Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

COLLABORATORS			
	TITLE : Univerzális programozás		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Bátfai, Norbert Ã©s Nagy, László Mihály	2019. szeptember 22.	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai
0.0.5	2019-03	2. és 3. fejezet kitöltése	nlm
0.0.6	2019-04	4., 5., 6. fejezet kitöltése	nlm
0.0.7	2019-05	7., 8., 9. fejezet kitöltése	nlm

Ajánlás

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

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

Tartalomjegyzék

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása: /bhax_sources/fejezet2/vegtelen_ciklus_1szal.c; /bhax_sources/fejezet2/vegtelen_ciklus_soksz
/bhax_sources/fejezet2/vegtelen_ciklus_noload.c

Látható, hogy alapból a leforduló C programok 1 szálon futnak. Amikor végtelen ciklust indítok, azonnal gőzerővel elkezdte darálni az ütemező által épp szabadnak ítélt magot. Az OpenMP könyvtár segítségével megoldható az is viszont, hogy több szálat terheljen a vezérlés. Ekkor egyszerre minden elérhető magot terhel a ciklus. A ciklus extrém processzorterhelése megelőzhető a polling lelassításával, pl. a példakódban használt sleep(1) segítségével. Ez az adott szálat kisorolja az átadott időtartamig.

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

```
Program T100
{

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

  main(Input Q)
```

```
  {
    Lefagy(Q)
  }
}
```

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{

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

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

  main(Input Q)
  {
    Lefagy2(Q)
  }

}
```

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

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

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

Ez a pszeudokód Turing megállási problémáját mutatja be. Ez példázza hogy egy Turing gép sosem lesz képes megmondani, hogy egy másik Turing gép megáll-e vagy végtelen ciklusba kerül, mert logikai ellentmondásba kerülne menet közben. Ha egy ilyen gépnek átadjuk a saját kódját, azt fogja mondani hogy nem áll meg, de ez nem lehetséges, hisz azzal hogy ezt megmondta, megállt.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: /bhax_sources/fejezet2/valtozocsere.c

A cserélendő változók ábrázolása lehetővé teszi ezt a fajta cserét. Az aritmetikai rész nem igényel további dokumentációt, sima matek, ld. kommentek a kódban.

2.4. Labdapattogás

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

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

Megoldás forrása: /bhax_sources/fejezet2/labda_iffel.c; /bhax_sources/fejezet2/labda_ifnelkul.c

Az eredeti forrás egyszerű: boundary checkinget hajtunk végre minden draw után. Az ncurses segítségével felvesszük a konzolablak teljes méretét, lehelyezünk egy O-t egy koordinátára, várunk, mozgatjuk az O-t, boundary checkelünk ifekkel, majd minden megy előről. A feltételek megnézik, hogy balról, jobbról, felülről és alulról elérte-e már a "labda" az ablak szélét, és ez alapján negálja a mozgatásra használt "vektorok" irányát.

Az if mentes módszer a C-ben való osztás tulajdonságát abuse-olja. Megnézi hogy milyen távolságra van a két széltől egy adott irányban (x vagy y), de mivel ez egészt ad vissza, így mindig 0 vagy 1 lesz az eredmény. A két részt összeadva (a labda előtti rész, és a labda utáni rész) megmutatja, hogy ha elértük az egyik szélt. 1 értéket vesz fel, ha szélen vagyunk, 0-t ha bárhol máshol. -1-et ezzel hatványra emelve megkapjuk, hogy mennyivel szorozzuk meg a mozgatóvektort. -1 a nulladikon az 1, így a mozgatóvektor 1-gyel lesz szorozva, azaz nem változik. -1 az elsőn viszont már -1, így a mozgatóvektort ezzel szorozva a negáltjára fordul, a labda mozgásának az iránya megfordul. Így már működik.

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 forrása: /bhax_sources/fejezet2/szohossz.c

Ahogy a kód alapján is látható, lesz két változónk: egy hossz nevű, amiben számoljuk a biteket, és egy pivot nevű, amiben egy ún. jelzőbitet helyezünk el, a legalacsonyabb helyiértéken (hex: 0x1 == dec: 1). Az utótesztelős (do-while) ciklus először leszámolja a pivot elemet, ami létezik, tehát 1 bitet már biztos tárol az int - majd loopba lép.

A ciklusfej a BogoMIPS-ben hivatkozott részen alapul. A trükk a <<= operátorban rejlik. Amit csinál, igazából nem is olyan bonyolult: balra bitshifteli a változó tartalmát, jelen esetben 1-gyel (ld. a kód). Így a jelzőbit fokozatosan "le fog csúszni" balra. Eközben egyre nagyobb értéket vesz fel a módosított változó - de mivel C-ben a 0 hamist, bármi más pedig igazt jelöl, a ciklus tovább robog. Amint "leesik" a jelzőbit a bal oldalon viszont, a változó értéke 0 lesz, és a ciklus terminál. Így le tudjuk számolni a szóhosszt, hisz az int 1 szó hosszú.

2.6. Helló, Google!

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

Megoldás forrása: /bhax_sources/fejezet2/pagerank.c

A PageRank algoritmus 1:1 implementálása a diasori példa alapján. 4 oldal egymásra mutató linkjeinek száma alapján (kifelé mutató linkek száma, saját magára mutató linkek száma) alapján súly lesz az adott oldalhoz linkelve, mátrixműveletek segítségével.

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

Először lássuk az előre elkészített kódot!

```
library(matlab)

stp <- function(x){

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

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

Vegyük sorra mi mit csinál! Először is létrehozunk egy stp nevű függvényt, mely egy tetszőleges x számig kiszámolja a Brun-konstans értékét. Ez a következőképp történik:

- eltárolja a prímeket x-ig
- minden prímet kivon a rákövetkezőből, és ezeket a különbségeket eltárolja
- minden olyan esetnek ahol a különbség 2 volt, kimenti az indexét
- ezzel előállítja a t1primes és t2primes vektort, amiben az ikerprímeket helyezi el, az idx indexű prímek és az idx indexű prímek + 2 (ami mindig egy másik prím lesz) segítségével
- képez egy vektort felhasználva az előbb említett 2 vektort, azokra reciprok és páronkénti összeadást alkalmazva
- visszaadja az összegét ennek a vektornak

Ezt követően arbitrary steppinggel megáll pár x értéknél, és kiszámolja a Brun-konstans értékét addig a pontig, majd plotolja. Látható hogy az értékek valóban konstans értékhez tartanak.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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

A Monty Hall probléma rendkívül híres, a valószínűségszámítás sajátosságait jól demózó probléma. Az alapprobléma a következő. van 3 ajtó, az egyik mögött pedig a főnyeremény (a többi mögött meg egy-egy kecske). Elsőkörben választunk egy ajtót, ami mögött valószínűnek érezzük a főnyereményt. Ekkor a helyes döntés esélye 1/3. Ezután a játékvezető kinyit egy (1) ajtót. Amennyiben nem a főnyeremény volt mögötte, dönthetünk: a megmaradt két ajtó közül maradunk annál, amit az első körben kiválasztottunk, vagy cserélünk a másikra.

A helyes döntés mindig cserélni; ekkor kétszer nagyobb esélyünk lesz a nyerésre. A gondolatmenet a következő: mikor először választunk ajtót, 1/3 esély van arra, hogy mögötte lesz a főnyeremény, 2/3 arra hogy a másik két ajtó közül valamelyik mögött. Amikor a játékvezető kinyit azok közül egyet, és nem a főnyeremény volt mögötte, az esélye 0%-ra nő, míg a másiké 100%-ra, azaz a két ajtóra eső valószínűség (2/3) 100%-át (egészét) birtokolja az az ajtó. Így tehát annak a másik nyitott ajtónak 2/3 eséllyel lehet a főnyeremény mögötte, míg az általunk választottnak továbbra is 1/3-dal, tehát érdemesebb arra váltani.

Statisztikai szimulációval ezt könnyen be lehet látni, ld. a hivatkozott kód.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

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

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

```
1.    S →  a B C
2.    S →  a S B C
3.  C B →  C Z
4.  C Z →  W Z
5.  W Z →  W C
6.  W C →  B C
7.  a B →  a b
8.  b B →  b b
9.  b C →  b c
10. c C →  c c
```

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.

A C nyelvű utasítások BNF (Backus-Naur-forma) szintaxissal leírva, forrása a Western Michigan University honlapja (https://bit.ly/2u2urLs):

```
<statement> ::= <labeled-statement>
              | <expression-statement>
              | <compound-statement>
              | <selection-statement>
              | <iteration-statement>
              | <jump-statement>

<labeled-statement> ::= <identifier> : <statement>
                      | case <constant-expression> : <statement>
                      | default : <statement>

<expression-statement> ::= {<expression>}? ;

<selection-statement> ::= if ( <expression> ) <statement>
                        | if ( <expression> ) <statement> else <statement>
                        | switch ( <expression> ) <statement>

<iteration-statement> ::= while ( <expression> ) <statement>
                        | do <statement> while ( <expression> ) ;
                        | for ( {<expression>}? ; {<expression>}? ; {< ↩
                           expression>}? ) <statement>

<jump-statement> ::= goto <identifier> ;
                   | continue ;
                   | break ;
                   | return {<expression>}? ;
```

A Backus-Naur-forma igazából a formális nyelveknél a környezetfüggetlen nyelvek leírására használt jelölés gépi feldolgozásra kidolgozott változata, működésben teljesen megegyezik azzal:

```
A -> AA | b
```

Ez egy általános és elterjedt jelölés, mely a generatív grammatikák leírására használatos; **A** egy nemterminális szimbólum, **b** egy terminális szimbólum, **->** a leképezés operátora, | a logikai xor művelete operátora. Ez a grammatika például egy **A**-t vagy egy **AA**-ra, vagy egy **b**-re cserél, és addig nem áll meg, amíg minden **A**-ból **b** nem lesz (terminál). Magyarul tetszőleges számú **b** generálására használatos.

Lássuk akkor a C89 vs. C99 példát!

```c
int main(void)
{
    /* nem lehet scope-ra vonatkozó lokális változót létrehozni */

    int i;
    for (i = 0; i < 10; i++);
    return 0;
}
```

Ez a kód, egy, a C89 szabványnak megfelelő code snippet. Két jellegzetességet is érdemes megfigyelni:

- // stílusú kommentek nem megengedettek, csak és kizárólag /* */ használata lehetséges

- nem lehet a ciklus scope-jának lifetime-jára vonatkozóan változót deklarálni, kötelező előtte

```
int main(void)
{
    // itt már lehet

    for (int i = 0; i < 10; i++);
    return 0;
}
```

C99-ben ezek egyike sem okoz már gondot.

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 forrása: a BHAX "Így neveld a programozód!" c. streamje:

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

A lexer megírásához a *nix rendszereken elérhető **lex** (aka. **flex**) tool-t használjuk. Ez az eszköz, amint az feljebb látható, egy saját szintaxist alkalmazó fájl alapján C forráskódot állít elő, mely garantáltan biztonságosan fogja feldolgozni az átadot bemenetet.

Minden **lex** forráskód 3 részből áll, melyeket **%%** választ el. Az első rész "vezérléssel kapcsolatos" definíciókat tartalmaz, a második magukat a matchelendő tokeneket és a match során végrehajtandó utasításokat, a harmadik pedig C kódot tartalmaz, ami az előállított C forrásban 1:1 megjelenik.

Az első szekcióban láthatjuk azonban, hogy **%{** és **%}** szimbólumok közt C kód szerepel. Ebben a szekcióban az ilyen szimbólumokkal körülhatárolt rész szintén C kód beillesztésére használatos, 1:1 másolásra kerül a kimeneti C forrásba.

Amint látható, amellett hogy behúzzuk a **stdio.h** headert, létrehozunk egy **realnumbers** változót is, melyben a megtalált valós számok darabszámát fogjuk eltárolni. Továbbá láthatjuk, hogy a C nyelvű részen kívül definiálásra kerül egy **digit** nevű placeholder, mely a **[0-9]** regex tokent fogja jelképezni.

A második szakasz **regex { <C utasítások> }** formátumban tokeneket definiál amikre matchelünk, majd utasításokat amiket a megtalálásukkor végre kell hajtani. Itt látható, hogy mintailleszkedés esetén növeljük a **realnumbers** változó értékét, illetve kiírjuk a szövegrészletet ami matchelt (ld. a **yytext** fenntartott változó), valamint magát a számot (az **atof()**, azaz alpha-to-float függvény segítségével).

A harmadik szakasz tiszta C kódot tartalmaz, ami az Áttekintésben leírtak mentén 1:1 belekerül a kimeneti C forrásba. Érdemes megfigyelni a **yylex()** hívást, az indítja el és hajtja végre az elemzést.

A **lex** forrást a **lex sample.lex** paranccsal tudjuk C kóddá alakítani (ahol a **sample.lex** a lex forrásfájl neve/elérési útja). Ezt követően létrejön egy **lex.yy.c** nevű C forrásfájl jön létre, amit innentől kezdve hagyományosan (pl. **gcc lex.yy.c -o sample_lexer**) lehet fordítani. A szöveget **STDIN** streamen keresztül kapja, a kimenetet a **STDOUT** streamre teszi.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás forrása: hasonlóan az előző kódhoz, ez is a BHAX "Így neveld a programozód!" c. streamjéből származik:

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

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

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

  {'a', {"4", "4", "@", "/-\\"}},
  {'b', {"b", "8", "|3", "|}"}},
  {'c', {"c", "(", "<", "{"}},
  {'d', {"d", "|)", "|]", "|}"}},
  {'e', {"3", "3", "3", "3"}},
  {'f', {"f", "|=", "ph", "|#"}},
  {'g', {"g", "6", "[", "[+"}},
  {'h', {"h", "4", "|-|", "[-]"}},
  {'i', {"1", "1", "|", "!"}},
  {'j', {"j", "7", "_|", "_/"}},
  {'k', {"k", "|<", "1<", "|{"}},
  {'l', {"l", "1", "|", "|_"}},
  {'m', {"m", "44", "(V)", "|\\/|"}},
```

```
  {'n', {"n", "|\\|", "/\\/", "/V"}},
  {'o', {"0", "0", "()", "[]"}},
  {'p', {"p", "/o", "|D", "|o"}},
  {'q', {"q", "9", "O_", "(,)"}},
  {'r', {"r", "12", "12", "|2"}},
  {'s', {"s", "5", "$", "$"}},
  {'t', {"t", "7", "7", "'|'"}},
  {'u', {"u", "|_|", "(_)", "[_]"}},
  {'v', {"v", "\\/", "\\/", "\\/"}},
  {'w', {"w", "VV", "\\/\\/", "(/\\)"}},
  {'x', {"x", "%", ")(", ")("}},
  {'y', {"y", "", "", ""}},
  {'z', {"z", "2", "7_", ">_"}},

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

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

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

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

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

        found = 1;
        break;
      }
```

```
    }

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

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

Az alapelv hasonló az előzőhöz, a kivitelezés azonban eltér. A kimeneti program egy tetszőleges bemeneti szövegből leetspeak stílusú szöveget készít (https://simple.wikipedia.org/wiki/Leet).

Az importok mellett definiáljuk az ún. **leetsize** (stilizáltan: **L337SIZE**) változót mely a lefedett karakterek számát jelöli, illetve egy struktúrát **leetdict** néven (stilizáltan: **l337d1c7**), mely minden egyes karakteréhez 4 lehetséges csere-opciót társít (tárol).

Más taktikát alkalmazunk mint az előzőnél: itt minden egyes karakterre matchelünk (**.**). Amint matcheltünk egy karakterre, elkezdünk végigiterálni a **leetdict**-en, keresve az adott karakter után, hogy le van-e fedve. Ha megtaláljuk, véletlenszerűen kiírjuk helyetete a hozzátartozó karakternégyes egyik tagját, majd break. Ha nem kiírjuk az eredeti karaktert.

A harmadik szakaszban az előző feladathoz hasonlóan rendkívül triviális dolgok történnek, mindössze annyiban több az előzőnél, hogy az elején rögzítjük a random függvényhez a seedet az **srand(time(NULL) + getpid())** hívással.

3.6. A források olvasása

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

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

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

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

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

Meglehetősen érdekesen viselkedik. Mindig sikeres a feltétel alapból, amikor nem próbálunk szignált átadni. Amint mégis megpróbálunk, sikeresen el is utasítja, de egy idő után már nem írja ki a jelkezelo()-ben megadott stringet. Ezt nem vizsgáltam tovább.

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

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

A két ciklus működésre megegyezik.

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

A tomb tömb i-edik eleme i-vel lesz egyenlő, majd az i értéke megnövelődik.

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

Meglepő módon lefordul! A működése számomra rejtély viszont, hisz az assigmentnek nem hiszem hogy kellene lennie logikai értékének.

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

Meghívja az f() fv-t, a-val és a+1-gyel, valamint a változók megcserélésével is, és kiírja az eredményt.

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

Kiírja az f(a) visszatérési értékét és az a értékét.

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

Az f() referenciát kap, ha úgy van megírva működik szépen és kiírásra kerül a visszatérési érték.

Ez a feladat a szignálok kezeléséről szól. Lássuk a kiinduló példakódot:

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

void jelkezelo()
{
  printf("ugh\n");
}

int main(void)
{
  for(;;)
  {
    if(signal(SIGINT, jelkezelo)==SIG_IGN)
```

```
        signal(SIGINT, SIG_IGN);
    }
    return 0;
}
```

Láthatjuk, hogy a **main**()-ben egy végtelen ciklus pörög. A benne lévő feltétel megnézi, hogy a **SIGINT**-ben lévő szignál állapota mi, miután meghívódott a **jelkezelo**(). Amennyiben **SIG_IGN** (azaz szignál ignorálása) a visszaadott érték, ismét beállítja hogy ignorálva legyen. Ebből az szűrhető le, hogy alapból ignorálásra kerülnek a szignálok, mi tudjuk befolyásolni őket, hogy ne legyenek (ld. **man 2 signal**).

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

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

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

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

- tetszőleges x számhoz létezik olyan y hogy nagyobb nála és prím (~ a prímek végtelenségig nőnek)
- nem értettem mi az az SS
- minden x prímnél van egy nagyobb y szám
- minden nem prímnél van kisebb y

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);
  ```

```
int main(void)
{
  int i; // egy egész
  int *mutat_i = &i; // egészre mutató mutató
  // referenciát passz
  int egesztomb[10]; // egész tömb
  // referenciát passz
  int *mutatotomb[10]; // egészre mutató mutatók tömbje

  int* pelda1(int &ref)
  {
    return ref;
  }


}
```

4. fejezet

Helló, Caesar!

4.1. int *** háromszögmátrix

Megoldás forrása: refaktorált sourceforge-os:

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

int main ()
{
    int nr = 5;
    double **tm;

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

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

    }

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

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

```
    }

    tm[3][0] = 42.0;
    (*(tm + 3))[1] = 43.0;  // mi van, ha itt hiányzik a külső ()
    *(tm[3] + 2) = 44.0;
    *(*(tm + 3) + 3) = 45.0;

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

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

    free (tm);

    return 0;
}
```

Mit csinál a kód, mi a lényeg? A lényeg a pointerek és a tömbök hasonlósága a C nyelvben. A tömbelemekre hivatkozás operátora [] használható pointer dereferenciára is.

Az elején lefoglalunk egy 5 double * méretű területet és a tm-hez rendeljük. Ez kb. ugyanaz, mintha lenne egy double *tm[5]-ünk. Ezt követően minden ilyen elemhez rendelünk egy egyre növekvő méretű double tömböt. Így lesz belőle háromszögmátrix.

Az ezt követő dupla ciklus bejárja a háromszögmátrix elemeit, és értéket rendel hozzájuk, majd egy másik ugyanilyen ciklusparos kiírja ezeket az értékeket.

Ezután megkíséreljük felüldefiniálni a háromszögmátrixban tárolt értékeket. Mennyünk végig soronként!

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

A 4. sor 1. elemét 42-re állítja.

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

A 4. sor 2. elemét 43-ra állítja. A (*(tm + 3)) dereferrálja a base + 3. elemét a tm-nek, azaz a 4. sort. Innentől pedig megy tovább a téma ugyanúgy. Ha nem lenne zárójel akkor a 4. oszlop lenne kiválasztva, és mivel az már nem tömb, annak a [1] eleme nem létezik, és elszállna.

```
*(tm[3] + 2) = 44.0;
```

Először kiválasztja a 4. sort, majd 2 offsettel a 3. oszlopot derefferálja. Magyarul tehát a 4. sor 3. oszlopát állítja 44-re.

```
*(*(tm + 3) + 3) = 45.0;
```

Itt már tisztán pointer aritmetikával dereferráljuk a 4. sor 4. oszlopának értékét és 45-re állítjuk.

Ezt követően végigmegy megint két ciklussal a program a háromszögmátrixon, kiírja minden elemét. Majd egy hasonló ciklus felszabadítja az összes benne lévő elemet, majd cikluson kívül kézzel mi magunk felszabadítjuk a tm változót is.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás forrása: sourceforge-os, refaktorálva:

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

#define MAX_KULCS 100
#define BUFFER_MERET 256

int main(int argc, char **argv)
{
    char kulcs[MAX_KULCS];
    char buffer[BUFFER_MERET];

    int kulcs_index = 0;
    int olvasott_bajtok = 0;

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

    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);
    }
}
```

Majd folytassuk a titkosítandó szöveggel és a kulccsal! (56789012)

```
A processz és a szál olyan absztrakciók, amelyeket egy program teremt meg ↩
   számunkra, az ope
rációs rendszer, azaz a kernel. A konkrétabb tárgyalás kedvéért gondoljunk ↩
   most egy saját C
 programunkra! Ha papíron, vagy a monitoron egy szerkesztőben nézegetjük a ↩
    forrását, akkor
valami élettelen dolgot vizsgálunk, amelyben látunk lexikális és ↩
   szintaktikai egységeket, u
```

```
tasításokat, blokkokat, függvényeket; nem túl érdekes. Ha lefordítjuk és ↩
   futtatjuk, akkor v
iszont már valami élő dolgot vizsgálhatunk, ez a processz, amely valahol ↩
   ott van a tárban.
Ennek a tárterületnek az elején a program környezete, a parancssor ↩
   argumentumai, a lokális
változóterülete és a paraméterátadás bonyolítására szolgáló veremterüle ↩
   található, amelyet
a dinamikusan foglalható területe, a halom követ. Majd jön az inicializált ↩
   globális és stat
ikus változóit hordozó adat szegmens és az iniciálatlan BSS. Végül jön a ↩
   kódszegmense, majd
 a konstansai. Ennek a tárterületnek a kernelben is van egy vetülete, ez a
PCB.hogy
```

A működése vihar egyszerű. A kulcsot elmenti, majd a bemenet karakterein egyenként végighaladva és a kulcs betűin egyenként körbemenve mindegyik betű párt össze xor-ozza (egy betű a bemenetről, egy betű a kulcsból), és visszaírja a kimenetre.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

forrás: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html#exor_titkosito

```
public class ExorTitkosító {
    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) {
            for (int i = 0; i < olvasottBájtok; ++i) {
                buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);
                kulcsIndex = (kulcsIndex+1) % kulcs.length;
            }

            kimenőCsatorna.write(buffer, 0, olvasottBájtok);
        }
    }

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

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

Tanulságok, tapasztalatok, magyarázat...

4.4. C EXOR törő

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

Megoldás forrása: a tankönyvtáras könyv, refaktolálva:

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

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

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

    return (double) titkos_meret / sz;
}

int tiszta_lehet(const char *titkos, int titkos_meret)
{
    // 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");

}
```

```
void exor(const char kulcs[], int kulcs_meret, char titkos[], int  ↩
   titkos_meret)
{
    int kulcs_index = 0;

    for (int i = 0; i < titkos_meret; ++i)
    {
        titkos[i] = titkos[i] ^ kulcs[kulcs_index];
        kulcs_index = (kulcs_index + 1) % kulcs_meret;
    }

}

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

    return tiszta_lehet(titkos, titkos_meret);
}

int main(void)
{
    char kulcs[KULCS_MERET];
    char titkos[MAX_TITKOS];
    char *p = titkos;
    int olvasott_bajtok;

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

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

    // osszes kulcs eloallitasa, karakterenként végigpörög
    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, igy nem kell egy masodik buffer
        exor(kulcs, KULCS_MERET, titkos, p - titkos);
    }

    return 0;
}
```

-O3 kapcsolóval a tankönyvi forrás szerint 5 perc alatt végez, bár ez az egyszálas implementáció. A működése egyszerű. Megpróbál minden 8 jegyű számkombinációt, újraexorozza a titkos szöveget vele, majd ha az átlagos szóhossz 6 és 9 karakter között van, valamint a supposed cleartexxt tartalmazza a hogy, az, ha és nem szavakat, megjelöli lehetséges kulcsként az adott kombinációt. Ezeket és a tisztaszöveget kiíratja stdout-ra.

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

Gyakorlatilag ezeknek a kapuknak az "analóg"-szerű megvalósítása. Egy neurális hálóba behuzalozásra kerül az egyes művelet, és ezeket tetszőleges a1 és a2 próbaadatokon végrehajtja.

4.6. Hiba-visszaterjesztéses perceptron

Megoldás forrása:

Megnéztem a forrást, de nem sikerült felélesztenem. Még majd foglalkozok vele, a perceptronokat már nagyjából értem mik.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás forrása: `/book_sources/fejezet5/mandelbrot.cpp`

Van egy x $\in$ [-2; 2] y $\in$ [-2i; 2i] komplex síkunk. Erre mi "ráterítünk" egy tetszőleges felbontású (pl. 800x800-as) rácsot (a kimeneti kép pixeleit). Minden egyes pixelhez tartozó pontban kiértékelünk egy speciális függvényt iteratívan (ld. később), majd ha egy bizonyos, általunk paraméterezhető határszámig iterálva a fv. érték még mindig része a Mandelbrot-halmaznak, a pixelt beszínezzük. Opcionálisan, ha az iterálás végetér a köszöbszám elérése előtt (a függvényérték elkezd minden határon túl nőni, ld. később), felhasználhatjuk azt az információt a pixel színének árnyalására. A kiértékelendő függvény és iterációja a következő képen működik: c egy konstans komplex szám (a pixelhez tartozó koordináta), z a fv. érték, ahol z0 mindig 0 (az első kiértékelendő z mindig a 0). Ezutóbbi információt elfelejtve, az általános képlet a következő: fc(z1) = z02 + c fc(z2) = z12 + c ... fc(zn+1) = zn2 + c, egészen n < küszöbérték-ig, valamint amíg el tudjuk hinni, hogy adott c esetén véges értékek közt marad z (|z| $\leq$ 2). Mivel viszont az utolsó mondatnak nagyon is tudatában vagyunk, látható, hogy igazából az iteráció a következőképpen fog kinézni: fc(z1) = 02 + c = c fc(z2) = c2 + c fc(z3) = (c2 + c)2 + c fc(z4) = ((c2 + c)2 + c)2 + c ... stb. A dolgot megbonyolíthatja, hogy a c konstans komplex szám. Nézzük meg először tehát ismétlésként, hogy miként működnek a komplex számokkal való műveletek! Komplex számok: szorzása Minden komplex szám egy valós és egy imaginárius komponensből tevődik össze (pl. a + bi, ahol a és b valós számok, i az ún. imaginárius állandó, amire igaz az, hogy i2 = -1; a a valós rész, bi a képzetes/imaginárius rész). Ennek ismeretében próbáljunk meg először négyzetre emelni, hiszen igazából úgyis arra lesz szükségünk, és a szorzást is bemutatja. (a + bi) * (a + bi) = a2 + a * bi + a * bi + b2i2 = a2 + 2 * a * bi - b2 (mivel i2 ugye -1) Átrendezve: a2 - b2 + 2abi, ahol a2 - b2 felfogható egy új komplex szám valós részének, 2abi pedig a képzetesnek. Jelölve őket x-szel és y-nal: = x + yi (ahol x = a2 - b2 és y = 2ab) Komplex számok: összeadása Komplex számok összeadása komponensenként történik, tekinthetünk rá úgy is, mint vektorok összeadására. Példa: (a + bi) + (c + di) = (a + b) + (c + d)i Ezek ismeretében most már elméletileg mindent tudnunk kell egy Mandelbrot-halmazt ábrázoló diagram elkészítéséhez.

Szükséges hozzá a libpng++ package: sudo apt-get install libpng++-dev Fordítás menete: g++ -fopenmp -c mandelbrot.cpp "libpng-config --cflags" g++ -fopenmp -lpthread -o mandelbrot mandelbrot.o "libpng-config --ldflags" Futtatás: ./mandelbrot "kimenet.png" A fentiekben elmagyarázott algoritmus 1:1 megvalósítása, apróbb extra infókért ld. kommentek (saját mind). Virtuálisan bármekkora felbontást támogat, csak a memória szab neki határt (na meg a számítási időigény :p).

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

Megoldás forrása: `/book_sources/fejezet5/mandelbrot.cpp`

A kód minimális változtatásokat igényelt csak. A néhány módosított sorban látható, hogy egyszerűbben kifejezhető lett a gondolatmenet, és az ujZ változóra sem volt szükség így, hisz nincs komponensenkénti data hazard.

5.3. Biomorfok

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

Megoldás forrása: `/book_sources/fejezet5/biomorf.cpp`

A biomorf fraktálok biológiai entitások formáihoz hasonlítanak (pl. sejtek), ezért különlegesek. Alapvetően elhibázott Julia-halmazról van szó minden esetben, ezért kevés módosítás kell. A Julia-halmazoknál a c rögzítve van, és a z jár be több értéket. Ha több Julia-halmazt kíván az ember animáltan megjeleníteni, felfogható egy kétváltozós függvényként is, ahol a c az adott pixel ami felett pl. az egérmutató van, z pedig a viewport összes pixelén csörtet végig; így megkapjuk az egérmutató által kijelölt komplex számhoz tartozó Julia-halmazt. A Julia-halmazok és a Mandelbrot-halmazok közötti egyik legjelentősebb kapcsolat az az, hogy ha egy Mandelbrot-halmazban lévő komplex számra értékeljük ki az ahhoz tartozó Julia-halmazt, akkor egy "pacába" tartozó alakzatot kapunk, ha pedig azon kívül, külön "szigetekbe" csoportosuló alakzatokat kapunk.

5.4. Mandelbrot nagyító és utazó C++ nyelven

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

Megoldás forrása: sourceforge repo

Nem sikerült felélesztени windows-on a qt-t.

5.5. Mandelbrot nagyító és utazó Java nyelven

forrás: https://forum.processing.org/two/discussion/comment/103600/#Comment_103600

Processing szükséges hozzá.

```
boolean flag;
int count=0;
int acc=3;
double space=0.90;
int stop=0;

float colorinc = 0;
```

```
float rotangle;
float posy;
void setup() {
  size(600, 600, P3D);

  // fullScreen(P3D);
  stop=(int)(width*2/space/space-width*2);
  noStroke();
}
void draw() {
  background(255);

  acc = int(map(mouseX, 0, width, -200, 200));

  flag=true;
  count+=acc;
  if (count < 0) count += stop;

  drawcircle(width/2, height/2, width*2+count%stop, 0.1, space, colorinc);
}


void drawcircle(float x, float y, float size, float umbral, double space,  ↩
   float _c) {

  color color1 = color(map(size, 0, width, 200, 0), 150, 50);
  color color2 = color(50, 150, map(size, 0, width, 0, 200));

  fill(flag?  color1:color2);
  ellipse(x, y, size, size);


  if (size > umbral) {
    _c+=0.1;
    posy+=0.001;
    flag=!flag;
    size*= space;
    drawcircle(x, y, size, umbral, space, _c);
  }
}
```

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás forrása: diasor

```
public class PolárGenerátor { // osztály
    // fieldek
    boolean nincsTárolt = true;
    double tárolt;

    public PolárGenerátor() { // konstruktor
        nincsTárolt = true; // újra beállítja, mert ez az eredeti src és  ↩
           kész
    }

    public double Következő() {
        if (nincsTárolt) {
            double u1, u2, v1, v2, w;
            do {
                u1 = Math.Random();
                u2 = Math.Random();
                v1 = 2 * u1 - 1;
                v2 = 2 * u2 - 1;
                w = v1 * v1 + v2 * v2;
            } while (w > 1);
            double r = Math.sqrt((-2 * Math.log(w)) / w);
            tárolt = r * v2;
            nincsTárolt = !nincsTárolt;
            return r * v1;
        } else {
            nincsTárolt = !nincsTárolt;
```

```
            return tárolt;
        }
    }

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

Ahogy a feladat utasítása is tisztázta, itt nem a matek a lényeg. Van egy osztályunk, benne vannak attribútumaink és metódusaink. Az első metódus amit láthatunk a konstruktor, alatta meg van egy ilyen Következő() nevű függvény. Ennek nagyon örülünk, mert szépen egységbe van zárva. A példányosított objektumnak mindig lesz egy állapota amit a fieldjei határoznak meg, és meg lehet rajta hívni a metódusokat, amik azt mutálják. Ez szerencsés, mert elfedi az implementációt. Látható ahogy a main()-ben példányosítunk egy ilyen objektumot, és kitolunk pár polárkoordinátát (?) a stdout-ra.

6.2. LZW

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

Megoldás forrása: https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/ziv/z.c + jó erős refaktor, ld. `/book_sources/binfa.c`

A forrás elég kaotikus volt, számos hibával, szóval refaktoráltam csöppet - ami meglepő módon legalább segített felfogni mi is történik, és rávett, hogy beletegyek némi valódi effortot. Márpedig ami lényegében történik, az nem túl bonyolult. Hozzávalók: egy szabványos fa-struktúra, némi boilerplate kód, és egy csipetnyi ciklusmag, ami a bemeneten csörtet végig karakterenként. Amennyiben az éppen beolvasott bájt értéke a [0-255] tartomány alsó felébe esik, 0-t tárolunk el a bal oldali levélelemében az aktuális faelemnek ahol épp vagyunk, ha pedig a felső felébe esik, 1-et tárolunk el a jobb oldali levélelemében az aktuális faelemnek, ahol épp vagyunk. Ezen annyit csavarunk csak, hogy ha két egymást követő olyan bájt van, amelyik ugyanabba a fél-intervallumba tartozik, akkor ahelyett hogy felülírnánk az adott levélelemet ahova a bájt tartozna, készítünk oda egy új levélelemet, továbbágaztatva a fát. Ezzel belekódoljuk a fába az adott "mintájú" bájtsorozatokat. A fa minden éle egy olyan mintázatot jelöl, ami megtalálható az eredeti bemenetben. Nyilván mivel mi az alapján helyezzük el ezeket a levélelemeket, hogy a [0-255] tartomány alsó vagy felső felébe esnek, ezzel nem megyünk sokra (az adat elvész). Ha azonban nem bináris fát, hanem többelemű fát használunk, ez a faszerkezetes algoritmus tömörítésre használható. A kiíratás inorder sorrendben történik (eredetileg right-first stílusban, én left-first-re írtam át), a memória-felszabadítás pedig postorder sorrendben. A kiíratás kódja elég triviális, magától értetődő. Sima rekurzió.

```
void printInorder(BINFA_PTR elem)
{
    if (elem != NULL)
    {
        printInorder(elem -> bal_nulla);
        printf("%d", elem -> ertek);
```

```
        printInorder(elem -> jobb_egy);
    }
}
```

Aki az elemek sorrendjére kíváncsi az ilyen bejárásoknál, látogasson el erre az oldalra: https://www.geeksforgeeks
tree-traversals-inorder-preorder-and-postorder/.

6.3. Fabejárás

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

forrás: az előző kód, a következő kiegészítésekkel, ld alább

```
void printPreorder(BINFA_PTR elem)
{
    if (elem != NULL)
    {
        printf("%d", elem -> ertek);
        printPreorder(elem -> bal_nulla);
        printPreorder(elem -> jobb_egy);
    }
}

void printPostorder(BINFA_PTR elem)
{
    if (elem != NULL)
    {
        printPostorder(elem -> bal_nulla);
        printPostorder(elem -> jobb_egy);
        printf("%d", elem -> ertek);
    }
}
```

Az elemek sorrendje, példával, ld. az előbb linkelt oldalon. Érdemes megfigyelni, hogy az elnevezés rendkívül bőbeszédű (nekem még nem említették soha, most figyeltem fel rá)! Ugye, mindig az -order kap egy előtagot. Ami befolyásolja hogy pre-, in- vagy post- előtagot kap, az az, hogy mikor kerül a gyökér kiíratásra! - ha előbb a gyökér, majd a két levélelem kerül meglátogatásra, preorder - ha levél-gyökér-levél, azaz szép sorban ahogy egymás után jönnek a sorrend, inorder - ha előbb a levelek, majd végül a gyökér kerül meglátogatásra, postorder Ezen kívül más extra ehhez a feladathoz nem nagyon kellett, max. a végén még a main()-ben meghívni a printPreorder() és a printPostorder() metódusokat, egy próba erejéig.

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 forrása: az előző kód, saját átalakításokkal, valamint némi inspirációkkal a "hivatalos" forrásból (https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/binfa/binfa.cpp), ld. `binfa.cpp`

Mivel az utasítás nem volt túlságosan tiszta, a következő feltételezést tettem: a legyen kompozícióban azt jelenti, hogy része legyen a gyökérelem mutatója a Tree osztálynak Egyébként kb. tök ugyanaz történik, mint a C változat esetében, csak itt struct-ok helyett class-okra támaszkodunk, és szépen eljátszogatunk a láthatósággal és az enkapszulációval. Van egy Tree objektumunk, benne egy root elemmel, ami a gyökérelemre mutat (így a root a Tree-vel "kompozícióban" van). Emellett a Tree-ből még nyilvános az addElement() és printInorder() metódusok is, előbbi az LZW-logikát tartalmazza, utóbbi pedig ahogy a nevéből is látszik, egy inorder printoutot. A privát mezőkben található egy position pointer, ami a fában mindig arra az adott elemre mutat, ahol épp dolgozunk - illetve egy resetPosition() helper metódus is, ami elvégzi a position gyökérelemre állítását, ha szükéges. Itt található még a nested Node class is, amiben megvannak az adott levélelemek szükséges fieldjei (érték, bal elemre mutató mutató, jobb elemre mutató mutató), és az ezeket elfedő getterek és setterek. Természetesen mind a Tree mind a beágyazott Node class rendelkezik a szükséges destruktorokkal és konstruktorokkal is.

6.5. Mutató a gyökér

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

forrás: Az előző kód, saját átalakításokkal. `/book_sources/fejezet6/binfa_mutato.cpp`

Mivel az utasítás nem volt túlságosan tiszta, a következő feltételezést tettem: a legyen aggregációban azt jelenti, hogy a gyökérelem mutatóját a Tree egy példánya csak megkapja Vegyük sorra a változtatásokat! A legapróbb és legtriviálisabb az osztály példányosításánál található: a konstruktor megkap egy Node típusú objektumpéldány-mutatót, amit majd a gyökérelem mutatójaként tárol el a Tree fa nevű példányának root elemében. Ahhoz, hogy ezt megtehessük, publikussá kell tenni a Node osztályt előbb. Ehhez az osztályt picit átmozgattam a kódban. További változtatás, hogy a konstruktor ugye át lett alakítva, hogy Node-pointert fogadjon és eltároljon, valamint hogy a destruktor mostantól a default destructor, mivel a gyökérobjektum külön létezik az osztálypéldánytól, így megtartja saját alfáját (ami ezesetben ugye az egész fa) bármi is történjék - ahhoz, hogy magát a fát töröljük ki, a gyökérobjektum életciklusát kell ezután majd kézzel menedzselnünk (delete xy;). Nagyon fontos például, hogy a jelenlegi példában a gyökérobjektumot csak a fa objektumon keresztül tudjuk elérni, mivel helyben példányosítottunk. Ez azzal jár, hogy ha a fa objektumot felszabadítjuk, a mögötte lévő bináris fa továbbra is jelen marad, viszont mi elvesztjük a mutatónkat rá, így fel sem fogjuk már tudni szabadítani többet (memory leak).

6.6. Mozgató szemantika

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

Próbáltam megoldani amennyire csak lehet, de nem igazán jártam sikerrel. Errefelé nézelődtem legfőképp: https://docs.microsoft.com/en-us/cpp/cpp/move-constructors-and-move-assignment-operators-cpp?view=vs-2019 és https://stackoverflow.com/a/3109981. Elméletileg a lényeg az lenne, hogy "ellopjuk" a másik Tree objektumtól a gyökérmutatót, ezáltal kivéve őt és az alatta lévő fát a másik objektum életciklusából.

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:

Amennyire kivettem a feladatból és a videóból, itt igazából ismét az OO szerkezeten van hangsúly, nem konkrétan a működésen. A hangyák mind külön osztálypéldányok, és úgy hatnak egymásra. Maga a feladat valamennyire hasonlít az életjátékra (ld. 7.2), de nem ugyanazt a működést valósítja meg. Az UML diagram (vagy legalábbis valami ahhoz nagyon hasonló) a könyvben linkelt forrásban is megtalálható. A legfontosabb megfigyelés talán itt, hogy minden Ant külön szálon, AntThread-en fut.

7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása: https://processing.org/examples/gameoflife.html

Ez már sokkal közelebb áll hozzám. Conway életjátékának a szabályainak az ismertetésétől most eltekintenék, akit érdekel, itt elolvashatja: https://hu.wikipedia.org/wiki/Életjáték. Maga a játéknak a célja filozófiai vonatkozású. Azt példázza, hogy néhány, egyszerű szabály segítségével nagyon komplex működések és folyamatok alakulhatnak ki. Erre egy példa a feladat utasításában megemlített "sikló-kilövő" is, amely mozgásban is látható a hivatkozott Wikipédia-oldalon. A kód működtetéséhez a Processing nevű eszközre lesz szükség. A működése nem túl bonyolult, pláne nem valakinek aki már dolgozott Processinggel (aki még nem, annak ajánlom!). Először a setup() metódusban felvesszük a cellarácsot, és véletlenszerűen kitöltjük cellákkal. Ezt követően a draw() felelős minden újonnan megrajzolt képkockért, onnan kerül meghívásra az iteration() metódus, mely a játéklogika alkalmazásáért felelős. A kód tartalmaz némi interaktív részt is, nevezetesen lehetővé teszi a szimuláció megállítását, és a cellák kézzel kitöltését is, egyedi formákat lehetővé téve.

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...

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Megoldás forrása: `/book_sources/fejezet8/szoftmax.py` https://www.geeksforgeeks.org/softmax-regression-using-tensorflow/

Az MNIST egy klasszikus ML kezdőfeladat. Cél hogy felismerjünk kézzel írt számjegyeket 0 és 9 között. Ezt a feladatot classifyingnak nevezzünk, azaz osztályozásnak - ennek a folyamatnak a során a bemenet pixeleiből eldöntjük hogy a 10 számjegy-kategóriából melyikbe mennyi eséllyel tartozik az adott kép. A működés alapvetőségének megfelelően triviális. Van egy bemeneti mátrixunk ahova az adott kép pixeleit töltjük be, egy átmeneti layerünk, és egy kimeneti layerünk, illetve ezek között súlyok. A súlyokat fokozatosan fogjuk változtatni az alapján, hogy az ún. training set-ben az adott mintaképhez milyen mintamegoldás tartozik. Ezt a folyamatot backpropagation-nek hívjuk. Magát az algoritmust pedig gradient descent-nek nevezzük. Az MNIST példaadatbázis 28x28 méretű kézzel írt számjegyeket tartalmaz, több tízezret. Segítségével ez a kód kb. 85%-os hatékonysággal képes megtanulni a számjegyek elolvasását.

8.2. Szoftmax R MNIST

Megoldás forrása: `/book_sources/fejezet8/softmax.r` https://tensorflow.rstudio.com/tensorflow/-articles/tutorial_mnist_beginners.html

Tök ugyanazt csinálja mint az előző Python-os példa, csak kicsit tömörebben, mivel az R statisztikai használatra specializált nyelv/eszköz.

8.3. Mély MNIST

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

Azért deep, mert ún. deep learningről van itt szó. Itt extra ún. hidden layer-ek, konvolúciós neurális hálók is részt vesznek a feldolgozásban. Az alapvetés azonban továbbra is változatlan. Kifinomultságának köszönhetően ez már magasabb accuracy-t ér el a teszteken, ezért érdemes használni.

8.4. Deep dream

A Deep Dream egy kísérleti projekt volt a Google-nél, mintákat lát bele képekbe előre betanított képek alapján. Attól függően milyen a training set, más-más dolgokat lát bele a képekbe. A végeredmény jellemzően meglehetősen "trippy". Kipróbálható a https://deepdreamgenerator.com/ weboldalon.

8.5. Minecraft MALMÖ

Megoldás videó:

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

A Project Malmö egy Minecraft-ot játszó AI ágens, tetszőleges feladatra betanítható.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Lisp egy funkcionális programozási nyelv, jellegzetessége a zárójelek hadserege amit minden egyes kód felvonultat. Már nem igazán használják. A Scheme egy Lisp-dialektus, a mostani példában azt használjuk.

```
(define (faktorialis n)
    (define (iter product counter)
        (if (> counter n)
            product
            (iter (* counter product) (+ counter 1))
        ))
    (iter 1 1))
(display (faktorialis 3))
```

Először is iterálnunk kellene tudni. Mivel a Scheme-ben nincsenek szokványos vezérlési szerkezetek, így az iterációt is saját kézzel kell, ez látható a második sorban. Az `iter` vár két változót argumentumaként, első a ciklusváltozó, második a számláló. Az eljáráson belül teszteljük a szabad változó értékét (n, előltesztelős ciklust írtunk, ha meghaladja értéke a counter értékét akkor visszaadja az addig kiszámolt faktoriális értéket, ellenben folytatja annak számítását. Hiába tűnik azonban iterációs módszernek, a háttérben ez továbbra is rekurzívan fog futni. Ennek persze megvannak a saját teljesítménybeli következményei, de ez most minket hidegen hagy. Viszont ha már amúgy is rekurzívan fog futni, érdemes inkább alapból rekurzív megközelítést használni, nem? Rekurzívan:

```
(define (faktorialis n)
(if (= n 1)
1
(* n (faktorialis (- n 1) ))
))
(display (faktorialis 3))
```

III. rész

Második felvonás

10. fejezet

Helló, Berners-Lee!

10.1. C++ vs. Java

Bár mind a C++ és a Java OO szemléletet előtérbe helyező nyelvek, megvalósításukban jelentős eltéréseket mutatnak. Ezek a különbségek más területen való használatra ösztönzik az adott nyelvek fejlesztőit, de ez nem azt jelenti, hogy nem helyettesíthetnék egymást sikererrel. Ehhez azonban alaposan ismerni kell mindkét nyelvi stacket, különben könnyen levonhatunk hamis következtetéseket.

Vegyük először a legnyilvánvalóbbat, a menedzseltséget. Minden Java kód a JVM (java virtual machine) köztes nyelvére fordul, nem pedig natív kódra. Futtatás során a JVM beindul, és elkezdi futtatni ezt a köztes nyelven írt kódot. Bár úgy tűnhetne, hogy ez jelentősen lelassítja a működést, ez az esetek többségében nem igaz.

A memóriakezelésben is eltér a két nyelv. Javában kizárólag referenciák vannak, mindent kivesz a kezünkből a JVM és a GC. A JVM gráfot épít az objektumreferenciáinkból, és bizonyos időközönként a GC végigjárja ezt a gráfot, nem referrált objektumok után kutatva. A memóriát így a GC tartja karban helyettünk, kizárva a memory leakek lehetőségét.

A java nem támogat többszörös öröklést. Helyette interfészeket kell használni.

10.2. Python alapok

A Python egy rendkívül széleskörűen használt, gyors prototipizálást lehetővé tevő szkriptnyelv. Általános vélekedés, hogy a tanulási görbéje viszonylag alacsonyan van, az előálló kód pedig tömör és jól olvasható. A gyors fejlesztést és tesztelést elősegíti, hogy a nyelv alapból interpretált, így minden változtatás kvázi azonnal leellenőrizhető / kipróbálható.

A nyelv maga számos apróbb ponton eltér a C/C++ stílusú nyelvektől. Gyengén típusos nyelv, a típusok közt automatikusan konvertál (duck typing). A scope-olás szintén eltér, a Python ugyanis whitespace-érzékeny, behúzások jelölik az egy scope-ba tartozó statementeket. A statementeket nem zárja pontosvessző, a statement-végek automatikusan detektáltak a tokenizáció során. Ha egy statement több soros, backtick használható több sorba tördelésre, mint más nyelvekben.

Az adattípusok listája limitált, az említett duck typingnak hála. Elkülönülnek a számszerű objektumok, a karakterek, a boolean értékek, a None, és a szekvenciák (stringek, tuple-ök, listák, dict-ek). A számok

magukban foglalják az előjeles és előjelmentes egészeket, a törteket, a komplex számokat. A None a null megfelelője. A szekvenciákhoz tartozik kb. 10 egységesen használható metódus, valamint mindegyik fajtához elérhetőek továbbiak is.

A stringek szekvenciaként vannak kezelve, ez talán emlékeztethet minket a C-ben lévő karaktertömbökre. Literálokat aposztrófok vagy idézőjelek közt lehet megadni. Köszönhetően annak, hogy az általános szekvencia-műveletek alkalmazhatóak rájuk, a stringmanipuláció általánosságban egyszerű Pythonban.

A tuple-ök elem n-esek. Több féle objektumot is össze lehet fogni velük, hasznos mikor például egyszerre több objektumot is vissza akarunk adni egy hívás végeztével. Zárójelek közt, az objektumok és/vagy literálok vesszővel elválasztott felsorolásával lehet őket elsősorban létrehozni.

A lista akár több féle objektum rendezett szekvenciája. Elemeire indexekkel lehet hivatkozni. Ha negatív indexet használunk, akkor a szekvencia végéről szedjük az elemeket (ld. később). Tetszőlegesen bővíthető. Létrehozásuk szögletes zárójelek közti, vesszővel elválasztott objektum-referencia- vagy literálfelsorolással történik.

A dictionary-k (azaz szótárak) kulcs-érték párokból álló elemhalmazok. Ebből kifolyólag nem rendezettek, az elemeikre a kulcsokkal lehet hivatkozni. Más nyelvekben ezeket asszociatív tömbökként is szokták hívni.

Most nézzük a nyelv elemeit! A változódeklarálás egyszeren egyenlőségjellel történik. Nem szükséges típust megadni, csak nevet (van amikor ez gondot is okoz). Különböző shorthandeket is lehet használni, mint az "x=y=z=1", ahol mindhárom változó létrejön 1 értékkel. Változócserét is le lehet így rövidíteni, a "x,y = y,x" pontosan ezt fogja tenni. A deklaráláson túl a nyelv a predikátumok leírását is le tudja rövidíteni: az "a<b<=c==d" pl. ugyanaz mint az "a<b && b<=c && c==d". Szekvenciákból ki lehet hasítani a Matlabban ismert módokon, pl: a[5:7] az az 'a' szekvenciából kiszedi az elemeket az ötös indextől a hetesig.

Az elágazások az if, elif, else kulcsszavakkal történnek, a kulcsszavak után a szükséges predikátum kerül, majd egy kettőspont. Pl.:

```
if szam < 0:
  print 'Kisebb, mint 0.'
elif szam == 0:
  print 'A szám egyenlő nullával.'
else:
  print 'A szám nagyobb nullánál.'
```

A ciklusok hasonlóak minden más nyelvhez. A szintaxis Matlabhoz hasonlít, jellemzően az iterátoros szekvencia-bejárás a célszerű.

A függvények definiálása a "def" kulcsszóval történik. A paraméterek érték szerint adódnak át, kivéve mutable típusoknál (pl. listák, szótárak). Lehetőség van alapértelmezett érték megadására is, más nyelvekből is ismerhető módon. Bár alapból a pozíció-szerinti argumentumegyeztetés a jellemző, Pythonban lehetőség van név-szerinti argumentum egyeztetésre is, ami extra flexibilitást kölcsönöz a kódnak.

A nyelv bár nem objektum-orientáltként indult, azóta az OO paradigma teljeskörű implementációt kapott. A class osztállyal hozhatunk létre osztályokat, melyekben ugyanúgy elhelyezhetünk attribútumokat és metódusokat mint bármilyen más OO paradigmát támogató nyelvben.

Az osztályokon belül a metódusok definiálása picit eltér a globálisan használható stílustól. Kötelező, hogy az első argumentumuk a self kulcsszó legyen, mely mindig az éppen aktuális objektumpéldányra fog referrálni. Az osztályokon belül létre lehet hozni egy speciális, konstruktor szerű metódust is, ennek az

"__init__" nevet kell adni. Más speciális funkciójú metódusok is léteznek, ezeket mivel a forrásanyag sem tárgyalta, most itt is eltekintek a részletezésétől.

A kivételkezelés szintén létező dolog a nyelvben, és a szokványos try/catch stílusban történik; annyi csavarral, hogy itt catch helyett "except" kulcsszót használunk. Funkcionalitásra ugyanaz, adott fajta exceptiont lehet vele elkapni.

Végezetül érdemes megemlíteni még a modulrendszert. A Python köré szervezett ecosystem modulokból épül fel; ezek behúzásával lehet extra funkcionalitást könnyen hozzáadni a kódbázisainkhoz. Hasonló más nyelvekben is megtalálható package rendszerekhez, mint C#-nál a Nuget, vagy Java esetén a Maven packagek, JS esetén az NPM modulok, stb..

IV. rész

Irodalomjegyzék

10.3. Általános

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

10.4. C

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

10.5. C++

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

10.6. 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.