angol ABC 26 betűjét tartalmazzák, viszont az már nyelv függő, hogy értelmezve van-e a nagy és kis betű, vagy különbséget tesz a kettő között. Egyes újabb nyelvek már nemzetiségi betűket is tartalmaznak, ennek következtében "magyarul" is programozhatunk. A számokat nagyjából mindegyik nyelv hasonlóan kezeli, a decimális számjegyeket veszik számokjegy karakternek. Egyéb karakterek közé soroljuk a műveleti jeleket(+, -, *, /), elhatároló jeleket([,], ,, {, }, :,), írásjeleket és speciális karaktereket. A hivatkozási nyelv és az implementáció karakterkészlete egyes esetekben eltérő lehet.

Lexikális egységek

A lexikális egységek azok, amelyeket a fordítóprogram felismer, és tokenizál. Lehetnek többkarakteres szimbólumok, szimbolikus nevek, címkék, megjegyzések, literálok.

Többkarakteres szimbólumoknak a nyelv tulajdonít jelentést, gyakran operátorok vagy elhatárolók, mint C-ben a ++, --, &&.

Szimbolikus neveket 3 csoportra bontjuk, az azonosítókra, a kulcsszóra és a standard azonosítóra. Az azonosító betűvel kezdődik és betűvel vagy számmal folytatódik, segítségével programozói eszközöket nevezünk meg, és ezzel a névvel tudunk rájuk hivatkozni a programon belül. A kulcsszónak az adott nyelv tulajdonít jelentést, ez nem változtatható meg. C-ben ilyenek az if, for, case, break. A standard azonosító pedig egy kicsit mind a kettő, mivel ezt is az adott nyelv látja el jelentéssel, de a programozó megváltoztathatja. C-ben sztenderd azonosítónak számít a NULL.

A címkéknek főleg az eljárásorientált nyelvekben van nagy jelentősége, mivel a segítségével képesek vagyunk megjelölni a végrehajtó utasításokat, így tudunk rájuk hivatkozni a program más részein.

A megjegyzések szerepe az évek során eléggé megkopott, de természetesen minden nyelv biztosít lehetőséget kommentek hozzáadásához. A komment lényegében egy karaktersorozat, amelyet a fordító ignorál. Több implementáció létezik a kommentek alakalmazására, lehet teljes komment sorokat elhelyezni, vagy

sor végi kommenteket, de lehetőség van több soros, egybefüggő kommentek elhelyezésére is. A C-ben mind a 3-ra van lehetőség. (Egysoros komment://, többsoros: /* */)

Végezetül tegyünk említést a literálokról is, mely programozási eszközök segítségével fix értékek építhetők be a programba. A literáloknak van típusuk és értékük. A literálrendszerek egyes programozási nyelvekben eltérő lehet.

Adattípusok

Az adattípus egy absztrakt programozási eszköz, mely konkrét eszközök komponenseként jelenik meg. Mindig rendelkezik egy névvel, ami azonosító. Nem minden nyelv ismeri ezt, azt eszközt, főleg az eljárás-orientált nyelvekben van fontos szerepe. Az adattípust a tartomány, a műveletek és a reprezentáció határozza meg.

A tartomány az adattípus által felvehető értékek halmazát adja meg. Mindegyikhez hozzátartoznak műveletek, amit az adott típussal végre tudunk hajtani. A reprezentáció pedig a beszélő ábrázolást írja le, vagyis a tárban a típus hány bájtira képződik.

Bizonyos nyelvek lehetőséget biztosítanak arra, hogy a programozó saját adattípusokat adjon meg, mindezt általában a beépített típusok segítségével. Meg tudjuk adni a tartományt, a műveleteket, és a reprezentációt is. Viszont ezt nagyon kevés nyelv támogatja.

Az adattípusokat 2 nagy csoportra osztjuk: skalár adattípus és összetett adattípus.

A **skalár** típus tartománya atomi értékeket tartalmaz. Skalár típus a minden nyelvben létező egész, melynek belső ábrázolása fixpontos. Vannak még a valós típusok, ezek reprezentációja lebegőpontos, tartománya implementációfüggő. Ezt a két típust együtt numerikus típusoknak nevezzük, melyeken numerikus és összehasonlító műveletek hajthatók végre.

A karakteres típus elemei karakterek, míg karakterlánc típusú karaktersorozatok. Ezeken szöveges és hasonlító műveleteket hajthatunk végre. Ábrázolásuk karakteres.

Bizonyos nyelvek támogatják a logika típusokat is, melyek értéke igaz vagy hamis lehet, logikai és hasonlító műveleteket tudunk velük végrehajtani.

Érdekességgé meg lehet említeni a felsorolás típust vagy a sorszámozott típust, de ezek kevésbé elterjedtek, csak néhány nyelvben fordulnak elő.

Az **összetett típusok** közül a két legfontosabb a tömb és a rekord. A tömb homogén, statikus összetett típus, meghatározzák a dimenzióinak száma, az indexkészletének típusa és tartománya, és az eleminek a típusa. Léteznek több dimenziós tömbök, melyeket lehet oszlopfolytonosan vagy sorfolytonosan ábrázolni. A tömb elemeit indexek segítségével tudjuk bejárni.

A rekord típus heterogén, tartományában különböző típusok szereplehetnek. Az egyes elemeket mezőknek nevezzük, melyeknek van saját neve és típusa. A mezőneveket minősített nevekké tudjuk ellátni, ezzel segítve az azonos nevű mezők megkülönböztetését.

A **mutató típus** egyszerű típus, de érdekessége az, hogy tartomány tárcímekből áll. Segítségével indirekt módon el tudjuk érni a mutatott elem értékét. Fontos a szerepe az absztrakt adatszerkezetek szétszórta reprezentációjában.

Nevesített konstans

3 komponensből áll: név, típus, érték. Mindig deklarálni kell, mely során megkapja az értékét, és azt nem lehet módosítani a futás folyamán. A névvel tudunk hivatkozni az értékkomponensére. Ennek köszönhetően, elég a deklarációnál módosítani az értéket, és nem kell minden egyes előfordulásnál. C-ben ezt a következőképpen használjuk:

```
# define név literál
```

Ez az előfordítónak szól, és a név minden előfordulását helyettesíti a literállal.

Változó

4 komponensből áll: név, attribútum, cím, érték. A név egy azonosító, a programban a változó mindig a nevével jelenik meg. Attribútumnak olyan jellemzőket nevezünk, amik befolyásolják a változó viselkedést. Attribútum például a változó típusa, mely a deklaráció során rendelődik a változóhoz.

Létezik explicit deklaráció, implicit deklaráció és automatikus deklaráció. Az explicit deklaráció a változó nevéhez rendel attribútumot, míg az implicit betűkhöz. Pl. megoldható, hogy az azonos kezdőbetűvel rendelkező változók azonos attribútumúak lesznek. Az automatikus pedig azt jelenti, hogy a fordítóprogram rendel attribútumot a változóhoz, ha az még nem volt deklarálva.

A változó címkomponense határozza meg a változó helyét a tárban. Egy változó élettartamának a futási idő azon részét nevezzük, ameddig a változó rendelkezik címkomponenssel. A címet lehet statikus vagy dinamikus tárkiosztással rendelni változóhoz. Létezik még programozó által vezérelt tárkiosztás is, a futási időben a programozó rendel címkomponenst a változóhoz.

A változó értékkomponense bitkombinációként jelenik meg a címen, melynek felépítését a típus határozza meg. A változónak értéket értékadó utasítással tudunk adni, mely a C-ben így néz ki:

```
változónév = kifejezés;
```

Alapelemek C-ben

A C típusrendszere 3 részre bontható, vannak az aritmetikai típusok, a származtatott típusok és a void típus. Az aritmetikai típusok közé tartoznak az egészek, a karakter, a felsorolás, a valósok. A származtatott típusok a tömb, a függvény mutató, struktúra, union.

Az aritmetikai típusok egyszerű, míg a származtatottak összetett típusok. Az előbbivel aritmetikai műveletek végezhetőek. Logikai típus nincs, az `int 0` felel meg hamisnak, az összes többi érték igaz értékkel rendelkezik. A logikai műveletek `int 1`-et adnak vissza igaz esetén. A egészek és karakterek előtt használható `unsigned/signed` típusminősítők segítségével tudjuk beállítani vagy letiltani az előjeles ábrázolást. A struktúra fix szerkezetű rekord, ezzel szemben a union, csak változó részt tartalmazó rekord. A void típus tartomány üres, tehát nincs reprezentációja, sem műveletei.

10.2. K&R könyv

Vezérlési szerkezetek

A C nyelvben létező vezérlésátadó utasítások segítségével vagyunk képesek meghatározni a számítások sorrendjét. Lássuk a C leggyakoribb vezérlésátadó utasításait.

Elsőként említsük a **utasításokat és blokkokat**. Az egyes kifejezések akkor válnak utasítássá, ha utánuk rakunk egy pontosvesszőt, ezzel jelezzük az utasítás végét. Képesek vagyunk az utasításokat blokkba foglalni, melyet a `{ }`-jel segítségével tudjuk megtenni. Ezt a jelölést használjuk a függvénye definíciójánál, a `for`, `while`, `if` utasítás által végrehajtott utasítások csoportosításra. Fontos, hogy ha kapcsoszárojelet használunk, akkor utána tilos pontosvesszőt rakni.


```
int a;  
for (a = 0; a < 5; a++)  
{  
    /*utasítások*/  
}
```

Ha valamit el szeretnénk dönteni a programunkban, akkor jön nagyon jól az **if-else** utasítás. A fejlécben mindig megadunk egy kifejezést, mely alapján eldöntjük, hogy egy utasítás végre legyen-e hajtva, vagy sem. Az else ágra nem feltétlenül van szükség, az akkor hajtódik végre, ha nem teljesül az if feltétele. Az if numerikus értéket vizsgál ezért lehetőségünk van rövidíteni a kifejezést. Pl.:

```
if (kifejezés)  
if (kifejezés != 0)
```

A két leírás ekvivalens egymással, az első használata viszont néha nehezen érthetővé teszi a kódot. Az else ággal is lehetnek értelmezéssbeli nehézségek, mivel nem mindig könnyű az egymásba skatulyázott if-ek közül eldönteni, hogy melyikhez tartozik az else ág. C nyelvben ez úgy van megoldva, hogy mindig a hozzá legközelebbi if-hez tartozik. Az esetleges kétértelműségek elkerülése érdekében érdemes használni a kapcsos zárójeleket, melyel egyértelműen meghatározható az utasítás vége. De ha az if/else csak egy utasítást hajt végre, akkor nincs szükség a kapcsos zárójelre, elég a ; használni.

```
if (kifejezés)  
{  
    /*több utasítás*/  
}  
if (kifejezés)  
    utasítás;
```

Az if-else utasítás szorosan összefügg az **else-if utasítás** használatával. Ennek a segítségével több elágazásos if utasítássorozatot hozhatunk létre. Ha az egyik if teljesül a sorozat többi tagja már nem, ha egyik se teljeül, akkor itt is az else ágra kerül a vezérlés, ami akár el is hagyható, ha a maradék esetben semmit sem kell csinálni.

```
if (1.kifejezés)  
    1. utasítás;  
else if (2.kifejezés)  
    2. utasítás;  
else  
    3. utasítás;
```

Az else-if utasítás kiváló arra, hogy pár esetet elkülönítsünk, de mi van akkor, ha több tíz darab eset van. Erre találták ki a **switch** utasítást, amely megvizsgálja, hogy a kifejezés megegyezik-e valamelyik esettel, és ahhoz az esethez továbbítja a vezérlést.

```
switch(kifejezés){  
case '1.lehetőség':  
case '2.lehetőség':  
    1. utasítás;  
case '3.lehetőség':  
case '4.lehetőség':
```

```
case '5.lehetőség':  
    2. utasítás;  
}
```

Amint látod, nem muszály mindegyik esethez külön megadni az utasítást, lehet őket csoportosan megadni, ezzel megkönnyítve a programozó dolgát. A switch magját pedig {} jellel határoljuk. A case-en kívül használhatunk default címkét is, amivel egy alapértelmezett utasítást adhatunk meg, arra az esetre ha egyik eset se teljesülne. Fontos a case-ek között nem fordulhat elő azonos. Ha egy case utasítása végrehajtódik akkor a vezérlés a következőre ugrik. Ez viszont nem mindig szerencsés, ezért break-et kell használni mindegyik esetén, de ha nem is rakunk mindegyikhez, akkor is az utolsó eset után érdemes rakni egy break-et, ezzel biztosítva, hogy a vezérlés nem fog szétesni.

Ha a programozási nyelvek ciklusairól beszélünk van 2 alapvető, amelyek mindegyikben megjelenik, így C-ben is, ezek a **for** és a **while** utasítások. Arra használjuk őket, hogy bizonyos utasításokat ismételjünk, egészen addig, ameddig a fejrészükben lévő kifejezés teljesül.

```
while(kifejezés)  
    utasítás
```

A while így néz ki szintaktikailag, egészen addig hajtja végre az utasítást, ameddig a kifejezés nem nulla. Itt egy kifejezés van, ezzel szemben a for ciklus nyelvtanilag 3 kifejezésből áll.

```
for(kif1;kif2;kif3)  
    utasítás  
  
/*ennek a while átírata*/  
  
kif1;  
while (kif2)  
    utasítás  
    kif3
```

Amint látod, a két fajta ciklus átírható egymásba, de bizonyos esetekben az egyik könnyebben használható mint a másik. A for ciklusnál nem muszály mindegyik kifejezést kiírni, sőt el is hagyhatóak, ekkor végtelen ciklushoz jutunk, amelyből vagy break, vagy return utasítással tudunk kiugrani. A fentebbi példában láthattad, hogy a for kifejezései közé ; kellett rakni, de ez nem feltétlenül kötelező, csak akkor más értelemmel rendelkezik.

```
int i, j;  
for (i = 0, j = 10; i!=j; ++i, --j)  
    printf("Még nem egyenlő");
```

Ilyen esetben az i-t és az j-t is léptejük egyszerre. Az i és a j definiálása és az értékük változtatása is egy-egy kifejezésnek felel meg.

A for és a while ciklusok a kiugrási feltétel teljesülését a ciklus elején vizsgálja, de bizonyos esetekben jól jöhet a **do-while** utasítás. Lényege az, hogy legalább 1-szer biztos, hogy belépünk a ciklusba, ezután pedig a gép kiértékeli a kifejezést.

```
do  
    utasítás  
while (kifejezés)
```

Már többször szó esett a ciklusokból való kilépés módjáról, ezt valósítja meg a **break** utasítás. Az utasítás hatására a vezérlés kiugrik a legbelsőbb ciklusból még az előtt, hogy a kifejezést kiértékeljük volna az újabb iterációban.

Ha van olyan utasítás, ami megállítja a ciklust, akkor lennie kell olyannak a folytatja azt. Na nem pont ilyen értelemben, de hasonlót csinál a **continue** utasítás. Ez azt jelenti, hogy a vezérlés visszatér a ciklus fejéhez, tehát abban az iterációban nem csinálunk semmit, csak továbblépünk. Ezt általában valamilyen feltételhez szokták kötni.

```
int i
for (i = 0; i < 20; ++i)
{
    if (i%2 != 0) continue;
    printf("%d", i);
}
```

Ebben az esetben csak a páros számokat írjuk ki, a többi számot kihagyjuk.

A függvények esetén a break-hez hasonló a **return** utasítás. Ez a függvény hívójához tér vissza, általában valamilyen értékkel, de előfordulhat, például void típusú függvények esetén, hogy nem ad vissza semmilyen értéket.

```
return;
return kifejezés;
```

Ahogy említettem a break utasítással mindig csak a legbelső ciklusból tudunk kilépni, de mi van akkor, ha mi az egymásba skatulyázott ciklusokból, ekkor kerül előtérbe a **goto** utasítás. Szintaxisát tekintve:

```
for (...)
    for (...)
        if (kifejezés)
            goto megoldás;

megoldás:
    tedd hamissá a kifejezést
```

Tehát egy címkére ugunk, amit a goto utasítás után megnevezünk, és a címkének megadjuk, hogy milyen utasítást hajtson végre. A goto használata viszont abszolút nélkülözhető, és használata nem is javasolt.

10.3. BME C++ könyv 1-16. oldal feldolgozás

C++ nem objektum-orientált újdonságai

A C++ lényegében a C programozási nyelv objektum-orientált változata, melynek első változata 1983-ban jelent meg. A nyelv atyjának Bjarne Stroustrup-ot tekintjük, aki AT&T Bell Laboratories-nak dolgozott. Első verzió neve C with class, ezzel utalva az objektum-orientáltságra. A C++ 1998-ban lett szabványosítva, azóta szabványt is kiadtak, a 2003-ast és a 2011-est. A C++ programok érdekessége, hogy többségük lefordul a C fordítókkal, sőt kezdetben az első C++ fordítók C kódot generáltak. Tehát minden C program C++ program, de nem minden C++ program C program.

Ebben a fejezetben pont azokat a különbségeket fogjuk sorra venni, amik C++-ban használhatóak, de a C-ben nem. Itt olyan funkciókra kell gondolni, amik segítettek kijavítani a C nyelv esetleges gyengeségeit.

Az első lényegi különbség a függvényparaméterekben és a visszatérési értékben rejlik. A C-ben, ha nem írtunk paramétert, akkor korlátlan számú paramétert használhattunk, ezzel szemben a C++-ban ez annyit jelent, hogy nincs paraméter, vagyis mintha a paraméternek `void` típust adtunk volna meg. Ahhoz, hogy a korlátlan számú paramétermegadást engedélyezzük C++-ban, a paraméterlistéba `...`-t kell beírni.

```
//C-ben
void f()
{
}

//C++-ban
void f(...)
```

Egy másik különbség, hogy a C nyelvben kihagyhatjuk a visszatérési érték típusának meghatározását, azt `int`-nek értelmezi. Ezzel szemben C++-ban fordítási hibát kapunk, nem szabad kihagyni a visszatérési érték típusának meghatározását.

```
void f(void); //nincs hiba
g(void); //fordítási hiba
```

A másik különbség a `main` függvényben keresendő. Az argumentum listát hagyhatjuk üresen, vagy használhatjuk az `argc`, `argv[]` argumentumokat, mellyel a parancssori argumentumok számát és tömbjét adja át a `main`-nek. Ez eddig hasiőonló a C-hez, de a `main` végére nem szükséges oda írni a `return`-t, azt a fordító automatikusan hozzáilleszti a kódhoz.

Sokáig a C++ nyelv sajátja volt a `bool` típus, mellyel logikai változókat deklarálhatunk, értéke `true` és `false` lehet. Ez máát a C-ben is elérhető, de előtte csak az `int` vagy az `enum` típusokkal fejezhettük ezt ki. `int` esetén 0 számít hamisnak és minden más érték igaz.

A C++-ban megjelet a `wchar_t`, mint beépített típus, mely segítségével Unicode karakter stringeket tudunk reprezentálni. Ez a C-ben is használható, viszont ehhez `#include`-olni kellett a bizonyos header fájlokat. Az ilyen típusú változókat a következőképpen definiálhatjuk:

```
wchar_t = L 's';
wchar_t* = L "sss";
```

C++-ban a változókat olyan helyeken is deklarálhatjuk, ahova utasításokat is írhatunk. Ilyen például, hogy a `for` ciklus-ban tudjuk definiálni az indexet, nem kell azt előtte deklarálni. Ettől függetlenül elmondható, hogy a változókat érdemes pont ott deklarálni, ahol a szeretnénk használni, ezzel áttekinthetőbb kódot kapunk. A változó csak a deklarációja után érhető el. A `for` cilus fejlécében deklarált változókat nem tudjuk használni a cikluson kívül.

A C nyelvben nem tudunk azonos nevű függvényket megadni, mivel azonosításként a linker csak a függvény nevét használta. Ennek következtében a programozóknak rengetegszer kellett erőltett függvényneveket kitalálni. Ezt a problémát oldja meg a C++ függvény túlterhelés funkciója, mely annyit tesz, hogy a linker a függvényt a neve és az argumentum listája alapján azonosítja. Tehát létre tudunk hozni azonos nevű függvényeket, ha azok argumentumlistája eltér. Ezt a technikát név felderítésnek nevezzük. Mivel ez a C-ben nem értelmezett ez a funkció, kitaláltak egy módszert arra, hogy a C és C++ közötti kompatibilitást

megoldjuk, ehhez a függvény deklaráció elé kell írni, hogy `extern int`. Ezzel letiltjuk a név felderítő mechanizmust.

A függvényekkel kapcsolatos másik különbség, hogy a C++ kódban lehetőségünk van alapértelmezett értéket adni az argumentumoknak. Ezzel ki tudjuk küszöbölni azt, hogyha nem adunk meg elég argumentumot a függvényhívásnál.

```
int f( int a, int b = 100, int c = 200){
    return a*b-c;
}
int main()
{
    int egyik = 10;
    int masik = 20;
    int harmadik = 30;
    f(egyik, masik, harmadik); //érték = 10*20*30
    f(egyik) // érték = 10*100*200
    f() // hiba: nincs elég argumentum
}
```

A C++ egyik legnagyobb újdonsága pedig a referencia típus bevezetése. A C-ből tudjuk, hogy ha egy függvénynek paraméterül adunk egy változót, akkor a azt lemásolja a függvény, és nem módosítja a változó értékét. Ha azt szeretnénk, hogy a függvény módosítson az értéken, ahhoz kell használni a pointert, ami a változó memóriacímére mutat. Ennek segítségével közvetlenül tudjuk átadni a függvénynek a változót, nem másoljuk az értékét. Ennek használata elég nehezen érthetővé teszi a kódot, és rengeteg hibázási lehetőséget rejt magában. Hogy ezt orvosolják, bevezették a referencia típust. A referncia típusú változót `&` jellel jelöljük. Ez a jel a C-ben egyoperandusú operátor, ami a változó memóriacímét adja vissza. A pointer és a referencia közötti különbség:

```
int x = 5;
int* a = &x;
int& b = x;
```

Míg a pointernek egy memóriacímet kell értékül adni, addig a refernciánál csak meg kell adni, hogy melyik változóra referáljon. Ezután mindegy, hogy a b-t vagy az x-et módosítjuk, mind a kettőn végrehajtódik a változtatás. Érdekesség még, hogy a referencia és a változó memóriacíme megegyezik, tehát a referencia nem foglal a memóriában területet, ellentétben a mutatóval. A programokban nagyon ritkán használjuk a referenciákat a fentebb látott módon, mivel nehéz lenne olvasni egy olyan kódot, ahol ugyan arra 2 változó is hivatkozik. Ezért ennek a fő használati értéke a cím szerinti paraméterátadásban rejlik, ezt szokták referencia szerinti paraméterátadás. A cím szerinti paraméterátadást általában akkor szoktuk használni, ha változtatni akarunk az értéken, ezért a refenciának csak olyan változót adhatunk értékül, ami módosítható. (Létezik konstans típusú referncia is, de ez speciális.) Egy fontos dolog még ebben a témakörben az, hogy lehet-e egy függvény visszatérési értéke pointer vagy referncia. A válasz igen, lehet, de csak korlátozott esetekben, főleg cím szerinti paraméterátadás esetén. Összességében elmondható, hogy nem szabad visszaadni pointert és refernciát lokális változókra, vagy érték szerinti paraméterekre, mivel előfordulhat, hogy érvénytelen memóriacímre fognak hivatkozni.

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.

DRAFT

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

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