Language Specification:

Back is a Concurrent Forth implementation, which consists mainly of 25 words and 3 prefixes each described thoroughly below.

Recursive colon definitions are NOT allowed and the compiler will throw an error whenever it encounters one.

Each program is separated into threads which can either run isolated or take part in a bigger job consisting of multiple threads working together by passing and receiving messages to each other.

The name of each thread really don’t matter, but it’s good practice to give them descriptive names.

Note that back can NOT technically extend itself at runtime, but rather does so at compile time to increase performance. (However note that it practically can)

Back's Compiler Specification

A typical Back file has two scopes:

A global one: which consists of Word and thread definitions.

A local one: which is local to each thread, can contain anything apart from thread definitions.

Notes:

- Word definitions cannot be nested, as that will make the code more ambiguous and is generally harder to parse
- If-then words cannot be nested, as that will make the code more ambiguous, and complicate the VM
- do-loop words cannot be nested, for the same reasons stated above
- Anything that is not recognized by the lexer, or is not a number is considered a valid Word name
- Comments are anything between parenthesis, they can be multiline too
- When encountering a number, make sure to output `` `26 the_number` `` because 26 is the opcode for push. (see the table below)
- Prefixing a word with '$' reads the word as a hexadecimal number and converts it to decimal before pushing it to the stack
- Prefixing a word with '~' encodes the word to a decimal number (see details below) and outputs `` `27 encoded` `` which "binds" to the encoded number (see table below)
- Prefixing a word '@' encodes the word to a decimal number (see details below) and outputs `` `28 encoded` `` which "puts" the encoded number (see table below)

Encode Function:

The encode function is responsible for encoding variable names to decimal number, which works like this (from left to right) :

1. Turn the character into its ASCII representation
2. Pad it to a 3 number string so '1' -> '001'
3. Concatenate the strings together from the same order they were in
4. Turn it the string to a decimal number, which is the result

So for example 'ab' -> '61062'

Code example:

```
: dup_add dup + ;
main [
	: cr 10 emit ; ( this is local )
	2 dup_add . cr
]
```

Output Format:

The compiler should output bytecode in the following format :

`THREAD_NAME  THREAD_CODE`

Where this pattern repeats for every thread in their respective order where THREAD_NAME represents the name of the thread, and THREAD_CODE represents corresponding opcodes for the code in between the '[' and ']' words

Back’s VM Specification

The VM reads a file that contains the output of Back’s Compiler

Notes :

- Thread ids are set by the order they have been defined in, starting at 0
- All words that uses addresses, convert the number they are going to use from decimal to hexadecimal form first, in order to use it properly
- Note that push is not an actual word used at runtime, but rather a special opcode used to push numbers to the stack generated by the compiler

The following table describes the corresponding opcodes for the predefined words and how they work :

(Note : thread ids are corresponding by the order they have been defined in, starting at 0)

Word	Opcode	Description
	0	Does nothing
.	1	pops the top number from the stack and prints it
,	2	waits for input from “STDIN” and pushes it to the stack
emit	3	Pops the top number from the stack and outputs it in ASCII form
+	4	Pops the two top numbers from the stack, adds them, then push the result to the stack
-	5	Pops a, and then pops b from the stack, and pushes the result of b-a
*	6	Pops the two top numbers from the stack, multiplies them, then push the result to the stack
/	7	Pops a, and then pops b from the stack, and pushes the result of b/a
%	8	Pops a, and then pops b from stack, and pushes the result of b % a

if	9	Pops the top number from the stack, if it’s true continue executing as normal, if not jump to the next “then” word
then	10	Serves as a label, for the “if” word, to skip executing in case of a false value
dup	11	Pops the top number from stack, and pushes it two times
rot	12	Rotates the top three values in the stack
swap	13	Swaps the two top numbers on the stack
drop	14	Pops the top number from the stack
over	15	Pushes the second top number from stack so (see table below)
alloc	16	Pops the top number from the stack, allocates memory with that size, then pushes the address, if the address is invalid it will push 1
free	17	Pops the top address from the stack, if it’s a valid one, free that buffer, if not push 1
write	18	Pops the top address from the stack, if it’s a valid one, pop the top number from the stack, and write it to the address, if not push 1
read	19	Pops the top address from the stack, if it’s a valid one, read it, and push the contents, if not push 1
send	20	Pops a, and then b from the stack where b is the id of the thread to send to, and a the value to send, and then sends the value for accepting by the corresponding thread (this action does not hang)
recv	21	Waits for a value from any thread, and then pushes it. (this action does hang)

recv#	22	Pops a number from the stack (n) , and waits until it receives n values and pushes them to the stack
exit	23	Pops the top number from the stack, and exists the program with it as the exit code
do	24	Pops s, and then end, and loops end-s times the code before the next 'loop' word
loop	25	Serves as a label for do, to set the instruction pointer and increment the loop counter
(push)	26	Pushes the following number to the stack
(prefix: ~)	27	Binds the following number to the top value on the stack
(prefix: @)	28	Fetches the value of the following number from the variable table and pushes it to the stack

The following table describes the corresponding stack effects for the each word:

Word	Stack Effect
.	(n --)
,	(-- n)
emit	(n --)
+	(a b – n)
-	(a b – n)
*	(a b – n)
/	(a b – n)
%	(a b – n)
if	(n --)
then	(--)
dup	(n – n n)
rot	(x1 x2 x3 – x2 x3 x1)
swap	(a b – b a)
drop	(n --)
over	(a b – a b a)
alloc	(n – n)
free	(n -- ?n)
Write	(a b -- ?n)
read	(a -- ?n)
send	(a b --)
recv	(-- n)

recv#	(n – n number of values)
exit	(n --)
do	(a b --)
loop	(--)
(prefix: ~)	(n --)
(prefix: @)	(-- n)

(Note: ‘?n’ denotes that it may exist or may not)