

F01: Brainfuck-Interpreter

Diese Aufgabe ist Teil der freiwilligen inoffiziellen Zusatzaufgaben von Eric Jacob und Jonas Wende, erstellt im WS 23/24 für *IN0002: Grundlagenpraktikum Programmierung*.

Weder sind sie durch die ÜL überprüft, noch unbedingt vollständig richtig.

Fehler gerne melden: eric.jacob.2003@gmail.com

Lernziele

Diese Aufgabe dient der Wiederholung folgender Konzepte:

- grundlegende Programmierkonstrukte (Schleifen, if/else, switch, etc.)
- Arrays
- logisches Denken

Backstory

Die älteren Pinguine gönnen sich gelegentlich auch mal Urlaub - und als Programmierlehrer-Pinguin Paddy vor zwei Wochen einen Spontanausflug zum Amundsensee vorschlug, waren alle Pinguine ganz begeistert - und nach hektischem Kofferpacken 3 Tage später auch schon verreist. Leider haben Sie in all dem Stress euch Babypinguine dabei ganz vergessen.

Nachdem ihr euch die ersten sturmfreien Tage mit Gartenparties vergnügt habt, setzen die Schneestürme nun wieder ein - bei der Kälte verlässt kein Pinguin freiwillig das Iglu! Und so widmet ihr euch wieder euren Programmieraufgaben. Jedoch wollt ihr Paddy auch eine kleine Lehre erteilen, damit er euch bei der nächsten Urlaubsplanung nicht nochmal vergisst.

Im Internet stoßt ihr dabei über eine Programmiersprache namens Brainfuck - das klingt doch ideal, um das ausgeruhte Paddy-Gehirn schnell wieder auf Trab zu bekommen!

Aufgabenbeschreibung

Ziel dieser Aufgabe ist es, einen Interpreter für die [esoterische Programmiersprache Brainfuck](#) in Java zu entwickeln.

Brainfuck ist eine (konzeptionell) sehr einfache Programmiersprache, ähnlich einer [Turingmaschine](#). Eine solche kann man sich als (unendlich) langes Band vorstellen, welches in einzelne Zellen unterteilt ist. Jede Zelle speichert dabei einen (theoretisch unendlich kleinen bzw. großen) Ganzzahlwert. Modifiziert werden diese Zellen durch einen Schreibkopf, welcher auf dem Band nach links und rechts bewegt werden und die aktuelle Zelle jeweils um 1 erhöhen oder verringern kann.

Kleiner Funfact: Eine Turingmaschine (und damit auch Brainfuck) ist - wie der Name vermuten lässt - [turing-vollständig](#). Das bedeutet, dass man jedes Programm einer Hochsprache (wie Java oder Python) auch als Instruktionsliste für eine Turingmaschine ausdrücken kann. Man könnte also auch ein ganzes Betriebssystem in Brainfuck schreiben - praktisch ist das selten, aber theoretisch möglich.

In Brainfuck gibt es folgende Befehle:

Befehl	Bedeutung	Pseudocode-Äquivalent (vereinfacht)
<code>+</code>	Inkrementiert die aktuelle Zelle um eins.	<code>cell++</code>
<code>-</code>	Dekrementiert die aktuelle Zelle um eins.	<code>cell--</code>
<code><</code>	Bewegt den Schreibkopf eine Zelle nach links.	<code>pos--</code>
<code>></code>	Bewegt den Schreibkopf eine Zelle nach rechts.	<code>pos++</code>
<code>.</code>	Gibt das zum Wert der aktuellen Zelle gehörige ASCII-Zeichen (siehe unten) auf der Konsole aus.	<code>print((char) cell)</code>
<code>,</code>	Liest ein Zeichen vom Nutzer ein und ersetzt die aktuelle Zelle mit dessen Zahlwert.	<code>cell = readChar()</code>
<code>[</code>	Falls der Wert der aktuellen Zelle größer 0 ist, wird der dahinter folgende Code ausgeführt. Sonst wird zum ersten Zeichen hinter dem zugehörigen <code>]</code> gesprungen.	<code>while (cell > 0) {</code>
<code>]</code>	Springt im Code zurück zum zugehörigen <code>[</code> .	<code>} endloop</code>

Beispiel I:

`+++` erzeugt folgendes Band (von welchem in den Beispielen jeweils nur die relevanten Abschnitte dargestellt werden):

```
1 | ... 0 3 0 ...
2 |      ^
```

Die Position des Schreibkopfs wird dabei mit `^` dargestellt, die Leerzeichen im Code dienen nur der besseren Lesbarkeit.

Beispiel II:

`++ [> ++ > - - - < < -] >` erzeugt das Band:

vor Eintritt in die Schleife:

```
1 | ... 0 2 0 0 0 ...
2 |      ^
```

nach einer Iteration der Schleife:

```

1 | ... 0 1 2 -3 0 ...
2 |      ^

```

am Ende des Programms:

```

1 | ... 0 0 4 -6 0 ...
2 |      ^

```

Beispiel III:

```

1 | +++++ +++++ (Zelle #0) wiederhole 10x
2 | [           setze Startwerte 70 und 100 für die beiden Zellen rechts
3 |     > +++++ ++      addiere 7 auf Zelle #1
4 |     > +++++ +++++    addiere 10 auf Zelle #2
5 |     << -             dekrementiere den Zähler (Zelle #0)
6 | ]
7 | > ++ .             addiere 2 auf Zelle #1 (72 = 'H'), gib Zeichen aus
8 | > +++++ .           addiere 5 auf Zelle #2 (105 = 'i'), gib Zeichen aus



```

erzeugt also `Hi` auf der Konsole.

Template

Das Template für diese Aufgabe sieht wie folgt aus:

( bedeutet, dass du in dieser Methode etwas ändern/ergänzen musst.)

-  `runBrainfuckSequence(int[] tape, String code, boolean printAsChars)` ist die Hauptmethode - hier iterierst du über die Zeichen des Codes und führst die Befehle auf deinem Band aus.
 - `tape` ist das Datenband in Form eines `int`-Arrays. Beachte, dass dieses zu Beginn beliebig lang sein und auch bereits Daten beinhalten kann.
 - `code` ist der Brainfuck-Code.
 - `printAsChars` gibt an, ob die Ausgaben bei einem `.` in `code` als `int`s oder `char`s dargestellt werden sollen (macht das Debugging leichter).
-  `addPlaceToTape(int[] oldTape, boolean inFront)` verlängert das Band bei Bedarf, indem am linken oder rechten Ende Zellen ergänzt werden. Diese Methode kannst du in `runBrainfuckSequence` nutzen.

Kleine Anmerkung: Diese Art der Arrayvergrößerung ist sehr ineffizient - noch kennen wir allerdings nichts besseres. Solange dein Code nicht hundertmal in eine Richtung läuft, sollte das auch kein Problem sein.

- `oldTape` ist das aktuelle Band.
- `inFront` gibt an, ob die zusätzliche Zelle links (front) oder rechts (back bzw. `!inFront`) angefügt werden soll.

- Rückgabe: das vergrößerte Array
- `readCharFromConsole()` liest das nächste Zeichen von der Konsole und gibt dieses zurück. Diese Methode kannst du in `runBrainfuckSequence` nutzen.
 - Rückgabe: das gelesene Zeichen

Aufgaben

1. Bandvergrößerung

Zuerst sollst du `addPlaceToTape()` implementieren. Achte darauf, die Daten des alten Arrays in das neue zu übernehmen und die neue Zelle am richtigen Ende hinzuzufügen.

2. Datenspeicherung

Überlege dir nun, wie du in `runBrainfuckSequence()` die aktuelle Position im Code und auf dem Band speichern willst.

Der einfacheren Lesbarkeit halber sollen Leerzeichen im `code` erlaubt sein - für die Abarbeitung des Codes musst du diese jedoch entfernen. Nutze dafür die Methode `.replace("zu ersetzender String", "neuer String")` auf dem entsprechenden String.

3. Pretty Print

Nach jedem Befehl aus `code` soll das aktuelle Band und die Zeigerposition wie folgt ausgegeben werden:

```
1 | tape: [0, 0, 3, -5, 0], instruction: +, position in code: 5
2 |           ^
```

Überlege dabei, wie du dir die aktuelle Position im Code und im Array zunutze machen kannst, den Zeiger an der richtigen Stelle auszugeben.

Tipp: Um das Array leicht ausgeben zu können, bietet sich möglicherweise ein Blick in die [Arrays-Library](#) an.

4. Befehle lesen & verarbeiten

Iteriere nun über die Befehle in `code` und implementiere eine Fallunterscheidung sowie die einfachen Instruktionen (also `+`, `-`, `<`, `>`, `.`, `,`). Achte bei `<` und `>` darauf, dass keine `OutOfBounds-Error` entstehen und das Band entsprechend vergrößert wird.

5. Loops

- Wie kannst du (ohne Rekursion! - nicht auf dumme Gedanken kommen) sicherstellen, beim Überspringen des Schleifencodes im Falle von `currentCell == 0`, an der richtigen Endklammer `]` rauszukommen?
- Wie läufst du bei einem `]` an die richtige Startklammer `[` zurück? Achte darauf, bei der Abarbeitung des nächsten Zeichens keines zu überspringen!

Beachte, dass beim Überspringen bzw. Zurücklaufen **keine** Befehle ausgeführt werden sollen!

Tipp: Es genügt eine primitive Variable.

6. Fehlererkennung

Woran kann man erkennen, dass der Code zu viele öffnende bzw. schließende Klammern enthält? Gib in diesen Fällen, sowie bei unbekannten Zeichen im Programmcode eine Meldung auf der Konsole aus und beende die Ausführung des Programms.



Tests

Für diese Aufgabe gibt es keine automatisierten Tests. Du kannst allerdings mit folgenden Beispielen testen, ob dein Programm die Anforderungen erfüllt.

1. Code: `+ + +`, Tape: `{2}`

erwartetes Ergebnis: `[5]`

2. Code: `+ + [> + + > - - - < < -] >`, Tape: `{}`

erwartetes Ergebnis: `[0, 4, -6]`

3. Code: `+ + + + + + + + + [> + + + + + > + + + + + + + + < < -] > + + . > + + + + + .`, Tape: `{1, 1}`

erwartetes Ergebnis: Konsole: `Hi`, Tape: `[0, 72, 105]`

4. Code: `> + + + + + + + + [< + + + + + + + > -] < . > + + + + + + + [< + + + + + > -] < . > + + + + + + + + + . . + + + . -] > + + + + + + + + [< + + + + + > -] < . > + + + + + + + + + [< + + + + + + + > -] < - . - - - - - - - . + + + . - - - - . - - - - - - - . [-] > + + + + + + + + [< + + + + + > -] < + . [-] + + + + + + + + + .`, Tape: `{}`

erwartetes Ergebnis: Konsole: `Hello world!`, Tape: `[10, 0]`



Anhang

Online-Compiler zum Testen eurer Programme

<https://minond.xyz/brainfuck/>

ASCII-Tabelle als Hilfestellung

(unter Linux mit `ascii -d` erzeugbar)

1	0	NUL	16	DLE	32		48	0	64	@	80	P	96	`	112	p
2	1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
3	2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
4	3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
5	4	EOT	20	DC4	36	\$	52	4	68	D	84	T	100	d	116	t
6	5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
7	6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v

8	7 BEL	23 ETB	39 '	55 7	71 G	87 w	103 g	119 w
9	8 BS	24 CAN	40 (56 8	72 H	88 x	104 h	120 x
10	9 HT	25 EM	41)	57 9	73 I	89 Y	105 i	121 y
11	10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
12	11 VT	27 ESC	43 +	59 ;	75 K	91 [107 k	123 {
13	12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124
14	13 CR	29 GS	45 -	61 =	77 M	93]	109 m	125 }
15	14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
16	15 SI	31 US	47 /	63 ?	79 O	95 _	111 o	127 DEL

Zu beachten dabei: Die Zeichen mit den Zahlwerten 0 bis einschließlich 31 sind *control characters*, welche auf den meisten Konsolen eher etwas *bewirken* (z.B. ist 13 ein Zeilenumbruch), als etwas anzuzeigen.