

Informatik I: Einführung in die Programmierung

16. Fingerübung: Ein Interpreter für Brainf*ck

Albert-Ludwigs-Universität Freiburg



Bernhard Nebel

1. Dezember 2017

1 Motivation



- Motivation
- Program-miersprache
- Beispiele
- Semantik
- Interpreter-Design
- Ausblick
- Zusammen-fassung

1. Dezember 2017

B. Nebel – Info I

3 / 54

Brainf*ck: Eine minimale Sprache



- Jeder *Informatiker* sollte **mindestens 2 Programmiersprachen** beherrschen!
- Python, C++, Scheme, ...
- Wir wollen heute eine minimale Programmiersprache kennen lernen, ...
- ... uns freuen, dass wir bisher eine sehr viel komfortablere Sprache kennen lernen durften,
- ... dazu einen **Interpreter** bauen,
- ... der **Daten-getriebene Programmierung** einsetzt.
- Außerdem sehen wir Dictionaries und Exceptions im Einsatz.
- Heute: Keine *rekursiven* Datentypen oder Funktionen!

- Motivation
- Program-miersprache
- Beispiele
- Semantik
- Interpreter-Design
- Ausblick
- Zusammen-fassung

1. Dezember 2017

B. Nebel – Info I

4 / 54

Entstehungsgeschichte



- Urban Müller hat die Sprache 1993 beschrieben, die 8 verschiedene Befehle kennt, und einen Compiler mit weniger als 200 Byte dafür geschrieben
- Die Sprache wird gerne für „Fingerübungen“ im Kontext Interpreter/Compiler benutzt.
- Obwohl minimal, ist die Sprache doch mächtig genug, dass man alle *berechenbaren Funktionen* implementieren kann: Sie ist **Turing-vollständig**.
- Gehört zur Familie der „esoterischen“ Programmiersprachen. Andere Vertreter z.B. *Whitespace*, *Chef*, *TrumpScript* und *Shakespeare*.

- Motivation
- Program-miersprache
- Beispiele
- Semantik
- Interpreter-Design
- Ausblick
- Zusammen-fassung

1. Dezember 2017

B. Nebel – Info I

5 / 54

2 Programmiersprache



- Befehle
- Schleifen

Motivation
Program-
miersprache
Befehle
Schleifen
Beispiele
Semantik
Interpreter-
Design
Ausblick
Zusammen-
fassung

Programmiersprache und Berechnungsmodell



- **Programme** bestehen aus einer Abfolge von ASCII-Zeichen (Unicode-Wert 0 bis 127).
- Bedeutungstragend sind aber nur die acht Zeichen:
`< > + - . , []`
Alles andere ist Kommentar.
- Das Programm wird Zeichen für Zeichen abgearbeitet, bis das Ende des Programms erreicht wird.
- Es gibt einen ASCII-Eingabestrom und einen ASCII-Ausgabestrom (normalerweise die Konsole)
- Die **Daten** werden in einer Liste gehalten: `data`. Wir reden hier von **Zellen**.
- Es gibt einen **Datenzeiger**, der initial 0 ist: `ptr`.

Motivation
Program-
miersprache
Befehle
Schleifen
Beispiele
Semantik
Interpreter-
Design
Ausblick
Zusammen-
fassung

Die Befehle



Die **aktuelle Zelle** ist das Listenelement, auf die der Datenzeiger zeigt: `data[ptr]`.

- > Bewege den Datenzeiger nach rechts `ptr += 1`.
- < Bewege den Datenzeiger nach links `ptr -= 1`.
- + Erhöhe den Wert in der aktuellen Zelle:
`data[ptr] += 1`.
- Erniedrige den Wert in der aktuellen Zelle:
`data[ptr] -= 1`.
- Gebe ein ASCII-Zeichen entsprechend dem Wert in der aktuellen Zelle aus:
`print(chr(data[ptr]), end='')`.
- Lese ein ASCII-Zeichen und lege den Wert in der aktuellen Zelle ab:
`data[ptr] = inp[0]; del inp[0]`.

Motivation
Program-
miersprache
Befehle
Schleifen
Beispiele
Semantik
Interpreter-
Design
Ausblick
Zusammen-
fassung

Ein Beispiel



Ein Programm ohne Verzweigungen und Schleifen, das einen Großbuchstaben in den entsprechenden Kleinbuchstaben übersetzt.

`konv.b`

Lese ein Zeichen (Annahme: Grossbuchstabe)

```
,  
Konvertiere in Kleinbuchstabe  
+++++  
Gebe das Zeichen aus
```

```
.  
Und hier ist das Programm zu Ende
```

Probiere aus auf: <https://fatiherikli.github.io/brainfuck-visualizer/>

Motivation
Program-
miersprache
Befehle
Schleifen
Beispiele
Semantik
Interpreter-
Design
Ausblick
Zusammen-
fassung

- Aus „normalen“ Programmiersprachen kennen wir die while-Schleife.
- Diese Rolle spielt in Brainf*ck das Paar []:
 - [Falls Inhalt der aktuellen Zelle = 0 ist (data[ptr] == 0), dann springe zum Befehl nach der **zugehörigen schließenden** Klammer (beachte Klammerungsregeln). Ansonsten setze die Ausführung mit dem Befehl nach der öffnenden Klammer fort.
 -] Springe zur **zugehörigen öffnenden** Klammer.

Motivation
 Programmiersprache
 Befehle
 Schleifen
 Beispiele
 Semantik
 Interpreter-Design
 Ausblick
 Zusammenfassung

- Schleife
- Hello World

Motivation
 Programmiersprache
 Beispiele
 Schleife
 Hello World
 Semantik
 Interpreter-Design
 Ausblick
 Zusammenfassung

loop.b

```
++++++      set cell #0 to 6
[ > ++++++  add 8 to cell #1
  < -       decrement loop counter cell #0
]
> +        add another 1 to cell #1
.          print ASCII 49 = '1'

-          now cell #1 is '0'
< ++++++  set cell #0 to 8
[ > .      print ASCII 48 = '0'
  < -     decrement loop counter (cell #0)
]
```

Ausgabe: 100000000

Motivation
 Programmiersprache
 Beispiele
 Schleife
 Hello World
 Semantik
 Interpreter-Design
 Ausblick
 Zusammenfassung

hello.b - Part 1

```
+++++ +++++ initialize counter (cell #0) to 10
[
  > +++++ ++      use loop to set 70/100/30/10
  > +++++ +++++  add 7 to cell #1
  > +++          add 10 to cell #2
  > +           add 3 to cell #3
  > +           add 1 to cell #4
  <<<< -        decrement counter (cell #0)
]
> ++ .          print 'H'
> + .          print 'e'
+++++ ++ .     print 'l'
.              print 'l'
+++ .          print 'o'
```

Motivation
 Programmiersprache
 Beispiele
 Schleife
 Hello World
 Semantik
 Interpreter-Design
 Ausblick
 Zusammenfassung

Hello World (2)

hello.b - Part 2

```
> ++ .          print ' '
<< +++++ +++++ +++++ . print 'W'
> .            print 'o'
+++ .         print 'r'
----- - .   print 'l'
----- --- . print 'd'
> + .         print '!'
> .           print '\n'
```

Motivation
Program-
miersprache
Beispiele
Schleife
Hello World
Semantik
Interpreter-
Design
Ausblick
Zusammen-
fassung

Programmier-Pattern (1)

- Die Sprache ist sehr arm, aber man sieht, wie man bestimmte Dinge realisieren kann.
 - Zuweisung von Konstanten an Variable (ggfs. durch Schleifen) ist einfach (wenn man voraussetzt, dass die Variable den Wert 0 hat): `+++ ...`
 - Auf Null setzen (negative Werte sollten nicht auftreten!): `[-]`
 - Übertragen des positiven Wertes von der aktuellen Zelle zu einer anderen Zelle, (mit gegebenem Abstand, z.B. +3), wenn diese 0 ist: `[->>> + <<<]`
 - (Destruktive) Addition ist ebenfalls einfach (transferieren, wenn initialer Inhalt des Ziels der eine Summand ist).
 - Übertragen in zwei Zellen: `[->>>+>><<<<]`
 - Dann kann man auch einen Wert *kopieren*: Erst in zwei Zellen transferieren, dann den einen Wert zurück transferieren.

Motivation
Program-
miersprache
Beispiele
Schleife
Hello World
Semantik
Interpreter-
Design
Ausblick
Zusammen-
fassung

Programmier-Pattern (2)

■ Kontrollstrukturen und logische Operatoren:

- *If*-Anweisung ($x \neq 0$):
 - Benutze Schleife und setze die Test-Variable am Ende auf Null (ist destruktiv für die getestete Variable!)
 - Annahme, Testvariable ist aktuelle Zelle: `[... [-]`
- Für die logischen Operatoren sei 0 *False*, alles andere *True*.
- Logisches *and*:
 - Setze Ergebnisvariable auf Null. Dann ein If-Statement über dem ersten Operanden, in dem der zweite Operand auf die Ergebnisvariable transferiert wird.
 - Annahme, Linker Op. aktuell, rechter Op. +1, Ergebnis +2:
`>>[-]<< [> [-> + <] [-]] >>`
- Logisches *or*: Transferiere beide Operanden zur Ergebnisvariable.
- Logisches *not*: Setze Ergebnisvariable auf 1. Dekrementiere Ergebnisvariable in einem If-Statement, das die Eingangsvariable abfragt.

Motivation
Program-
miersprache
Beispiele
Schleife
Hello World
Semantik
Interpreter-
Design
Ausblick
Zusammen-
fassung

Programmier-Pattern (3)

- Vergleiche
 - Vergleich zweier Zellen/Variablen:
 - Dekrementieren beider Variablen, bis eine der Variablen Null wird.
 - Falls beide Null sind, waren die Werte gleich, ansonsten entsprechend.
 - Einfacher Vergleich mit einer Konstanten:
 - Initialisiere Hilfsvariable mit 1, ziehe die Konstante mit Folge von Minuszeichen ab, starte Schleife, dekrementiere Hilfsvariable, dann addiere auf ursprüngliche Zelle die Konstante drauf, danach setze auf Null.
 - `>[-]+< -...- [->+...+ [-]] >`
- Weitere Tipps:
 - `http://www.iwriteiam.nl/Ha_bf_intro.html`
 - Aufbauend auf diese Ideen könnte man hergehen, und für alle Konstrukte eine Übersetzung in Brainf*ck vorsehen.
 - Letztendlich ist dies etwas, was *Compilerbauer* machen.

Motivation
Program-
miersprache
Beispiele
Schleife
Hello World
Semantik
Interpreter-
Design
Ausblick
Zusammen-
fassung

- Offene Fragen
- Portabilität

- Motivation
- Program-
miersprache
- Beispiele
- Semantik
 - Offene Fragen
 - Portabilität
- Interpreter-
Design
- Ausblick
- Zusammen-
fassung

Short: 240 byte compiler. Fun, with src. OS 2.0
 Uploader: umueller amiga physik unizh ch
 Type: dev/lang
 Architecture: m68k-amigaos

The brainfuck compiler knows the following instructions:

| Cmd | Effect |
|-----|---------------------------------------|
| --- | ----- |
| + | Increases element under pointer |
| - | Decreases element under pointer |
| > | Increases pointer |
| < | Decreases pointer |
| [| Starts loop, flag under pointer |
|] | Indicates end of loop |
| . | Outputs ASCII code under pointer |
| , | Reads char and stores ASCII under ptr |

Who can program anything useful with it? :)

Leider lässt die Angabe der Semantik einige Fragen offen.

- 1 **Zellgröße:** In der ursprünglichen Implementation 1 Byte (= 8 Bits) entsprechend den Zahlen von 0...255. Andere Implementationen benutzen aber auch größere Zellen.
- 2 **Größe der Datenliste:** Ursprünglich 30000. Aber auch andere Größen sind üblich. Manche Implementationen benutzen nur 9999, andere erweitern die Liste auch dynamisch, manchmal sogar links (ins Negative hinein).
- 3 **Zeilenendezeichen:** \n oder \r\n? Hier wird meist die Unix-Konvention verfolgt, speziell da C-Bibliotheken diese Übersetzung unter Windows unterstützen.
- 4 **Dateiende (EOF):** Hier wird beim Ausführen von , entweder 0 zurückgegeben, die Zelle wird nicht geändert, oder es wird (bei Implementationen mit größeren Zellen) -1 zurück gegeben.
- 5 **Unbalancierte Klammern:** Das Verhalten ist undefiniert!

- Motivation
- Program-
miersprache
- Beispiele
- Semantik
 - Offene Fragen
 - Portabilität
- Interpreter-
Design
- Ausblick
- Zusammen-
fassung

- Alle Programmiersprachen haben mit diesen oder ähnlichen Problemen zu kämpfen.
- Speziell der Bereich der darstellbaren Zahlen ist ein Problem.
- Oft wird festgelegt, dass es **Implementations-abhängige** Größen und Werte gibt (z.B. max. Größe einer Zahl).
- Zudem lässt man oft **Freiheiten bei der Implementation** zu (z.B. Reihenfolge von Keys in Dicts).
- Außerdem gibt es immer Dinge, die außerhalb der Spezifikation einer Sprache liegen (z.B. Verhalten bei unbalancierten Klammern).
- Hier ist das **Verhalten undefiniert**, aber idealerweise wird eine Fehlermeldung erzeugt (statt erraticem Verhalten).

- Motivation
- Program-
miersprache
- Beispiele
- Semantik
 - Offene Fragen
 - Portabilität
- Interpreter-
Design
- Ausblick
- Zusammen-
fassung

Implikationen für einen Interpreter



- In einem sehr Ressourcen-beschränktem Kontext (z.B. Mikrocontroller) gibt man die Beschränkungen (z.B. Zellengröße und -anzahl) vor ... und vertraut darauf, dass der Benutzer sie einhält.
- Will man hohe Flexibilität zusichern baut man einen Interpreter, bei dem man verschiedene Möglichkeiten vorsieht, die dann der Benutzer steuern kann.
- Insbesondere
 - sollte man statt undefiniertem Verhalten eine Fehlermeldung erzeugen;
 - und sowohl eingeschränkte (Zellgröße = 1Byte, 9999 Zellen) als auch liberale Interpretation erlauben (bignums, beliebig viele Zellen);
 - verschiedene EOF-Markierungen erlauben.

Motivation
Program-
miersprache
Beispiele
Semantik
Offene Fragen
Portabilität
Interpreter-
Design
Ausblick
Zusammen-
fassung

Implikationen für portable Brainf*ck-Programme



Will man Brainf*ck-Programme schreiben, die auf möglichst vielen Interpretern lauffähig sind, sollte man nur solche Sprachbestandteile nutzen, die auf allen Implementationen laufen:

- Bei Zellgröße nur ein Byte annehmen. Ggfs. sogar nur den Bereich von 0–127 nutzen, da es bei einer vorzeichenbehafteten Darstellung einen arithmetischen Überlauf geben könnte!
- Für die EOF-Markierung kann man jeweils zuerst die Zelle auf Null setzen und dann lesen. Damit bekommt man sowohl bei der Zurückgabe einer Null als auch bei der leeren Zurückgabe das gleiche Ergebnis.

Motivation
Program-
miersprache
Beispiele
Semantik
Offene Fragen
Portabilität
Interpreter-
Design
Ausblick
Zusammen-
fassung

5 Interpreter-Design



- Designkriterien
- Datenstrukturen
- I/O
- Ausnahmebehandlung
- Hauptfunktion
- Datengetriebene Programmierung
- Einfache Fälle
- I/O
- Schleifen

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schleifen
Ausblick
Zusammen-
fassung

Designkriterien



Egal, was für Software Sie schreiben, Ihre Lösungen können Sie immer anhand der folgenden Kriterien bewerten:

- **(Praktische) Effizienz:** Wie schnell läuft das Programm und wie viel Speicher erfordert es? Gibt es schnellere oder sparsamere Alternativen? Sollte uns hier *noch* nicht interessieren!
- **Skalierbarkeit:** Wie stark wächst Laufzeit und Speicherbedarf mit der Größe der Eingabe?
- **Eleganz:** Wie „schön“ sieht das Programm aus? Z.B. viele Einzelfälle versus eine generelle Lösung.
- **Lesbarkeit:** Wie einfach ist das Programm zu verstehen?
- **Wartbarkeit:** Wie einfach ist es, Fehler zu finden oder neue Funktionalität zu integrieren?

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schleifen
Ausblick
Zusammen-
fassung

- Welchen Datentyp sollen wir für die Darstellung des Brainf*ck-Programms wählen?
 - String?
 - Liste?
 - Tupel?
 - Rekursive Datenstruktur (organisiert entlang der Klammerstruktur)?
 - Dictionary? Wobei dann die jeweilige Stelle durch den Schlüssel beschrieben wird?
- Am besten wohl **String**! Wir müssen ja bloß auf einzelne Stellen zugreifen. Ändern brauchen wir im Programmtext ja nichts.

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

- Welchen Datentyp sollen wir für die Darstellung der Brainf*ck-Datenzellen wählen?
 - String?
 - Tupel?
 - Liste?
 - Dictionary? Wobei dann die jeweilige Stelle durch den Schlüssel beschrieben wird?
- **Dict** ist wohl am bequemsten, da wir unbenutzte Zellen einfach initialisieren können.
- Listen wären etwas schneller, aber man müsste Bereich vorgeben oder dynamisch erweitern.

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Wir haben es mit drei Ein-/Ausgabeströmen zu tun:

- 1 Das Programm: Sollte einmal eingelesen und dann verarbeitet werden.
 - 2 Eingabestrom: Sollte Datei oder Konsoleingabe sein können.
 - 3 Ausgabestrom: Sollte ebenfalls Datei oder Konsolenausgabe sein.
- Das Modul `sys` stellt zwei Datei-ähnliche Objekte für die **Standareingabe** und **Standardausgabe** zur Verfügung: `sys.stdin` und `sys.stdout`

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

- Falls ein Dateiname angegeben wurde, soll die dazugehörige Datei geöffnet werden.

```
bf.py: Open files
import sys

def open_files(sfn, infn, outfn):
    if infn:
        fin = open(infn, "r")
    else:
        fin = sys.stdin
    if outfn:
        fout = open(outfn, "w")
    else:
        fout = sys.stdout
    return(open(sfn, "r"), fin, fout)
```

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Ausnahmebehandlung



Wo können Fehler passieren?

- Dateifehler (Existenz/Lesen/(Über-)Schreiben)
- Sollten wir besser abfangen!
- Fehler beim Interpretieren des Programms (Teilen durch 0 usw.)
- Für die Fehlersuche bei der Entwicklung erst einmal nicht abfangen, später dann schon.
- Verletzung von Sprachregeln wie z.B. Nicht-ASCII-Zeichen > 127, oder unbalancierte Klammern.
- Man sollte einen speziellen Ausnahmetyp einführen.

Spezielle Exception

```
class BFEError(Exception):  
    pass
```

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Die Hauptfunktion



bf.py: Main function

```
def bf(sfn, infn, outf):  
  
    try:  
        (src,fin,fout) = open_files(sfn, infn, outf)  
        pass # TBI: Aufruf des Interpreters  
    except IOError as e:  
        print("I/O-Fehler:", e)  
    except BFEError as e:  
        print("Abbruch wegen BF-Inkompatibilität:",e)  
    except Exception as e:  
        print("Interner Interpreter-Fehler:", e)  
    finally:  
        fout.close()
```

- Hier gibt es noch ein/zwei Problemchen!

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Die Hauptfunktion



bf.py: Main function

```
def bf(sfn, infn, outf):  
    fout = None  
    try:  
        (src,fin,fout) = open_files(sfn, infn, outf)  
        pass # TBI: Aufruf des Interpreters  
    except IOError as e:  
        print("I/O-Fehler:", e)  
    except BFEError as e:  
        print("Abbruch wegen BF-Inkompatibilität:",e)  
    except Exception as e:  
        print("Interner Interpreter-Fehler:", e)  
    finally:  
        if fout: fout.close()
```

- Hier gab es noch ein/zwei Problemchen!

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Ein erster Entwurf des Interpreters



bf0.py

```
def bfinterpret(srctext, fin, fout):  
    # Program counter points into source text  
    pc = 0;  
    # data pointer  
    ptr = 0;  
    # data cells are stored in a dict  
    data = dict();  
  
    while (pc < len(srctext)):  
        if srctext[pc] == '>'  
            ptr += 1  
        elif srctext[pc] == '+'  
            data[ptr] = data.get(ptr,0) + 1  
        elif ...  
            pc += 1
```

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Daten-getriebene Programmierung



- Sehr lange if-else-Anweisungen sind schwer lesbar, speziell wenn dann bei jedem Fall viele Dinge passieren (**Spagetti-Code**)
- Man kann die Fallunterscheidung auch Daten-getrieben vornehmen:
 - Wir legen eine Tabelle (dict) an, die für jeden BF-Befehl die notwendigen Operationen beschreibt (in Form einer Funktion).
- Von **Daten-getriebener Programmierung** spricht man, wenn das Programm nicht sequentiell die Daten abarbeitet, sondern der Datenstrom die Operationen determiniert.
- Diese Unterscheidung ist oft nur eine Frage der Perspektive, macht in unserem Fall aber einiges einfacher – die Funktion passt jetzt auf eine Folie!

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Umsetzung des Daten-getriebenen Entwurfs



Jetzt passt die Interpreter-Funktion auf eine Folie:

```
bf.py: Main interpreter loop
def bfinterpret(srctext, fin, fout):
    pc = 0;
    ptr = 0;
    data = dict();
    while (pc < len(srctext)):
        (pc, ptr) = instr.get(srctext[pc],noop)(pc,
            ptr, srctext, data, fin, fout)
        pc += 1
```

Wir benötigen also jetzt ein dict **instr**, in dem mit jeder BF-Instruktion eine Funktion assoziiert wird, die 5 Parameter besitzt und die ein Paar (pc, ptr) zurückgibt.

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Die Instruktionstabelle



```
bf.py: instr_table
instr = { '<': left, '>': right,
         '+': incr, '-': decr,
         '.': ch_out, ',': ch_in,
         '[': beginloop, ']' : endloop }
```

- Diese Tabelle darf erst definiert werden, nachdem alle Funktionen definiert wurden.

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Die einfachen Fälle (1)



```
bf.py: Simple cases
def noop(pc, ptr, src, data, fin, fout):
    return(pc, ptr)

def left(pc, ptr, src, data, fin, fout):
    return(pc, ptr - 1)

def right(pc, ptr, src, data, fin, fout):
    return(pc, ptr + 1)
```

Beachte: Die Variable pc wird in der Hauptschleife erhöht!

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schließen
Ausblick
Zusammen-
fassung

Die einfachen Fälle (2)



bf.py: Simple cases

```
def incr(pc, ptr, src, data, fin, fout):
    data[ptr] = data.get(ptr,0) + 1
    return(pc, ptr)
```

```
def decr(pc, ptr, src, data, fin, fout):
    data[ptr] = data.get(ptr,0) - 1
    return(pc, ptr)
```

Beachte: Wir lassen auch negative Indizes zu und es sind beliebig viele Zellen erlaubt.

- Motivation
- Programmiersprache
- Beispiele
- Semantik
- Interpreter-Design
- Designkriterien
- Datenstrukturen
- I/O
- Ausnahmebehandlung
- Hauptfunktion
- Datengetriebene Programmierung
- Einfache Fälle
- I/O
- Schleifen
- Ausblick
- Zusammenfassung

I/O



bf.py: I/O

```
def ch_in(pc, ptr, src, data, fin, fout):
    ch = fin.read(1)
    if ch:
        data[ptr] = ord(ch)
    return(pc, ptr)
```

```
def ch_out(pc, ptr, src, data, fin, fout):
    print(chr(data.get(ptr,0)), end='')
    return(pc, ptr)
```

Was passiert, wenn Ein- oder Ausgabe kein gültiges ASCII-Zeichen?

- Motivation
- Programmiersprache
- Beispiele
- Semantik
- Interpreter-Design
- Designkriterien
- Datenstrukturen
- I/O
- Ausnahmebehandlung
- Hauptfunktion
- Datengetriebene Programmierung
- Einfache Fälle
- I/O
- Schleifen
- Ausblick
- Zusammenfassung

I/O



bf.py: I/O

```
def ch_in(pc, ptr, src, data, fin, fout):
    ch = fin.read(1)
    if ch:
        data[ptr] = ord(ch)
        if data[ptr] > 127:
            raise BFEError(
                "Non-ASCII-Zeichen gelesen")
    return(pc, ptr)
```

```
def ch_out(pc, ptr, src, data, fin, fout):
    if data.get(ptr,0) > 127:
        raise BFEError(
            "Ausgabe eines Non-ASCII-Zeichen")
    print(chr(data.get(ptr,0)), end='')
    return(pc, ptr)
```

- Motivation
- Programmiersprache
- Beispiele
- Semantik
- Interpreter-Design
- Designkriterien
- Datenstrukturen
- I/O
- Ausnahmebehandlung
- Hauptfunktion
- Datengetriebene Programmierung
- Einfache Fälle
- I/O
- Schleifen
- Ausblick
- Zusammenfassung

Schleifen (1)



bf.py: Loop begin

```
def beginloop(pc, ptr, src, data, fin, fout):
    if data.get(ptr,0): return (pc, ptr)
    loop = 1;
    while loop > 0:
        pc += 1
        if src[pc] == ']':
            loop -= 1
        elif src[pc] == '[':
            loop += 1
    return(pc, ptr)
```

Frage: Was passiert bei unbalancierten Klammern?

- Motivation
- Programmiersprache
- Beispiele
- Semantik
- Interpreter-Design
- Designkriterien
- Datenstrukturen
- I/O
- Ausnahmebehandlung
- Hauptfunktion
- Datengetriebene Programmierung
- Einfache Fälle
- I/O
- Schleifen
- Ausblick
- Zusammenfassung

Schleifen (1')



bf.py: Loop begin

```
def beginloop(pc, ptr, src, data, fin, fout):
    if data.get(ptr,0): return (pc, ptr)
    loop = 1;
    while loop > 0:
        pc += 1
        if pc >= len(src):
            raise BFEError(
                "Kein passendes ']' gefunden")
        if src[pc] == ']':
            loop -= 1
        elif src[pc] == '[':
            loop += 1
    return(pc, ptr)
```

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schleifen
Ausblick
Zusammen-
fassung

Schleifen (2)



bf.py: Loop end

```
def endloop(pc, ptr, src, data, fin, fout):
    loop = 1;
    while loop > 0:
        pc -= 1
        if src[pc] == ']':
            loop += 1
        elif src[pc] == '[':
            loop -= 1
    return(pc - 1, ptr)
```

Frage: Was passiert bei unbalancierten Klammern?

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schleifen
Ausblick
Zusammen-
fassung

Schleifen (2')



bf.py: Loop end

```
def endloop(pc, ptr, src, data, fin, fout):
    loop = 1;
    while loop > 0:
        pc -= 1
        if pc < 0:
            raise BFEError(
                "Kein passendes '[' gefunden")
        if src[pc] == ']':
            loop += 1
        elif src[pc] == '[':
            loop -= 1
    return(pc - 1, ptr)
```

Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Designkriterien
Datenstrukturen
I/O
Ausnahmebehand-
lung
Hauptfunktion
Datengetriebene
Programmierung
Einfache Fälle
I/O
Schleifen
Ausblick
Zusammen-
fassung

6 Ausblick



Motivation
Program-
miersprache
Beispiele
Semantik
Interpreter-
Design
Ausblick
Zusammen-
fassung

Was kann man jetzt damit machen?



- Man kann jetzt BF-Programme schreiben und von unserem Interpreter ausführen lassen!
- Zum Beispiel das **Hello-World**-Programm
- Oder ein Programm zum Berechnen aller Werte der **Fakultätsfunktion**
- Oder ein **Adventure-Spiel**
- Oder ein Programm, das BF-Programme interpretiert, also einen **BF-Interpreter geschrieben in BF**.
- Wie wäre es mit einem Brainf*ck-Python Compiler (in Python oder Brainf*ck)?
- Oder umgekehrt?

Motivation
Programmiersprache
Beispiele
Semantik
Interpreter-Design
Ausblick
Zusammenfassung

7 Zusammenfassung



Motivation
Programmiersprache
Beispiele
Semantik
Interpreter-Design
Ausblick
Zusammenfassung

Zusammenfassung



- Brainf*ck ist eine **minimale** Programmiersprache.
- Sie ist aber so mächtig, dass alle berechenbaren Funktionen mit ihrer Hilfe ausgedrückt werden können (**Turing-vollständig**).
- Es ist relativ einfach, für diese Sprache einen Interpreter zu schreiben.
- Wir können auch einen Interpreter für Brainf*ck in Brainf*ck schreiben.
- Ähnlich können wir einen Interpreter für Python in Python schreiben: **Selbstanwendungsprinzip**
- Während das nicht wirklich sinnvoll ist, sind Compiler geschrieben in C, die C-Programme übersetzen, durchaus sinnvoll.

Motivation
Programmiersprache
Beispiele
Semantik
Interpreter-Design
Ausblick
Zusammenfassung