Brainfuck—A Formal Semantics

Abstract

This is a formal semantics for the turing complete language "brainfuck". Brainfuck was designed by Urban Müller in 1993. More information can be found at http://en.wikipedia.org/wiki/Brainfuck. Some comments below are adapted from the Wikipedia article.

$\mathbb{K}$ formal semantics for BF written by Chucky Ellison (celliso2@illinois.edu). It is written in the "fully labeled" style.

BF has eight commands: "+", "-", ">", "<", "[", "]", ".", and ",". All other characters are treated as comments. Below is a commented "Hello World!" program:

```
+++++ +++++             initialize counter (cell #0) to 10
[                       use loop to set the next four cells to 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'
> ++ .                  print ' '
<< +++++ +++++ +++++ .  print 'W'
> .                     print 'o'
+++ .                   print 'r'
----- - .               print 'l'
----- --- .             print 'd'
> + .                   print '!'
> .                     print '\n'
```

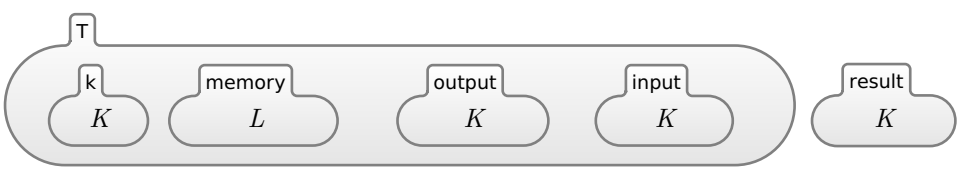
MODULE BF

We use an abstract syntax in this formal semantics. The mapping from concrete to abstract syntax is given by the following chart:

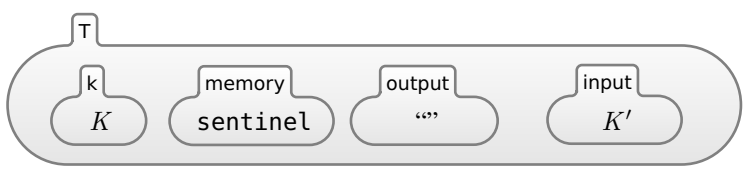
Concrete	Abstract
+	Increase
-	Decrease
>	GoRight
<	GoLeft
[ ]	While
.	Print
,	Read

Note: the [and] commands of BF are assumed to be matching, and are represented by a single "While" construct.

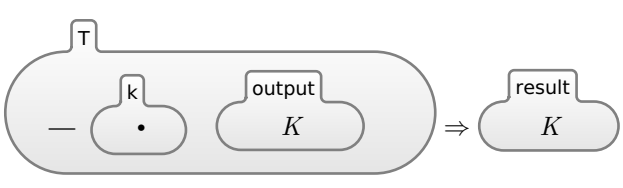
SYNTAX *KLabel* ::= Seq
SYNTAX *K* ::= GoRight | GoLeft | Increase | Decrease | Print | Read
SYNTAX *KLabel* ::= While
SYNTAX *K* ::= sentinel
SYNTAX *Bag* ::= eval(*K*) | eval-inp(*K* , *K*)
SYNTAX *KResult* ::= *#Int* | *#String*
CONFIGURATION:



RULE eval(*K*) ⇒ eval-inp(*K* , "") [Start structural]

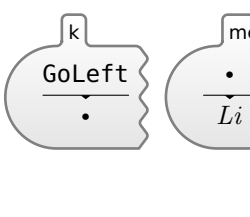
RULE eval-inp(*K* , *K′*) ⇒ 

[Start-With-Input structural]

RULE 

[Finish structural]

We use a sentinel to denote the edge of memory, and automatically create more memory when it is reached,

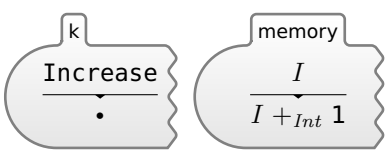
RULE 

[structural]

This rule simply turns sequenced commands into the first command followed by the rest,

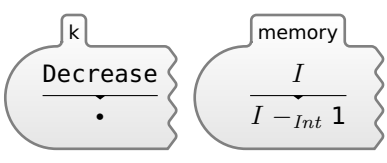
RULE Seq (K_1 , K_2) ⇒ $K_1 \curvearrowright K_2$ [Sequence structural]

Increase: increment (increase by one) the byte at the data pointer.

RULE 

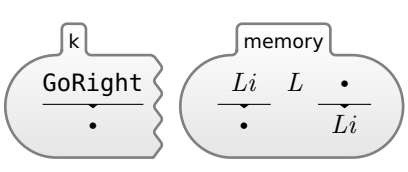
[Increase]

Decrease: decrement (decrease by one) the byte at the data pointer.

RULE 

[Decrease]

GoRight: increment the data pointer (to point to the next cell to the right). Note that the "data pointer" is represented by whatever is left-most in the memory cell. Thus, to move the pointer right, we simply roll the contents of the cell to the left.

RULE 

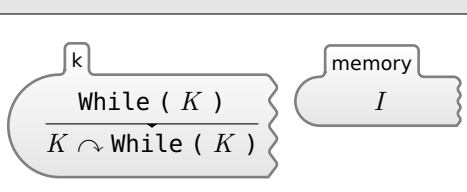
[GoRight]

GoLeft: decrement the data pointer (to point to the next cell to the left).

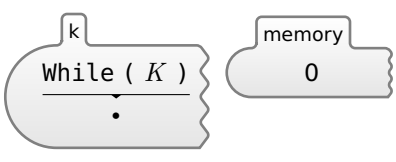
RULE

[GoLeft]

While: if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it past the end of the loop. If the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, enter the loop.

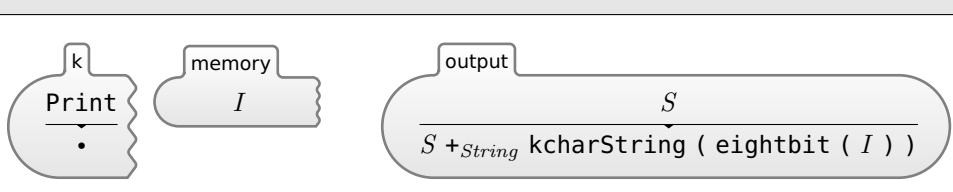
RULE 

when $I \neq_{Bool}$ 0 [WhileNZ-NZ]

RULE 

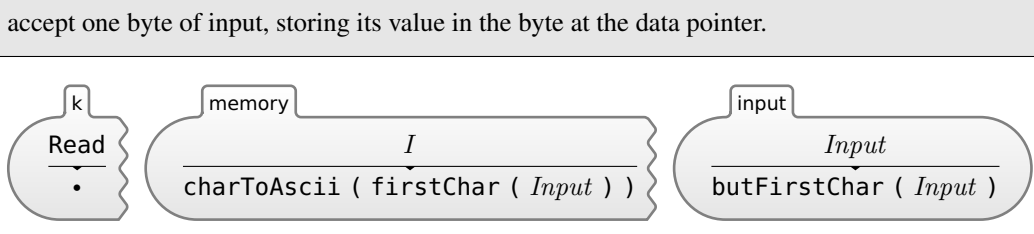
[WhileNZ-Z]

Print: output the value of the byte at the data pointer.

RULE 

[Output]

Read: accept one byte of input, storing its value in the byte at the data pointer.

RULE 

[Input]

The rest of the definition is simply helper functions.

SYNTAX *KLabel* ::= eightbit
RULE eightbit (*I*) ⇒ 256 +Int *I* %Int 256 %Int 256 [structural]

SYNTAX *KLabel* ::= kcharString
RULE kcharString (*N*) ⇒ charString (*N*) [structural]

SYNTAX *KLabel* ::= +String
RULE +String (S_1 , S_2) ⇒ S_1 +String S_2 [structural]

SYNTAX *KLabel* ::= firstChar | butFirstChar | charToAscii
RULE firstChar (*S*) ⇒ substrString (*S* , 0 , 1) [firstChar structural]

RULE charToAscii (*C*) ⇒ asciiString (*C*) [charToAscii structural]

RULE butFirstChar (*S*) ⇒ substrString (*S* , 1 , lengthString (*S*)) [butFirstChar structural]

END MODULE