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 names cannot be nested, as that will make the code more ambiguous and is generally harder to parse
- If-then blocks cannot be nested, as that will mess up the resulting bytecode
- 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

Example :

```
: dup_add dup + ;
Main [
        : dup_dup_add dup dup+ ; ( this is local )
5 dup_add . dup_add .
]
```

Back’s VM Specification

The Back VM’s Bytecode must adhere to the following format :

`THREAD_NAME CODE`

Where THREAD_NAME represents the name of thread, and CODE represents the generated bytecode from what’s in between ‘[‘ and ‘], this pattern can repeat as many times, but with a different THREAD_NAME each time

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

Word	Opcode	Description
.	0	pops the top number from the stack and prints it
,	1	waits for input from “STDIN” and pushes it to the stack
emit	2	Pops the top number from the stack and outputs it in ASCII form
+	3	Pops the two top numbers from the stack, adds them, then push the result to the stack
-	4	Pops a, and then pops b from the stack, and pushes the result of b-a
*	5	Pops the two top numbers from the stack, multiplies them, then push the result to the stack
/	6	Pops a, and then pops b from the stack, and pushes the result of b/a
%	7	Pops a, and then pops b from stack, and pushes the result of b % a
if	8	Pops the top number from the stack, if it’s true continue executing as normal, if not jump to the next “then” word
then	9	Serves as a label, for the “if” word, to skip executing in case of a false value

dup	10	Pops the top number from stack, and pushes it two times
rot	11	Rotates the top three stack entries so : “(x1 x2 x3 – x2 x3 x1)”
swap	12	Swaps the two top numbers on the stack
drop	13	Pops the top number from the stack
over	14	Pushes the second top number from stack so : “(x1 x2 – x1 x2 x1)”
alloc	15	Pops the top number from the stack, allocates a buffer with that size, then pushes the address, and 0
free	16	Pops the top address from the stack, if it’s a valid one, free that buffer, and push 0, if not push 1
write	17	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	18	Pops the top address from the stack, if it’s a valid one, read it, and push the contents, if not push 1
send	19	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	20	Pops a from the stack, where a is the id of the thread to receive from (this action does block the execution)
exit	21	Pops the top number from the stack, and exists the program with it as the exit code
push	22	Pushes the following number to the stack

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