Level 2 Report — BRAINF-CK Project

MIAOU Is An Open Umbrella

October 29, 2016

1 Introduction

Level 2 is now complete. It consists mainly of the following new features:

- Reading long, short and image syntax
- Translating from long to short syntax and from text to image
- Supporting input, output, jump and back instructions
- Checking consistency of jump and back instructions

2 Reminder from Level 1

2.1 Instructions

The instructions are implemented in separate classes, all inheriting the `Instruction` class. Their behaviour is defined by a unique method which alter the machine state. This way, we can easily implement new Instructions if they follow the same system as the others.

All instructions are instantiated in the `InstructionSet` class which provides methods to easily fetch them by symbol, keyword or color code using `HashMap`s.

2.2 Textual brainfuck program reading

In Level 1, only the long syntax was understood. There was no intermediate operation between parsing a line and executing an instruction: it was done at the same time. Level 2 requires us to be able to parse short syntax too, and understand images, so further abstraction is needed.

2.3 Exit codes

In level 1, we initially used exceptions, and then got rid of them to call only `System.exit()` with the required error code. This changed another time.

3 New features implementation

3.1 Reading both short and long syntax

Both syntax can be present in the same file. This forces us to alter the method we previously used for parsing the instructions, rather than writing a separate one. Also, prevention of the next features, we separated the parsing from the execution of the instructions.

As before, the text file reader (as `ReadTextFile` class) gives us a stream of lines. There is now a parser (`InstructionParser` class) which will read a line and fetch the corresponding instruction, supporting the long syntax. If it fails, it will read each character of the line and fetch the instructions in order to support the short syntax. Finally if it fails, it will throw an exception (`InvalidInstructionException`). We find this structure great, because it allows us to separate the action of reading a file and extracting its content from parsing this same data. We're getting more modular that way and more "Object Oriented", since each class has only one job on its own. So we judged it as the best structure for reading files and parsing instructions.

All the fetched instructions are appended to a list of instructions as we want a non-fixed size collection of Instructions.

The interpreter (`Interpreter` class) now only loop through the list of instructions and tells the machine to execute them.

3.2 Reading brainfuck images

Image reader (as the `ReadImageFile` class) is implemented using Java's `ImageIO` API. It supports BMP format out of the box so it perfectly suits our needs. Using ImageIO's `read()` method, we get a `BufferedImage` object on which the `getRGB()` method gives us the color of a specific pixel (described by its coordinates) as an integer in the ARGB format.

That way, we can iterate over every columns and rows of 3x3 pixels. In fact, we increment the x coordinate from 0 to the image width and the y coordinate from 0 to the image height, both by 3. In this process, an IntStream is built out of every block's color.

Later on, the instruction parser (as the `InstructionParser` class) will try to fetch the instruction for each color code and constructs a list of instructions out of every instructions fetched. Matching is done using the color attribute of each Instruction where we wrote color codes in hexadecimal, ARGB format again.

Black is ignored, and as with the text format, an unmatched instruction will throw an exception (`InvalidInstructionException` here).

3.3 Translating from long to short syntax and from text to image

3.3.1 Translator

As each `Instruction` class contains three fields (one for each syntax) and an accessor for each of them, we thought that the best way to translate an instruction from one syntax to another was using these accessors. The `Translator` class contains several methods which are mainly used for, given a specific collection of Instructions, getting the representation of these Instructions in a particular syntax. In this way, we could generalize the Translator to make possible the translation of every instruction to every syntax available. We thought this was a great solution, because a modification of the specifications about the translation modes

(like adding long syntax translation, for example) would be much easier to implement with this architecture. In the same way, if a future implementation requires us to write an image aside of translation, the writer is not linked to the translator at all.

3.3.2 Translating from long to short syntax

The first translation mode required in the specifications was from long text syntax to short text syntax (for example, from "INCR" to "+"). As we have to display text, the `Translator` simply puts the short syntax result on the standard output.

3.3.3 Translating from text to image syntax

The second translation mode that was required in the specifications was from text syntax (long or short) to image syntax. If the Brainfuck program that has to be translated is in text format, the Translator returns a List of integers, corresponding to each color that has to be drawn (each Instruction that has to be written). This List is passed as an argument to WriteImage class, which will construct a BMP image. Else, if the Brainfuck program is in image syntax, it simply prints the content of the file on the standard output.

3.3.4 Writing images

In order to produce an image from a text file, we need to be able to write image files. As with the image reader, the image writer (`WriteImage` class) is implemented using Java's `ImageIO` API. It can also generate BMP files for us using its `write()` method. In our case we tell it to write our image in the BMP format to the standard output. Since we want square images, the number of columns/lines is set as the square root of the number of squares to write, rounding it upward to the next integer.

The image itself (a `BufferedImage` again) is drawn with the help of `Graphics` from `awt`. Iterating over a list of colors as integers in the ARGB format, we create a filled rectangle (thanks to `fillRect()` method) of 3x3 pixels for each color.

3.4 In and Out instructions

In and Out are two new Instructions who interact respectively with a given `InputStream` and an `OutputSteam`. By default, those streams are the standard input (`System.in` in Java) and the standard output (`System.out` in Java). Though, it's possible with the arguments to overwrite this default configuration and to make the In instruction read from a file and/or the Out instruction write to a file. Those streams are stored in an Io class along with methods allowing to interact directly with them (i.e : reading from the `InputStream` and writing to the `OutputStream`). But we chose to make the interaction between the Io and the Instructions only possible through the `Machine`, which contains an Io field. This structure keeps the `Machine` as a hub between `Instructions`, `Memory`, and `Io`.

The Io object is instantiated in our launched for it to receive the correct file names if relevant, and is then given to the Machine, which can be compared to plugging in a peripheral to a computer.

When In instruction is called, it fetches a byte from the `InputStream` and writes its value in the selected memory case. On the contrary, when Out instruction is called, it writes the byte read from the current memory case to the `OutputStream`. If the output stream is the standard output, the system will display the characters corresponding to the byte's value according to the default encoding.

3.5 Conditional Jumps

Since they have an additional behaviour, Jump and Back instructions are subclasses of the `ConditionalJump` which is an abstract subclass of `Instruction`. `ConditionalJump` declares an abstract method called `incr()` which increments a given `BracketCounter`'s right or left counter. It is later defined in the instructions to increment the correct counter. That way, when we are currently jumping forward or back, we know when to stop (when we reached the corresponding Back or Jump instruction) and return to normal behaviour. `BracketCounter` will in practice run a callback function defined at its instantiation in the Machine (effectively creating an anonymous class). This implementation allows us to reuse `BracketCounter` for the *–check* option, and delegates the equality check to the `BracketCounter` rather than implementing it in both the `Machine` and the `Checker`.

3.6 Check

Also using the `BracketCounter`, the `Checker` will throw a `BracketMismatchException` if it found either too many or too few Back instructions compared to Jump instructions. It also defines a callback function to run when it finds too much Back instructions. However, we need an additional check at the end to see if there was enough Left instructions.

3.7 Error handling: exit codes, exceptions

All error handling is done using exceptions now. We have got a package containing a bunch of exceptions which will be thrown whenever an unexcepted behaviour happens. Additionnaly, they define a method which will return an error code. The error code is the one defined in the specifications if it exists, otherwise it's arbitrary. Exceptions are then catched at the top-level in the `main()` for their error message to be printed and their error code to be returned.

4 Keep Calm and Take a Step Back

Implementing I/O instructions required us to add a new Io class, interfaced with the Machine. The impact was however quite limited thanks to the in-place architecture which was already similar. Implemention itself relies on Java's power to handle both standard input/output streams the same as file streams.

The short syntax was very easy to implement, since we prepared it in the first level (by creating this representation in the different instruction classes). Image syntax became a little bit harder to create. We first thought about an attribute which would be an array of three integers (RGB values). But we decided to adapt our representation to the BufferedImage.getRGB() method and to store the color value as one integer (in code, written in hexadecimal format) in the ARGB format (e.g : 0xFF000000 for black. The alpha channel value is always at the maximum (FF) since we only want full opacity colors). Then, reading a color was as simple as requesting the InstructionSet to return the Instruction corresponding to the result of BufferedImage.getRGB() method. Another issue arose when we had to load the image: we initially wanted to use `Stream<>` for both text and image data, but because of type erasure, we couldn't overload methods and keep the underlying object type at runtime. Our workaround was to use an `IntStream` instead of `Stream<Int>` for the image.

Code modularity is great when it pertains to instructions. Adding one or altering another does not require much effort as all you have to do is create a new subclass of instructions and instanciate it in the `InstructionSet`. However, integrating a new file format isn't that easy: after adding the class to read it, you need to implement a parsing method in `InstructionParser`, then add support for the file type in the

`ArgParser` (in order to distinguish between the different formats, currently by the file extension) and finally handle the loading in the Launcher, adding a new case and a new method. This isn't great at all and should be reworked, by using similar to our `InstructionSet` for example, which would limit the change to adding the class for the new format support and registering it in another. The same could be said with the different modes to a lesser extent. Then should all have their own class (which is more or less the case but not clearly defined) and registered only once with the corresponding arguments on the command line.

Based only on the meaning of the instructions, the rewriting mechanism is bijective. However, when looking at the actual representation, it is clearly not because it is not even injective: different source files can lead to the same rewritten program because both the short and the long syntax will be rewritten to the short one. Also, the specifications limit the target set to the short syntax only.

[Some shit about the well-formed property algorithm of which I didn't understand a word…]

The linear approach to execute loop has clearly one flaw: it is not efficient, especially when you consider heavy loops which should be run many times. Also, considering a modern computer architecture, optimisations such as a branch predictor and instructions cache would not be efficient either: we don't know where to jump and we fill the cache with potentially useless instructions. Anyway our Brainfuck virtual machine is nowhere near competing with such an architecture...