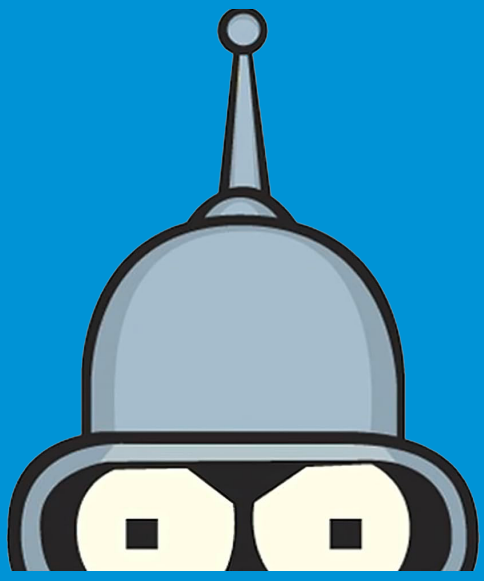
{EPITECH}

BOOTSTRAP - COREWAR

ELEMENTARY PROGRAMMING IN C



BOOTSTRAP - COREWAR

language: C
Authorized functions: For this bootstrap, the only authorized functions are those of the standard `libC`.

- ✓ The totality of your source files, except all useless files (binary, temp files, objfiles,...), must be included in your delivery.
- ✓ All the bonus files (including a potential specific Makefile) should be in a directory named bonus.
- ✓ Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

Each exercise is to be submitted in a file called partX/stepY in the repository's root, and a Makefile at the root of the directory must compile all of the exercises.

If you haven't already done so, please read the Corewar project description and subject.
If you didn't come to the Corewar Kick-off, well, you should have!

The first part of this bootstrap is the **Robot Factory**. It covers the main concepts of the project, including the importance of reading transcribed bytecode.

This bootstrap concentrates on the assembler (the compiler that we're asking you to implement in the subject).
It aims at helping you, step by step, generating bytecode and creating a small compiler.

Part 1 : Binaries

Step 1

Binary name: ./part1/step1/write-some-text
Write a program that creates a file named `./some-text.yolo`. This file must contain `"Hello bambino\n"` inside of it.

Step 2

Binary name: ./part1/step2/write-a-number-as-text
Write a program that creates a file named `./number-as-text.yolo`. This file must contain the character string `"12345678"`.
Your file should be 8 bytes and human-readable.

Step 3

Binary name: ./part1/step3/write-a-number-as-int
Write a program that creates a file named ./number-as-int.yolo. This file must contain the integer 12345678.
Your file should be 4 bytes and not human-readable.

Step 4

Binary name: ./part1/step4/without-padding
Write a program that creates a file named `./several-variables.yolo`. This file must contain `192837` as an int, `'k'` as a char and `"Corewar is swag!!"` as a char[40].
These variables must be written one after another in the file.

Step 5

Binary name: ./part1/step5/with-padding
Write a program that creates a file named `./one-structure.yolo`. This file must contain `192837` as an int, `'k'` as a char and `"Corewar is swag!!"` as a char[40].
These variables must form a structure, which must be written in the file.
Note that we're stocking the same information as the previous step, but the file isn't the same size. How
come?

Part 2 : Yolo !

The yolotron

Let's invent a new programming language: the yolotron. This language, much more limited than C or even the assembly language proposed in the Corewar project, only tolerates the following instructions:

- ✓ add: adds two numbers and displays the result,
- ✓ sub: substracts two numbers and displays the result,
- ✓ mul: multiplies two numbers and displays the result,
- ✓ put: displays a word.

Here is an example of a super program written in yolotron:

```
Terminal
~/B-CPE-200> cat ./example.yolotron
add 17891 21
add 59 1
sub 21 10
put Yolo!!!!!
```

Once this program is compiled and interpreted by the VM, it must display:

```
Terminal
~/B-CPE-200> ./yolotron-vm ./example.bytecode
17912
60
11
Yolo!!!!!
```

If, during compilation, the slightest error is found (for example, too many or not enough parameters passed to an instruction), the `yolotron-asm` compiler must display `"ahhhh no, don't agree!\n"` and terminate.
The compiler must ensure that each line begins with a valid instruction, followed by 2 or 3 parameters (depending on the instruction).
The instruction and its parameters must be separated by one single space. Spaces in the beginning and the end are not tolerated

Step 1

Binary name: ./part2/step1/yolotron-asm
Create a program named `yolotron-asm` that can compile yolotron code.
You'll need to translate the instructions and their parameters into binary, using the conversion table:

Instruction	add	sub	mul	put
Hexadecimal Code	0x01	0x02	0x03	0x04

Encoding put will be done in the following format `[code instruction][int word length][word]`, which results in (5+strlen(word)) bytes.
Encoding other instructions will be done in the following format `[code instruction][int param1][int param2]`, which results in 9 bytes.
For instance, reusing the previous example:

```
Terminal
~/B-CPE-200> ./yolotron-asm -h
Usage: ./yolotron-asm [source file] [output file]
```

```
Terminal
~/B-CPE-200> ./yolotron-asm example.yolotron example.bytecode && hexdump -C
example.bytecode
00000000 01 e3 45 00 00 15 00 00 00 01 3b 00 00 00 01 00 |..E.......;.....|
00000010 00 00 02 15 00 00 00 0a 00 00 00 04 09 00 00 00 |................|
00000020 59 6f 6c 6f 21 21 21 21 21                     |Yolo!!!!!|
00000029
```

Step 2

To find out more about what will be required in Corewar:

Binary name: ./part2/step2/yolotron-vm
Write a program called `yolotron-vm` that can interprete the instructions of a compiled yolotron program (you have an example higher up in the description).

Part 3 : .cor files

The Robot Factory project let you to transcribe .s files (in assembler) into .cor files (in bytecode).

Now it's time to see what you can do with them!

Step1

Binary name : ./part3/step1/get-champion-infos
Write a program called `get-champion-infos` that takes a path to a .cor file as a parameter and displays the values of the champion's own `header_t` structure: `prog_name`, `prog_size`, `comment`.

The header_t structure is specified in the op.h file attached to the Robot Factory or Corewar project.

Chances are that the strings are the expected ones, but the numbers (prog_size, magic) are... different from what's expected. But didn't we already have the same problem in the Robot Factory? Little endian, big endian...

Terminal

```
~/B-CPE-200> ./get-champion-infos tyron.cor
prog_name: Tyron
prog_size: 22
comment: Just a basic pompes, traction program
```

Step2

Binary name : ./part3/step2/check-magic-champion
Write a program called `check-champion-infos` that takes a path to a .cor file as a parameter and displays if provided .cor begin with correct magic number.

```
Terminal
~/B-CPE-200> ./check-magic-champion tyron.cor
Valid magic number!
```

Step 3

Before deciphering the instructions, you need to know how to read them, and therefore the number of bytes to read in order to retrieve their arguments. If you've followed this correctly, it's the **coding byte** we're interested in! A value on a byte that gives us precisely this information.

Write a function called that takes a coding bytes as an integer, hexadecimal or string and displays parameters values according to Robot Factory subject values.

```
Terminal
~/B-CPE-200> ./your_binary_calling_your_function
For value: 0x78
1: 01 -> register
2: 11 -> indirect
3: 10 -> direct
4: 00
```

Step 4

Now you've got everything you need to start deciphering all the instructions! Get the instruction code, then its byte coding if it exists, and determine how many bytes you need to read your champion by, and so on.

v1