

BGRg 13 Wenzgasse

1130 Wien, Wenzgasse 7

Vorwissenschaftliche Arbeit

zum Thema

**Problemlösung mit
esoterischen Programmiersprachen**

Verfasser Jakob Kuen

Klasse 8C

Betreuer Mag. Werner Jägersberger

Schuljahr 2015/16

Abgabe Wien, am 12.02.2016

Unterschrift _____

Abstract

Diese vorwissenschaftliche Arbeit behandelt das Thema der esoterischen Programmiersprachen. Esoterische Programmiersprachen werden als Experiment oder auch als Witz entwickelt und sind deswegen meistens nutzlos. Solche Sprachen sind großteils ein Internet-Phänomen und werden auf der Webseite „esolangs“ gesammelt.

In der Arbeit werden unterschiedliche esoterische Programmiersprachen und ihre Besonderheiten im Vergleich zu normalen Programmiersprachen vorgestellt. Im Hauptteil der Arbeit wird die bekannteste esoterische Programmiersprache, „Brainfuck“, genauer untersucht. In dieser Sprache wurden im Rahmen der Arbeit zwei Programme geschrieben.

Das erste Programm ist eine Umsetzung eines Algorithmus zur Lösung einer mathematischen Aufgabenstellung. Das Ziel des Programms ist es, alle geraden Fibonacci-Zahlen unter 4.000.000 zu berechnen.

Das zweite Programm ist ein sogenannter „Roboter“. Es tritt in einem Spiel, das „Capture the Flag“ ähnelt, gegen Roboter anderer Programmierer an. Das Spiel wird gegen mehrere Gegner gespielt und die Ergebnisse werden ausgewertet.

Mithilfe der zwei Programme können durchaus sinnvolle Verwendungszwecke esoterischer Programmiersprachen gezeigt werden. Durch die sehr abstrahierten Programme werden bestimmte Methoden die auch in normalen Programmiersprachen Gebrauch finden verständlicher gemacht, und eine neue Perspektive auf bestimmte logische Abläufe ermöglicht.

Inhaltsverzeichnis

Abstract	1
1. Einleitung.....	4
2. Esoterische Programmiersprachen	5
2.1 Was sind esoterische Programmiersprachen?	5
2.1.1 Minimalismus	5
2.1.2 Neues Konzept	8
2.1.3 Bizarrerie	10
2.1.4 Thematik	11
2.1.5 Humor	13
2.2 Werkzeuge zum Arbeiten mit Brainfuck.....	15
3. Code Golf und Project Euler	17
3.1 Über Project Euler.....	17
3.3 Das Programm	19
4. BFJoust.....	24
4.1 Das Spiel.....	24
4.2 Regeln	24
4.3 Strategien.....	25
4.4 helyx_FightAndFlight	27
4.4.1 Verwendete Strategien	27
4.4.2 Ergebnisse gegen andere Bots	28
5. Fazit.....	34
Literaturverzeichnis	35
Abbildungsverzeichnis	37

1. Einleitung

Das Thema der esoterischen Programmiersprachen interessiert wenige, und noch weniger Menschen kennen sich wirklich mit ihnen aus. Programmiersprachen, die entwickelt wurden, um keinen Nutzen zu besitzen, oder um einen Witz zu machen. Oft ist es aber doch möglich, sie zur Lösung von mathematischen Problemen zu verwenden, zum Beispiel bei der Aufsuchung von geraden Fibonacci-Zahlen oder bei der Eroberung von der Flagge eines gegnerischen Programms. Die Community rund um esoterische Programmiersprachen ist 44 Jahre nach der Erfindung der ersten Sprache, INTERCAL, immer noch aktiv. Aus dieser einen Programmiersprache entwickelten sich hunderte von Abwandlungen und neuen nutzlosen Programmiersprachen.

Eine der wenigen Quellen für Informationen zu esoterischen Programmiersprachen ist die Wikipedia-ähnliche Webseite „esolangs“. Auf dieser Webseite werden von der Community geschriebene Artikel über ihre esoterischen Sprachen gesammelt. Außerdem gibt es Artikel über Wettbewerbe und Herausforderungen rund um esoterische Programmiersprachen. Außer dieser Webseite gibt es nur sehr wenige Quellen, das Thema ist fast nur auf das Internet begrenzt. Es gibt nur sehr wenige Bücher die sich mit dieser Thematik befassen.

Das Hauptziel der Arbeit ist es, die wenigen Verwendungszwecke von esoterischen Programmiersprachen zu erkunden, und solche Programme zu schreiben. Im Rahmen dieser vorwissenschaftlichen Arbeit werde ich zwei Programme in esoterischen Sprachen schreiben: eines zur Lösung einer mathematischen Aufgabenstellung, und einen Bot der in einem „Capture the Flag“ Spiel gegen andere Bots antritt.

Im ersten Teil der Arbeit wird ein Überblick über die absonderliche Welt der esoterischen Programmiersprachen gegeben. Es wird auf mehrere Sprachen genauer eingegangen, und ihre Funktionsweise erklärt. Der zweite Teil erklärt die zwei Programme und geht auf Herausforderungen und Schwierigkeiten der Problemlösung genauer ein.

2. Esoterische Programmiersprachen

2.1 Was sind esoterische Programmiersprachen?

Esoterische Programmiersprachen sind Programmiersprachen, die nicht für das Programmieren von Software entwickelt worden sind, sondern um mit unkonventionellem Sprach-Design zu experimentieren und Programmiersprachen zu bestimmten Themen zu entwickeln. Einige esoterische Programmiersprachen wurden auch als humorvolle Parodie anderer Programmiersprachen entwickelt. Esoterische Programmiersprachen werden von einigen wenigen Internet-Communities entwickelt und verwendet. Mit esoterischen Programmiersprachen werden zum Beispiel mathematische und informatische Herausforderungen gelöst. Bei dem Spiel Brainfuck-Joust werden Roboter(Bots), programmiert in einer esoterischen Programmiersprache, verwendet, die gegeneinander eine Variante von "Capture the Flag" zu spielen. Die meisten esoterischen Programmiersprachen können in fünf verschiedene Kategorien eingeteilt werden.

2.1.1 Minimalismus

Esoterische Programmiersprachen aus dieser Kategorie bestehen aus den möglichst wenigen Befehlen und sind nur sehr gering für das Schreiben von Programmen optimiert. Eine minimalistische Sprache, die Turing-Complete ist, wird Turing-Tarpit genannt. Eine Turing-Complete Programmiersprache muss jede Berechnung, die von einer Turing-Maschine¹ berechnet werden kann, auch selbst berechnen können.

Brainfuck

Brainfuck ist wahrscheinlich die bekannteste esoterische Programmiersprache. Ein Brainfuck Programm manipuliert einen Array (auch "Memory Tape" genannt). Jedes Element des Arrays besitzt am Anfang des Programmablaufs den Wert 0. Ein Brainfuck Programm kann diesen Array mithilfe von 8 Befehlen modifizieren. Brainfuck ist Turing-Complete, und kann somit jede mathematische Berechnung durchführen, wenn auch meist sehr ineffizient.

¹ Eine von Alan Turing entwickelte Maschine, welche die Arbeitsweise eines Computers mathematisch leicht analysierbar macht.

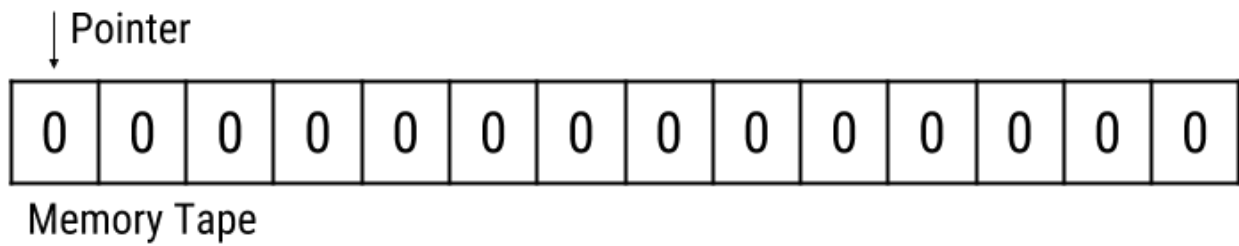


Abbildung 1

(Quelle 1, esolangs 2005, Brainfuck)

Befehl	Beschreibung
>	Bewegt den Pointer nach rechts
<	Bewegt den Pointer nach links
+	Erhöht den Wert an dem Pointer
-	Verringert den Wert an dem Pointer
.	Druckt den Wert an dem Pointer
,	Speichert einen eingegebenen Wert an dem Pointer
[Springt zum passenden] im Programm, falls der Wert an dem Pointer 0 ist
]	Springt zum passenden [im Programm, falls der Wert an dem Pointer nicht 0 ist

Tabelle 1

(Quelle 2, esolangs 2005, Brainfuck Language Overview)

In Tabelle 1 sind alle Befehle der Sprache aufgelistet. Fast alle dieser Befehle können sehr leicht in anderen Programmiersprachen emuliert werden. Aus diesem Grund gibt es sehr viele Interpreter in unterschiedlichen Sprachen. Für mein Project Euler Programm habe ich einen eigenen Interpreter in Python geschrieben. Um das Schreiben von Brainfuck Pro-

grammen zu erleichtern, existieren auch Transpiler, die bestimmte Programme aus unterschiedlichen Programmiersprachen wie zum Beispiel C oder Java zu Brainfuck verarbeiten. Da die Befehle von Brainfuck in ihrer Funktionalität begrenzt sind, müssen komplexere Befehle wie *if*, *for*, und *while* zu sehr langen Ketten aus Brainfuck-Befehlen übersetzt werden, um denselben Effekt zu erzielen.

```
>>++++<---
```

Dieses einfache Brainfuck Programm modifiziert zwei Zellen des Memory Tapes.

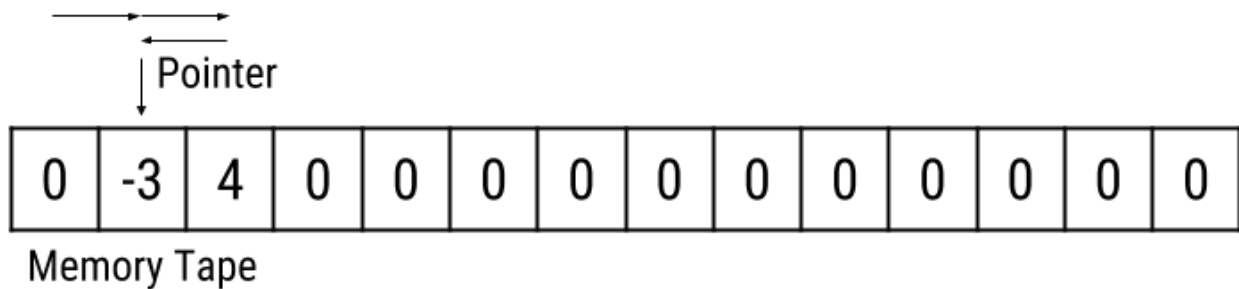


Abbildung 2

In der originalen Implementation des Brainfuck Interpreters und den meisten folgenden repräsentiert der Wert einer Zelle ein ASCII Zeichen. Wird dieser Wert gedruckt, wird nicht sein Zahlenwert, sondern sein korrespondierendes ASCII Zeichen ausgegeben. Dieses Programm druckt so die Worte "Hello World!". (*Quelle 3*, esolangs 2005, Brainfuck Examples)

```
1:  +++++++
2:  [
3:  >++++
4:  [
5:  >+>+>+>+>+>+
6:  <<<<-
7:  ]
8:  >+>+>->+>+
9:  [
10: <
11: ]
12: <-
13: ]
14: >>.>---.++++++..+++>>.
15: <-.<..+++
16: -----
17: -----
18: >>.>+>.
```

2.1.2 Neues Konzept

Programmiersprachen, die alternative Wege des Sprachdesigns erforschen.

Befunge

Bei der Programmiersprache Befunge wird das Programm in einem zweidimensionalen Raum geschrieben, und der Ablauf des Programms wird mit den Befehlen `<>^v` in eine Richtung "gelenkt". Neben diesen Befehlen gibt es eine Vielzahl von anderen, die unterschiedliche Auswirkungen auf den Ablauf des Programms haben. Mit dem `p` Befehl kann sich ein Befunge Programm sogar selbst verändern, während es abläuft.

(Quelle 4, esolangs 2006, Befunge Language Overview)

```
2>:3g" "-!v\  g3θ      <
|!`"0":+1_:.:03p>03g+: "0" `|
@      ^  p3\" " :<
2 23456789012345678901234567890123456789012345678901234567890123456789
```

Dieses Befunge Programm ist eine Implementation des Sieb des Eratosthenes, und wird zur Berechnung von Primzahlen verwendet.

(Quelle 5, esolangs 2006, Befunge Examples)

Befunge ist Turing-Complete, wodurch unter anderem dieser Befunge Interpreter möglich ist, der selbst in Befunge geschrieben worden ist.

028p038p108p018pv

```
vp91+56p900<      v_v#!-+1"!":<      >:"<"-!#v_-!"^"-!#v_ v
>"*"09g:19g\19gg29p p 29g28g #^_:" "-!#v_:"v"-#^_ v
^p91+g91g81p90+g90g 8 0pg91g90g92$      <      <
      >:      >38g7p38g1+38p^p811p800<
      >28g!28p      ^p810p80-10<      <
      ^p81-10p800      <
      ^p810p801<      _v#!-">":<
      ^ -"0":_v#`\+1"9":_v#`-1"0":<      #
      >      #      >:"!1+-!#v_v
#####>19g+\48gp      ^      p #82!g82<
0"!dlroW olleH">v      #      ^      g7-1g83_v#!-"":<
      ,:      #      >$,      ^ <      #>:"p"-!#v_v
      ^_25*,@#      v_^#-4:_v#-3:_v#-1:_v#-2:\g7p83:-1_v#:g83<2<
#####      >:5-#v_v$      ^ #      0 #<
      ^      _v#-6< > $6      v      >$09g+48p1 >>      ^
      >$0>      #      ^      <
      v_      ^
      >*      ^      ^3_v#!-"": <
      >:""-#v_4 ^
      ^5_v#!-"*":<
      >      #@      ^
```

(Quelle 6, catseye 1997, befbef.bf)

2.1.3 Bizarrie

Das Ziel solcher Sprachen ist es, bizarr, anders, oder schwer zu verwenden zu sein.

INTERCAL

INTERCAL wurde im Jahr 1972 entwickelt, und ist die erste esoterische Programmiersprache. INTERCAL steht für „*Compiler Language With No Pronounceable Acronym*“. Das Ziel der Sprache ist es, möglichst wenige Ähnlichkeiten mit anderen normalen Programmiersprachen zu haben. Ein INTERCAL Programm ist aus einer Liste von Befehlen aufgebaut, die der Reihe nach durchgeführt werden.

(Quelle 7, esolangs 2005, Intercal Command syntax)

```
1: DO ,1 <- #13
2: PLEASE DO ,1 SUB #1 <- #238
3: DO ,1 SUB #2 <- #108
4: DO ,1 SUB #3 <- #112
5: DO ,1 SUB #4 <- #0
6: DO ,1 SUB #5 <- #64
7: DO ,1 SUB #6 <- #194
8: DO ,1 SUB #7 <- #48
9: PLEASE DO ,1 SUB #8 <- #22
10: DO ,1 SUB #9 <- #248
11: DO ,1 SUB #10 <- #168
12: DO ,1 SUB #11 <- #24
13: DO ,1 SUB #12 <- #16
14: DO ,1 SUB #13 <- #162
15: PLEASE READ OUT ,1
16: PLEASE GIVE UP
```

Einer der größten Unterschiede zu normalen Programmiersprachen ist in diesem "Hello World" Beispiel sichtbar. Der Befehl PLEASE wird verwendet, um die Höflichkeit des Programmierers zu testen. Falls zu wenige der Befehle mit PLEASE beginnen, läuft das Programm nicht.

(Quelle 8, Wikipedia 2015, Intercal Hello, world)

2.1.4 Thematik

Sprachen, die zu einem bestimmten Thema entwickelt worden sind.

Magicard!

Die Programmiersprache Magicard! ist aus Befehlen aufgebaut, die ein virtuelles Kartendeck modifizieren. Ein Magicard! Programm ähnelt deswegen einer Anleitung für einen Kartentrick.

(Quelle 9, esolangs 2013, Magicard!)

Befehl	Beschreibung
Unbox deck [number] {of [num] cards}	Das Kartendeck mit der Nummer [number] wird zum aktuellen Deck, falls es zum ersten Mal verwendet wird, wird die Anzahl der Karten mit {of [num] cards} festgelegt.
Rebox deck	Legt das aktuelle Kartendeck zur Seite und das zuletzt verwendete Deck wird wieder zu dem aktiven Deck
Shuffle {preserving [num] on [top/bottom]}	Mischt das aktuelle Deck, wobei die Reihenfolge der obersten/untersten [num] Karten erhalten bleibt
Count [num].	Mischt die obersten [num] Karten an das untere Ende des Decks
TA-DA!	Beendet das Programm

Tabelle 2

(Quelle 10, esolangs 2013, Magicard! Instruction Set)

```
Unbox deck 0 of 127 cards.
Take packet of 127 from top.
Repeat on next 5 decks.
Put packet on top of deck.
Biddle count 127 stealing when i='H' || i='e' || i='l'.
Biddle count 127 stealing when i='l' || i='o'.
Biddle count 127 stealing when i=' ' || i='W' || i='o' || i='r'
Biddle count 127 stealing when i='l'.
Biddle count 127 stealing when i='d'.
Biddle count 127 stealing when i='!'.
Deal 12 cards flipping each one with a flourish.
TA-DA!
```

(Quelle 11, esolangs 2013, Magicard! Examples)

ArnoldC

Arnold C ist eine Programmiersprache, die aus Arnold Schwarzenegger Zitaten besteht. Ein ArnoldC Programm wird zu Java kompiliert. Die verschiedenen Befehle der Programmiersprache sind äquivalent zu Befehlen aus normalen Programmiersprachen, wurden aber zu Arnold Schwarzenegger Einzeilern umbenannt. Viele andere esoterische Programmiersprachen bestehen wie ArnoldC nur aus einer Reihe von umbenannten Befehlen einer anderen Programmiersprache, wie zum Beispiel LOLCODE.

```
IT'S SHOWTIME
TALK TO THE HAND "hello world"
YOU HAVE BEEN TERMINATED
```

"IT'S SHOWTIME" und "YOU HAVE BEEN TERMINATED" signalisieren den Anfang bzw. das Ende eines ArnoldC Programms, und "TALK TO THE HAND" gibt einen Wert aus.

Befehl	Bedeutung
NO PROBLEMO	True
I LIED	False
BECAUSE I'M GOING TO SAY PLEASE	If
BULLSHIT	Else
YOU HAVE NO RESPECT FOR LOGIC	Endif
YOU ARE NOT YOU YOU ARE ME	Ist gleich
DO IT NOW	Methodenruf

Tabelle 3

(Quelle 12, Github 2013, ArnoldC Readme)

2.1.5 Humor

Sprachen, die keinen besonderen Zweck besitzen, sondern nur als ein Witz entwickelt worden sind. Solche Programmiersprachen sind häufig komplett nutzlos.

Seed

Diese Programmiersprache basiert auf Seeds, also auf Anfangswerten für ein Programm, das aus diesen Werten einen zufälligen Text generiert. Der Compiler generiert aus einem Seed ein zufälliges Befunge Programm. Um ein Programm in Seed zu schreiben, muss zuerst ein Befunge Programm geschrieben werden. Um von dem Text des Programms auf einen Seed rückzuschließen, wird unrealistisch viel Prozessorleistung benötigt. Die benötigte Leistung geht mit jedem weiteren Befunge Befehl exponentiell in die Höhe, ab mehr als sechs Befehlen wird das berechnen eines Seed Programms fast unmöglich.

(Quelle 13, esolangs 2009, Seed Writing programs)

340 983247832

Dieses Seed Programm wird von dem Seed Compiler zu einem Befunge Programm umgewandelt. Da der Seed zufällig gewählt worden ist, ist es sehr unwahrscheinlich, dass das resultierende Befunge Programm sinnvoll ist.

```
q
Z?T7yQ
;RyHIw*#{8}).' }iN*P{u>z#ok<w\\?!KPrV07U;b> B
f:rDj':T3'O~J(>BLLxj(>{5n) oM/?nwC{c(OT>Fv?=)tW*` 6oL8yCI:D_%4d}:ubmL"6v'(o4^5zi{E
3F+vDhK"*}a&nu=S*syIgT>MQ9_vyi'b&i^_xT"WP-"lk=#/r)8%:rG,I?'DTz<)|JJ0|^LDakzrx]Gjy=^.0
$R<y9#S1,_K5y@\\~z+jSlARiA6D#:gVlmb^>[MQea
9mUdq>MJxW0<PY%o{u:aw*rK9i\\;Wt8v4$01VRz]7rUg.#MJRRwt?M[cD{j='lz;$79J;ye
gDQF\\1
```

Dies ist das berechnete Programm, welches keinen offensichtlichen Nutzen besitzt.

Eines der wenigen berechneten Seed Programme, das funktioniert, ist 4 80814037. Das generierte Befunge Programm zu diesem Seed ist "h", welches den Buchstaben "h" druckt.

(Quelle 14, esolangs 2009, Seed Examples)

Byter

Byter besteht aus 11 Befehlen, welche einen Pointer auf einem "Spielfeld" bewegen, und so bestimmte Zeichen drucken. Das Spielfeld ist eine 16 mal 16 große Matrix.

[illegible]

Ein "." repräsentiert Zeichen die von Zeichensatz zu Zeichensatz unterschiedlich sind. Der Pointer startet in der oberen linken Ecke, und wird mit Befehlen bewegt.

Be- fehl	Beschreibung
>	Bewegt den Pointer nach rechts, und ersetzt den Befehl mit <
<	Bewegt den Pointer nach links, und ersetzt den Befehl mit >
V	Bewegt den Pointer nach unten, und ersetzt den Befehl mit A
A	Bewegt den Pointer nach oben, und ersetzt den Befehl mit V
+	Druckt das Symbol an dem Pointer und bewegt ihn nach oben
-	Druckt das Symbol an dem Pointer und bewegt ihn nach unten
\$	Druckt das Symbol an dem Pointer und bewegt den Pointer zurück zur ursprünglichen Position
#	Programm beenden

Tabelle 4

(Quelle 15, esolangs 2008, Byter)

2.2 Werkzeuge zum Arbeiten mit Brainfuck

Interpreter

Es gibt viele verschiedene Interpreter, die in allen möglichen Programmiersprachen geschrieben wurden, wie zum Beispiel JavaScript, C, Haxe, und auch Brainfuck selbst. Um ein Project Euler Problem zu lösen, habe ich einen Interpreter in Python geschrieben, mit einer leichten Abwandlung von normalen Interpretern. Um das Lösen des Problems zu ermöglichen, behandelt mein Interpreter Werte als Zahlen und druckt ihre Zahlenwerte, anstatt ihrer entsprechenden ASCII Zeichen.

```
code = ">++.--."
ptr = 0
arr = [0] * 512
bracMap = [0] * 512
breakTimer = 10

i = 0

def makeBracMap():
    for i in range(0, len(code)):
        char = code[i]
        if char == "[":
            x = 1
            c = i
            while(x != 0):
                c+=1
                if code[c] == "]":
                    x-=1

                if code[c] == "[":
                    x+=1
            bracMap[i] = c;
            bracMap[c] = i;

makeBracMap();

while i < len(code):
    char = code[i]
    if char == ">":
        if ptr < 127:
            ptr+=1

    if char == "<":
        if ptr > 0:
            ptr-=1

    if char == "+":
        arr[ptr]+=1
```

```

if char == "-":
    arr[ptr]-=1

if char == "[" and arr[ptr] == 0:
    i = bracMap[i]

if char == "]" and arr[ptr] != 0:
    i = bracMap[i]

if char == ".":
    print(arr[ptr])

i+=1

if char == ":":
    breakTimer -= 1
    print(arr[ptr])
    if breakTimer <= 0:
        i = len(code)

if char == ";":
    breakTimer -= 1
    if breakTimer <= 0:
        i = len(code)

if char == "@":
    print(ptr)

```

Bei einer so simplistischen esoterischen Programmiersprache muss ein Interpreter nicht sehr kompliziert sein, da er einfach der Reihe nach einfache Befehle ausführen muss. Der komplexeste Aspekt von Brainfuck sind "[" und "]", also Schleifen, da der Interpreter wissen muss, zu welcher Klammer er zurückspringen muss.

3. Code Golf und Project Euler

Code Golf ist eine Art von Wettbewerb, mit dem Ziel, einen bestimmten logischen Vorgang so einfach und schnell wie möglich in einer Programmiersprache umzusetzen. Der Name "Code Golf" bezieht sich darauf, dass das Programm mit den wenigsten Buchstaben bzw. verwendeten Tastenschlägen gewinnt. Code Golf Wettbewerbe werden meistens in Online-Foren wie zum Beispiel „StackExchange“ organisiert.

"Playing Perl golf (fewest (key)strokes wins[sic]) with people who have lots of experience is fine, but it's not going to help much for people who are still trying to get the hang of it."

(Quelle 16, Google Groups 1999, Increasing a value in a slice, Greg Bacon)

3.1 Über Project Euler

Projekt Euler bzw. Project Euler ist eine Serie von mathematischen und informatischen Herausforderungen. Um eine Herausforderung zu lösen werden fortgeschrittene mathematische Kenntnisse benötigt. Außerdem werden Programmierfähigkeiten unter Beweis gestellt. Speziell bei esoterischen Programmiersprachen existiert die zusätzliche Herausforderung, eine Aufgabenstellung trotz den Limitationen der Sprache zu lösen.

"Project Euler exists to encourage, challenge, and develop the skills and enjoyment of anyone with an interest in the fascinating world of mathematics."

(Quelle 17, Project Euler 2015, About Project Euler)

3.2 Problem Nummer 2

Die zweite in den Project Euler Archiven gelistete Herausforderung ist die Programmierung eines Algorithmus, der alle Fibonacci-Zahlen mit einem Wert der geringer als 4 Millionen ist berechnet. Außerdem soll das Programm jede gerade Zahl dieser Sequenz ausgeben.

Herausforderungen

Berechnung der Fibonacci Zahlen

Aufgrund des minimalistischen Designs der Programmiersprache stehen viele gewohnte Werkzeuge normaler Programmiersprachen nicht zur Verfügung. Zur Berechnung von Fibonacci Zahlen würde in anderen Sprachen zum Beispiel eine while-Schleife verwendet werden. Um in Brainfuck diese Werkzeuge aus anderen Programmiersprachen zu emulieren, müssen oft um einiges komplexere Alternativen verwendet werden. Um die Funktionalität einer for-Schleife nachzuahmen verwendet das Programm eine *[]-Schleife* die nach jeder Iteration der Schleife eine bestimmte Zelle im „Memory Tape“ um 1 verringert. Sobald diese Zelle den Wert null annimmt, bricht die Schleife ab und das Programm beendet sich.

In der Schleife werden pro Iteration jeweils die nächsten zwei Zahlen in der Fibonacci-Reihe berechnet, indem zwei Zahlen abwechselnd miteinander addiert werden.

Modulo Rechnung

Um festzustellen ob eine berechnete Fibonacci Zahl gerade ist, wird ein Algorithmus zum berechnen des Modulo der Zahl mit 2 benötigt. Dafür wird der DivMod-Algorithmus verwendet. Dieser Algorithmus wird benötigt, um in Brainfuck den Modulo bzw. die Division von 2 Zahlen zu berechnen.

```
[ ->+>-[>+>>]>+[[-<+>]>+>>]<<<<<<]
```

Für die Berechnung des Modulo werden 5 Zellen benötigt. Die Zelle n beinhaltet die Zahl die dividiert werden soll, also die gerade berechnete Fibonacci-Zahl. In der Zelle d wird der Divisor gespeichert. Um zu testen, ob eine Zahl gerade ist, wird sie mit 2 dividiert und der Rest der Division überprüft. Der Algorithmus speichert die Ergebnisse der Berechnungen in den nächsten 3 Zellen. Ist das Ergebnis 0, ist die Zahl gerade.

(Quelle 18, esolangs 2005, Brainfuck Algorithms Divmod)

3.3 Das Programm

```

1:  +>+<>>>>>>>>>+++++
2:  [<<<<<<<<<
3:  [>>+<<-]
4:  >>[<+<+>>-]<
5:  [>+>>+<<<-]
6:  >[<+>-]>>>[-]++<
7:  [->-[>+>>]>+[[-<+>]>+>>]<<<<<]>>-[<<<<<.>>>>>+][-]<[-]<<<<
8:  [>+<-]
9:  >[<+<+>>-]<<
10: [>+>>+<<<<-]
11: >>[<<+>>-]>>>[-]++<
12: [->-[>+>>]>+[[-<+>]>+>>]<<<<<]>>-[<<<<<.>>>>>+][-]<[-]<<<<
13: >>>>>>>>-]

```

Datenstruktur

Bei „größeren“ Programmen in Brainfuck empfiehlt es sich, das Memory Tape klar zu strukturieren und einzuteilen, welche Bedeutung Zellen haben werden. Dabei kann sich die Bedeutung bestimmter Zellen während des Ablaufs des Programms ändern.

Benötigte Zellen

Zur Berechnung der Fibonacci-Zahlen werden drei Zellen benötigt. Die ersten zwei Zellen werden zum Speichern der Fibonacci-Zahlen verwendet, die immer gegenseitig addiert werden. Eine dritte Zelle wird als Zwischenspeicher für die Zahl verwendet, die zu der anderen addiert werden soll. Zellen 5 bis 9 werden benötigt, um den DivMod Algorithmus zu verwenden und die Ergebnisse der Berechnungen zu speichern. Die zehnte Zelle wird von der Hauptschleife verwendet, um mitzuzählen, wie viele Iterationen noch ablaufen müssen, bis die letzte gefragte Zahl berechnet ist. In Abbildung 3 ist diese Aufteilung zu sehen.

Zahl 1	Zahl 2	Kopie	n Dividend	d Divisor	d - n%d Ergebnis	n%d Ergebnis	n/d Ergebnis			Index
0	0	0	0	0	0	0	0	0	0	0

Abbildung 3

Erklärung

Zahl 1	Zahl 2	Kopie	n Dividend	d Divisor	d - n%d Ergebnis	n%d Ergebnis	n/d Ergebnis		Index
1	1	0	0	0	0	0	0	0	14

Abbildung 4

1: +>+<>>>>>>>>+++++

Die erste Zeile des Programms läuft im Gegensatz zu den anderen nur einmal ab. In ihr werden bestimmten Zellen im Array ihre anfänglichen Werte zugewiesen. Die Zellen „Zahl 1“ und „Zahl 2“ werden jeweils auf 1 gesetzt, um die Berechnung der Fibonacci-Folge zu beginnen. Die Zelle „Index“ bekommt den Wert 14. Dies lässt die Schleife im Programm 14 Mal laufen, die benötigte Anzahl an Iterationen um alle gesuchten Zahlen zu errechnen.

Zahl 1	Zahl 2	Kopie	n Dividend	d Divisor	d - n%d Ergebnis	n%d Ergebnis	n/d Ergebnis		Index
0	1	0	1	0	0	0	0	0	14

Abbildung 5

Zahl 1	Zahl 2	Kopie	n Dividend	d Divisor	d - n%d Ergebnis	n%d Ergebnis	n/d Ergebnis		Index
1	2	0	0	0	0	0	0	0	14

Abbildung 6

3: [>>+<<-]

4: >>[<+<+>>-]<

Um einen Wert in Brainfuck zu bewegen wird normalerweise eine einfache Schleife wie zum Beispiel „`[>+<-]`“ verwendet. Diese Schleife subtrahiert 1 von einer Zelle und addiert 1 zu einer anderen, bis der Wert von der ersten Zelle 0 ist. Um einen Wert zu kopieren, d.h.

ihn zu verdoppeln, muss er zuerst in eine andere Zelle bewegt werden, und dann zurück auf seine ursprüngliche Zelle sowie die Zellen, auf welche der Wert kopiert werden soll. Dieser Vorgang passiert in den Zeilen 3 und 4 meines Programms. Dies hat den Effekt, dass der Wert von Zahl 1 zu Zahl 2 addiert wird, um die nächste Zahl der Fibonacci-Folge zu berechnen.

Zahl 1	Zahl 2	Kopie		n Dividend	d Divisor	d - n%d Ergebnis	n%d Ergebnis	n/d Ergebnis		Index
1	0	2	0	2	0	0	0	0	0	14

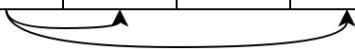


Abbildung 7

Zahl 1	Zahl 2	Kopie		n Dividend	d Divisor	d - n%d Ergebnis	n%d Ergebnis	n/d Ergebnis		Index
1	2	0	0	2	2	0	0	0	0	14




Abbildung 8

```
5: [>+>>+<<<-]
6: >[<+>-]>>>[-]++<
```

Derselbe Algorithmus wird in Zeile 5 und 6 verwendet. Hier wird das Ergebnis der Addition aus Zeilen 3 und 4, also die nächste Zahl in der Fibonacci-Folge, in die Zelle n kopiert. In Zeile 6 wird auch der Wert der Zelle d auf 2 gesetzt.

Zahl 1	Zahl 2	Kopie	n Dividend	d Divisor	d - n%d Ergebnis	n%d Ergebnis	n/d Ergebnis	Index	
1	2	0	0	2	2	0	1	0	14

Abbildung 9

7: [->-[>+>>]>[+[-<+>]>+>>]<<<<<]>>-[<<<<<.>>>>>+][-]<[-]<<<<

In der Aufgabenstellung wird nach geraden Fibonacci-Zahlen gefragt. Um herauszufinden, ob eine Zahl gerade ist, wird der Rest der Division mit 2 berechnet. Ist dieser 1, ist die Zahl ungerade, bei 0 ist die Zahl gerade. Falls die Zahl gerade ist, wird sie vom Programm ausgegeben.

Der zweite Teil der Schleife wiederholt dasselbe Prinzip, mit dem Unterschied, dass Zahl 2 zu Zahl 1 addiert wird, und Zahl 1 als nächste Zahl in der Fibonacci-Reihe verwendet wird.

Dieser Vorgang wird 17 Mal wiederholt, bis zur Fibonacci-Zahl 3.524.578, der letzten geraden Fibonacci-Zahl unter 4.000.000. Das erwartete Ergebnis der Berechnungen ist eine Menge aller geraden Fibonacci-Zahlen unter 4.000.000:

N	f(n)	n	f(n)	n	f(n)	n	f(n)	n	f(n)
1	1	8	21	15	610	22	17.711	29	514.229
2	1	9	34	16	987	23	28.657	30	832.040
3	2	10	55	17	1.597	24	46.368	31	1.346.269
4	3	11	89	18	2.584	25	75.025	32	2.178.309
5	5	12	144	19	4.181	26	121.393	33	3.524.578
6	8	13	233	20	6.765	27	196.418	34	5.702.887
7	13	14	377	21	10.946	28	317.811	35	9.227.465

Tabelle 5

Nach dem Ablauf des Programms in dem Python Brainfuck-Interpreter, wird diese Zahlensequenz ausgegeben, die mit dem erwarteten Ergebnis übereinstimmt. Das Ergebnis ist in Abbildung 10 zu sehen.

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
2
8
34
144
610
2584
10946
46368
196418
832040
3524578
```

Abbildung 10

4. BFJoust

4.1 Das Spiel

BFJoust ist ein von Kerim Aydin entworfenes Spiel, welches auf den Regeln des Spiels „Capture the Flag“ basiert. Die Regeln wurden zum ersten Mal am 6. Jänner 2009 veröffentlicht. Das Spiel wird von zwei Bots (geschrieben in Brainfuck) gespielt, die auf einem geteilten Array versuchen, die gegnerische Flagge von 127 auf 0 zu setzen. Um Bots für Menschen leichter verständlich zu machen wird eine Art „Preprocessor“ verwendet. Dieses Programm wandelt abgekürzte Programme in volle Brainfuck Programme um. Zum Abkürzen werden die Zeichen `()*{}%` verwendet. `(>+)*5` wird zum Beispiel zu `>+>+>+>+>+` verarbeitet.

(Quelle 19, agora-notary 2009, Brainfuck Joust)

4.2 Regeln

Zu Beginn des Spiels wird das „Schlachtfeld“ generiert. Es besteht aus einem Array mit einer zufälligen Länge zwischen 10 und 30 Elementen. Jedes Element ist ein 8 Bit Integer, und kann die Werte -127 bis 127 annehmen. Bei Spielstart werden die Flags auf 127 gesetzt, die restlichen Elemente auf 0. Ziel des Spiels bzw. der Bots ist es, die gegnerische Flagge auf 0 zu reduzieren. Der Bot gewinnt, falls die gegnerische Flagge für 2 Züge zu 0 gesetzt ist. Bewegt sich ein Bot über die gegnerische Flagge hinweg, verlässt also den Array, führt er keine weiteren Befehle mehr aus und wartet das Ende des Spiels ab. Falls ein Element mit dem Wert -127 reduziert wird, nimmt es den Wert 127 an, ein Element mit dem Wert 127 nimmt beim Inkrementieren den Wert -127 an. Die meisten Spielprogramme lassen die beiden Bots auf jeder möglichen Spielfeldlänge (10-30) einmal normal und einmal mit umgedrehter Polarität (d.h. + und - im Programm werden jeweils durch das andere ersetzt) spielen und ermitteln so den insgesamt besseren Bot.

(Quelle 20, esolangs 2009, BFJoust The current rules)

4.3 Strategien

Obwohl das Spiel aus einfachen Regeln aufgebaut ist, haben sich mit der Zeit viele verschiedene Strategien entwickelt.

4.3.1 Decoy

Ein Decoy ist ein Element, welches so verändert wird, dass der gegnerische Bot es fälschlich als Flagge identifiziert und auf 0 setzt. Durch diese Strategie wird der Bot verzögert und von der Flagge ferngehalten.

4.3.2 Rush

Ein Rush Bot versucht so schnell wie möglich zu der gegnerischen Flagge zu kommen, ohne Decoys zu setzten.

Einer der einfachsten Rush Bots ist `(>)*9([-].>)*21`. Dieser Bot überspringt die ersten 9 Elemente und setzt dann der Reihe nach jedes Element auf 0.

4.3.3 Poke

Das Ziel eines Poke Bots ist es, die Position des Gegners herauszufinden und Decoys direkt vor ihm aufzubauen. Der einfache Bot `(>[])*30` bewegt sich vorwärts bis er sich auf einem Element mit dem Wert `≠0` befindet und führt dann den Code in den eckigen Klammern aus. Ein einfacher Poke Bot würde beginnen, den Gegner dort aufzuhalten wo er sich gerade befindet.

4.3.4 Tripwire

Eine Tripwire ist eine Zelle die auf einen niedrigen Wert gesetzt wird. Der Bot, der die Zelle als Tripwire verwendet, wartet bis sie der andere Bot auf 0 gesetzt hat, und fährt erst dann mit seinem Programm fort. `+ []` ist eine einfache Tripwire. Ein Element wird auf 1 gesetzt, und der Bot wartet mit einer leeren Schleife darauf, dass der Wert der Zelle 0 annimmt, ausgelöst von dem gegnerischen Bot, der die Zelle auf 0 setzt. Sobald die Tripwire ausgelöst wurde, sind der gegnerische Bot und der Tripwire Bot auf derselben Zelle. Dadurch können zum Beispiel Decoys dem gegnerischen Bot genau in den Weg gesetzt werden.

4.3.5 Clear

Um zu gewinnen muss die gegnerische Flagge auf 0 gesetzt werden. Eine einfache Methode ist `[-]`. Dieser Code verringert eine Flagge bis sie den Wert 0 annimmt. `(>)*9([-]>)*21` ist ein kompletter Clear Bot. `[-]` wird "two-cycle" Clear genannt, da er für jede Verringerung des Wertes in einer Zelle zwei Schritte benötigt. Um den Vorgang zu beschleunigen wird die Schleife um das Minus entfernt. Eine solcher "one-cycle" Clear wäre zum Beispiel `(-)*128`. Da die Schleife fehlt, wird nicht nach jedem Schritt überprüft ob der Wert der Zelle 0 ist. Dadurch kann in manchen Fällen vom Programm „verpasst“ werden, dass die Zelle auf 0 gesetzt wurde, und der Wert immer weiter verringert werden, bis er nach 256 Schritten wieder 0 annimmt.

4.3.6 Wiggle Clear

Wiggle Clear ist eine der komplexesten Clear Methoden. Um Decoys zu verringern, können sie entweder reduziert oder inkrementiert werden. Bei Decoys ist immer eine dieser Methoden schneller, z.B. ist ein Decoy mit dem Wert -3 beim Inkrementieren in drei Schritten auf den Wert 0, wird das Decoy reduziert dauert derselbe Vorgang 252 Schritte. Um zu verhindern, dass Decoys in die „falsche“ Richtung verändert werden, gibt es Wiggle Clear. `([-{([+{[-]})%8}])%4>` ist ein einfacher Wiggle Clear. Der Preprocessor verarbeitet diesen Code zu `[-[-[-[-[+[+[+[+[+[+[+[[-]]]]]]]]]]>`. Bei diesem Wiggle Clear wird der Decoy zuerst um 4 verringert und anschließend um 8 erhöht. Nimmt das Element dabei 0 an, bewegt sich der Bot zum nächsten Element, ansonsten wird es ohne spezielle Methode weiter reduziert. Elemente mit Werten zwischen -4 und 4 werden nach dieser Methode sehr schnell auf 0 gesetzt.

(Quelle 21, esolangs 2011, BFJoust strategies)

4.4 helyx_FightAndFlight

```
1: (>)*8+(<)*7  
2: (  
3: (-)*13>  
4: (+)*13>  
5: )*4  
6: (([-{([+{[-]}])%8}])%4>)*21
```

4.4.1 Verwendete Strategien

Mein Bot verwendet eine Kombination von unterschiedlichen Strategien, die speziell angepasst wurden um miteinander zu arbeiten.

Decoy

In der ersten Zeile bewegt sich der Bot 8 Elemente auf den Gegner zu, setzt ein Decoy und bewegt sich wieder auf das Element vor der eigenen Flagge zurück. Dieses Decoy ist eine einfache Verteidigung gegen Poke Strategien, da der Gegner die Position des Bots viel näher einschätzt als er eigentlich ist.

In den Zeilen 2 bis 5 bewegt sich der Bot 8 Elemente vorwärts und setzt weitere Decoys mit den Werten 13 bzw. -13.

Rush

Nachdem die Decoys gesetzt worden sind, beginnt der Bot mit einem Rush, und reduziert jedes Element auf 0. Im Idealfall wird der gegnerische Bot durch die gesetzten Decoys lange genug aufgehalten, um die Flagge nicht rechtzeitig zu erreichen.

Wiggle Clear

Um den Rush zu beschleunigen, werden Elemente mit dem Wiggle Clear Algorithmus reduziert. Dieser Vorgang wird wiederholt, bis die Gegnerische Flagge auf 0 gesetzt worden ist.

4.4.2 Ergebnisse gegen andere Bots

Die meisten Bots werden nach dem Schema [Nickname des Programmierers]_[Bot Name]. Mein Bot ist helyx_FightAndFlight benannt. Die Website Zem.fi bietet eine Web Applikation an, mit der BFJoust Bots getestet werden können. Die Ergebnisse von Spielen gegen 5 andere Bots sind für meinen bewusst einfach gestalteten Bot überraschend gut ausgefallen.

Moop_Alternator

```
1: (>+>-)*4
2: >+
3: (>[-][.])*21
```

(Quelle 22, StackExchange 2014, BFJoust bots Alternator)

Moop_Alternator ist darauf ausgelegt, eine Schwäche der Clear-Methode eines gegnerischen Bots auszunutzen. Es wird angenommen, dass ein Bot mit seinem Clear-Algorithmus alle Decoys entweder durch Inkrementieren oder Dekrementieren auf 0 setzt. Vom Bot Moop_Alternator werden Decoys gesetzt, die abwechselnd die Werte 1 und -1 annehmen. Viele Clear-Algorithmen brauchen durchschnittlich 128 Züge pro Zelle, da jede zweite Zelle einen ganzen Additionszyklus vollenden muss, bis sie wieder den Wert 0 annimmt. Da helyx_FightAndFlight Wiggle Clear verwendet, werden die Decoys von diesem Bot in sehr geringer Zeit beseitigt, und die gegnerische Flagge kann in den meisten Runden schnell erreicht werden.

35 von 42 Runden gewonnen.

ccarton_AnybodyThere?

```
1: >>>+<(+)*5<(-)*5>>
2: [
3: (>[([(+)*10[-])>)*29])*4
4: +
5: <<<<
6: [
7: >>>
8: (+)*5<(-)*5
9: <<[-]
10:]
11: >>>>
12: ]
13: ([-[(+)*10[-]])>)*29
```

(Quelle 23, Github 2014, BFJoust bots AnybodyThere?)

Dieser Bot setzt zuerst in den ersten 2 Zellen kleine Decoys, und verwendet dann die dritte Zelle als Tripwire. Falls die Tripwire nicht ausgelöst wurde, bewegt sich der Bot 4 Felder vorwärts und setzt weiter Decoys. Dieser Vorgang wiederholt sich, und wird nach jeder Iteration um 4 Zellen vorverlegt. Sobald ein Anzeichen des gegnerischen Bots gefunden wurde (entweder ein eigenes Decoy, das auf 0 gesetzt wurde, oder ein Decoy vom Gegner), beginnt der Bot mit einem einfachen Rush. Dieser Bot setzt sehr viele Decoys, die erst auf längeren Memory-Tapes wirklich nützlich sind. `helyx_FightAndFlight` verliert erst ab einer Spielfeldlänge von 25 Zellen konsistent gegen den Bot, und gewinnt deswegen das Spiel.

31 von 42 Runden gewonnen.

weston_MickeyV4

```
1: ++>----->->---<<----->----->->
2: ---->----->>--->-----<-----
3: -----<-----<-<<----->-----
4: -<-->----->----->----->-----
5: ----->----->----->-----[>
6: [--[-[+]]]>[--[+]]-]-----[>[--[-[+]]]>[
7: --[+]]-]<--<----->----->-----
8: --[>[--[-[+]]]>[--[+]]-]<--<-----
9: ----->----->->-<-----
```

(Quelle 24, StackExchange 2014, BFJoust bots MickeyV4)

MickeyV4 wurde mithilfe eines genetischen Algorithmus entwickelt. Bei einem genetischen Algorithmus werden Prinzipien aus der Evolution auf Programme angewandt, und so durch Mutationen ein Programm von selbst weiterentwickelt. Bei diesem Algorithmus wurde ein zufällig generierter Bot, also eine Reihe von zufälligen Befehlen, gegen 60 andere Bots getestet. Nach einem Durchgang ("Generation") wird das Programm zufällig verändert ("Mutation"). Mutationen, die den Bot verbessern werden behalten, schlechte Mutationen verworfen. Dieser Bot ist das Ergebnis von 1400 Generationen. Da er zufällig generiert wurde, besitzt er keine wirkliche Strategie und ist schwer zu verstehen. Von den ersten 21 Runden konnte mein Bot nur eine gewinnen. Bei umgedrehter Polarität werden jedoch die „Decoys“ von MickeyV4 nutzlos, da alle positive Werte haben, und von meinem Bot sehr schnell beseitigt werden. Deswegen gewann mein Bot ab einer Spielfeldlänge von 21 Zellen jede Runde.

22 von 42 Runden gewonnen.

```

1: >((-)*18>)*2
2: (->)*6
3: [
4:   +[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+
5:   (-)*18
6:   -[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-
7:   (-)*107
8:   [-.]
9:   ]]]]]]]]]]]]]]]]]]]]]]
10: ]]]]]]]]]]]]]]]]]]]]]]
11: ]
12: +
13: ([>
14:   [
15:     +[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+[+
16:     (-)*18
17:     -[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-[-
18:     (-)*107
19:     [-.]
20:     ]]]]]]]]]]]]]]]]]]]]]]
21: ]]]]]]]]]]]]]]]]]]]]]]
22:   ]
23: --
24: ]
25: -
26: )*5
```

BurlyBalder ist in zwei Phasen aufgeteilt. Am Anfang des Spiels setzt er zwei Decoys nahe an seiner Flagge und fängt dann an sich schnell auf die gegnerische Flagge zuzubewegen. Die erste Schleife beinhaltet einen Clear Algorithmus für den Fall, dass das Memory Tape 10 Zellen lang ist. Der restliche Code ist ein modifizierter Wiggle Clear, der nach jedem Decoy, das auf 0 gesetzt wurde, ein eigenes Decoy mit dem Wert -2 hinterlässt.

30

LymiaAliysia_NyurokiMagicalFantasy

```
// Copyright (C) 2014 Lymia Aluysia <lymiahugs@gmail.com>
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is furnished
// to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
```

```
raw +margins "Nyuroki Magical Fantasy by Lymia Aluysia
              Released under the terms of MIT license
              "
```

```
>>>>>>>+<--<+<--
// 61 of each. Written out here to workaround a rather serious Arena.py bug.
<+++++
<-----
<-----
<+++++
<(-)*(128-109) // Change our home flag to throw off simpler bots.
```

```
// And, then let's just rush!
@wobbleClear($neg0, $pos0, $neg1, $pos1, $neg2, $pos2) {
  @wobbleClearPos($adjustment, $count) {
    if($count == 0) {
      (-)*$adjustment
      @clearBody()
    } else {
      +[
        @wobbleClearPos($adjustment, $count - 1)
      ]
    }
  }
  @wobbleClearNeg($adjustment, $count) {
    if($count == 0) {
      (+)*$adjustment
      @clearBody()
    } else {
```

```

        -[
            @wigggleClearNeg($adjustment, $count - 1)
        ]
    }
}

@clearBody() {
    @clearBody() {
        @clearBody() {
            @clearBody() {
                @clearBody() {
                    @clearBody() {
                        (+)*82 [[+..].]
                        @break() // In case anyone tries to beat us by locking us in our loop.
                    }
                    @wigggleClearPos(0, $pos2 - $pos1)
                }
                @wigggleClearNeg($neg2 + $pos1, $neg2 - $neg1)
            }
            @wigggleClearPos($pos1 + $neg1, $pos1 - $pos0)
        }
        @wigggleClearNeg($neg1 + $pos0, $neg1 - $neg0)
    }
    @wigggleClearPos($pos0 + $neg0, $pos0)
}
@wigggleClearNeg($neg0, $neg0)
}
(>)*8 // Rule of 9
(
    > callcc(@break) {
        // Clear counts shamelessly adjusted to hill.
        [@wigggleClear(3, 3, 13, 11, 30, 32)]
    }
    --
)*21

// ... oh screw it. Not worth the binary size.
// Used to be (-...-...)*-1, but, that only earned me /1/ win. Not a big deal.
terminate

```

(Quelle 26, StackExchange 2014, BFJoust bots NyurokiMagicalFantasy)

Dieser Bot wurde nicht in Brainfuck geschrieben, sondern in einer eigen für BFJoust geschriebenen Programmiersprache namens JoustExt. Bots, die in dieser Programmiersprache geschrieben sind, müssen anschließend von einem „Transpiler“ zu Brainfuck übersetzt werden, um bei BFJoust verwendet zu werden. Die übersetzte Version des Bots in Brainfuck ist in Abbildung 11 zu sehen.

5. Fazit

Können Probleme mit esoterischen Programmiersprachen gelöst werden?

Ja.

Ist es sinnvoll?

Vielleicht.

Das minimalistische Design vieler esoterischer Programmiersprachen ermöglicht oft das schnelle Umsetzen von einfacheren mathematischen Berechnungen und anderer Algorithmen. Diese Programme sind auf das Wesentliche konzentriert, ohne komplexe „User Interfaces“ und andere Ablenkungen. Sie sind eine Umsetzung des Lösungsansatzes, und nichts Anderes. Durch das Schreiben solcher Programme versteht der Programmierer oft erst die wirkliche Funktionsweise eines Algorithmus, und, aufgrund der sehr einfachen Befehle der Programmiersprache, die Berechnung, die wirklich auf dem Prozessor abläuft. Das Programm läuft ohne jegliche Abstraktionsschichten. Mithilfe dieser neuen Perspektive fällt es leichter, einen Algorithmus abzuändern oder zu optimieren.

Ein anderer Aspekt der Programmierung mit esoterischen Programmiersprachen ist die Kombination von Programmieren und Unterhaltung. Bei Programmen wie den Bots, die bei BFJoust verwendet werden, steht das Spiel an erster Stelle. Um einen Bot zu programmieren sind aber Kenntnisse über viele mathematische Konzepte erforderlich. Durch das Programmieren eines solchen Bots lernt man nebenbei Mathematische Konzepte und Methoden zur effizienten Umsetzung eines Algorithmus.

Selbst wenn das Schreiben eines Programmes in einer esoterischen Programmiersprache nur einen geringen Wert für kommerzielle Software hat, ist es für den Programmierer eine Aktivität, die Vorgänge und Berechnungen auf einer sehr minimalistischen Ebene verständlicher macht.

Meiner Meinung nach ist das Schreiben von Programmen in esoterischen Programmiersprachen ein oft unterhaltsamer Zeitvertreib, der manchmal auch das Verständnis von logischen Konzepten stärkt.

Literaturverzeichnis

Quelle 1: Esolangs: Brainfuck. 2005. URL: <https://esolangs.org/wiki/Brainfuck> (Zugegriffen 26.1.2016)

Quelle 2: Esolangs: Brainfuck, Language Overview. 2005. URL: https://esolangs.org/wiki/Brainfuck#Language_overview (Zugegriffen 26.1.2016)

Quelle 3: Esolangs: Brainfuck, Examples. 2005. URL: <https://esolangs.org/wiki/Brainfuck#Examples> (Zugegriffen 26.1.2016)

Quelle 4: Esolangs: Befunge, Language Overview. 2005. URL: https://esolangs.org/wiki/Befunge#Language_overview (Zugegriffen 26.1.2016)

Quelle 5: Esolangs: Befunge, Examples. 2005. URL: <https://esolangs.org/wiki/Befunge#Examples> (Zugegriffen 26.1.2016)

Quelle 6: catseye: befbef.bf. 1997. URL: <http://catseye.tc/view/befunge-93/eg/befbef.bf> (Zugegriffen 26.1.2016)

Quelle 7: Esolangs: Intercal, Command syntax. 2005. URL: https://esolangs.org/wiki/Intercal#Command_syntax (Zugegriffen 26.1.2016)

Quelle 8: Wikipedia: Intercal, Hello world. 2015. URL: https://en.wikipedia.org/wiki/INTERCAL#Hello.2C_world (Zugegriffen 26.1.2016)

Quelle 9: Esolangs: Magicard!. 2013, URL: <http://esolangs.org/wiki/Magicard!> (Zugegriffen 26.1.2016)

Quelle 10: Esolangs: Magicard!, Instruction Set. 2013, URL: http://esolangs.org/wiki/Magicard!#Instruction_Set (Zugegriffen 26.1.2016)

Quelle 11: Esolangs: Magicard!, Examples. 2013, URL: <http://esolangs.org/wiki/Magicard!#Examples> (Zugegriffen 26.1.2016)

Quelle 12: Github: ArnoldC, Readme. 2013. URL: <https://github.com/lhartikk/ArnoldC> (Zugegriffen 26.1.2016)

Quelle 13: Esolangs: Seed, Writing programs. 2009. URL: https://esolangs.org/wiki/Seed#Writing_programs (Zugegriffen 26.1.2016)

Quelle 14: Esolangs: Seed, Examples. 2009. URL: <https://esolangs.org/wiki/Seed#Examples> (Zugegriffen 26.1.2016)

Quelle 15: Esolangs: Byter. 2008. URL: <https://esolangs.org/wiki/Byter> (Zugegriffen 26.1.2016)

Quelle 16: Greg Bacon: Increasing a value in a slice. 1999. URL: <https://groups.google.com/forum/#!msg/comp.lang.perl.misc/zYRU5D2IyuI/II0sSTTEl3sJ> (Zugegriffen 26.1.2016)

Quelle 17: Project Euler: Project Euler. 2015. URL: <http://www.projecteuler.net> (Zugegriffen 26.1.2016)

Quelle 18: Esolangs: Brainfuck Algorithms, Divmod. 2005. URL: https://esolangs.org/wiki/Brainfuck_algorithms#Divmod_algorithm (Zugegriffen 26.1.2016)

Quelle 19: Agora Notary: Brainfuck Joust. 2009. URL: <http://agora-notary.wikidot.com/deleted:brainfuck-joust> (Zugegriffen 26.1.2016)

Quelle 20: Esolangs: BFJoust, The current rules. 2009. URL: https://esolangs.org/wiki/BF_Joust#The_Current_Rules (Zugegriffen 26.1.2016)

Quelle 21: Esolangs: BFJoust Strategies. 2011. URL: https://esolangs.org/wiki/BF_Joust_strategies (Zugegriffen 26.1.2016)

Quelle 22: StackExchange: BFJoust Bots, Alternator. 2014. URL: <http://codegolf.stackexchange.com/a/36659> (Zugegriffen 26.1.2016)

Quelle 23: StackExchange: BFJoust Bots, AnybodyThere?. 2014. URL: <http://codegolf.stackexchange.com/a/36706> (Zugegriffen 26.1.2016)

Quelle 24: StackExchange: BFJoust Bots, BurlyBladerV3. 2014. URL: <http://codegolf.stackexchange.com/a/36782> (Zugegriffen 26.1.2016)

Quelle 25: StackExchange: BFJoust Bots, MickeyV4. 2014. URL: <http://codegolf.stackexchange.com/a/37137> (Zugegriffen 26.1.2016)

Quelle 26: StackExchange: BFJoust Bots, NyurokiMagicalFantasy. 2014. URL: <http://codegolf.stackexchange.com/a/37385> (Zugegriffen 26.1.2016)

Abbildungsverzeichnis

Alle in der Arbeit verwendeten Abbildungen wurden vom Autor selbst erstellt.

Name: Jakob Kuen

Selbstständigkeitserklärung

Ich erkläre, dass ich diese vorwissenschaftliche Arbeit eigenständig angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Ort, Datum

Unterschrift