Semester (kick-off) project: The BRAINF∗CK Language

Sébastien Mosser — `mosser@i3s.unice.fr`

2016 – Third Year – Polytech Nice Sophia–Antipolis

1 Introduction & Objectives

The goal of this project is to leverage knowledge you acquired in your previous life (before entering the department) and focus it on the implementation of a computational model. This project *does not require a strong technical knowledge in programming*, but requires that *you are understanding what you are doing*. It will boost your reasoning capabilities, through the definition of a minimalist programming language named BRAINF∗CK [Mül93], you will use knowledge from multiple fields of Computer Science and Software Engineering. This language contains only 8 instructions to interact with memory and input/output streams, and a very simple execution model. It is composed of 6 modules (M_i), layered on top of each others as described in FIG. 1a:

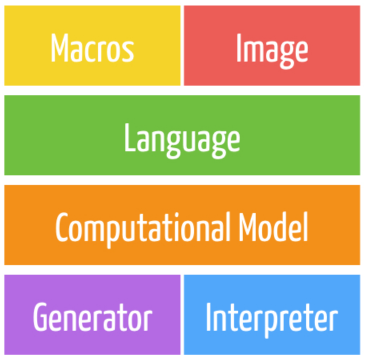
- M_1 The *computational model*, *i.e.*, the virtual machine able to store data in memory (*e.g.*, *incr*ementing the stored value, moving the memory pointer to the *left*) and interact with input and output streams;
- M_2 The concrete *language* itself, a minimalist text-based representation of BRAINF∗CK programs;
- M_3 The *interpreter*, able to read a program, execute it and output the memory contents at the end;
- M_4 The *image* reader, as we will use bitmap images as alternate representation of programs to be executed;
- M_5 The *macro* expansion mechanism, which will save us precious instructions by introducing higher-level concepts to ease the definition of BRAINF∗CK programs;
- M_6 the *generator*, used to transform a BRAINF∗CK program into an equivalent code in another programming language (*e.g.*, C, Java, Python)

1.1 The Software Engineering point of view

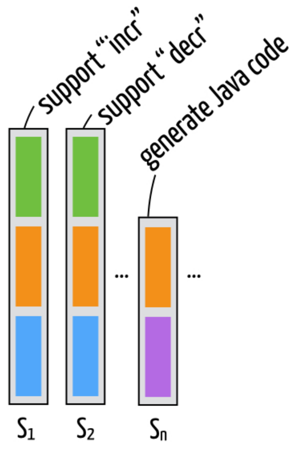
The project specifications are, for this very project only[1], already dispatched into several *vertical slices*. We will not build the whole computational model, then the language, then the interpreter, then the code generator, ... This *horizontal* way of developing a piece of software layer by layer is the best way to deliver a product that does nothing valuable at the end, considering that if one of the layer is not present at the end the complete product will not work (think about a tower made with *Jenga* bricks).

Instead, we will define this piece of software by iterating over multiple *slices* of the specifications. Contrarily to a layer that contains the whole features at a given technological level, a *slice* is rather thin in terms of functionalities, but cross-cuts at least several (if not all) layers. It actually implements a demonstrable feature.

[1]It will be **your** job in the upcoming ones to perform such a task.



(a) Layered Architecture



(b) Features as *slices*

Figure 1: *Layers* versus *Slices* in software development

For example, consider the feature *"Support the incrementation of a data by one"* (denoted as S_1 in FIG. 1b). Developing the code that implements this feature requires to improve 3 modules: *(i)* the language itself to add the new keyword, *(ii)* the computational model to change the contents of the memory and finally *(iii)* the interpreter that will react to the new keyword and do the right change in the model. As a consequence, after the completion of each *slice* S_i you will have a working product, ready for demonstration: $Product = S_1 \bullet S_2 \bullet \cdots \bullet S_n$. Introducing a new demonstrable feature in your product will then be as easy as developing a new *slice*: $Product' = Product \bullet S_{n+1}$.

To support your teams during the development of this project, you will benefit of the facilitation framework available in the school. It consists for this project in *(i)* weekly meetings, *(ii)* intermediate demonstrations and *(iii)* serious game workshops. For the upcoming projects, you will also benefit from the complete framework, *e.g.*, ticketing system, version control, external coaching, industrial experts.

1.2 Role of the project in your curriculum

This project binds together various fields, from theoretical ones to very applied ones. The list below is a non exhaustive list of topics you will encounter during your studies in the department, related to this project:

- Discrete Mathematics (3^{rd} year, S_5): This part of the "Theoretical computer science #1" course will discuss about the definition of languages, the associated notions of alphabets and words. It will also address the inductive definition of functions;
- Object-oriented programming (3^{rd} year, S_5): This course will identify the use of object abstractions and interfaces to support the modularity and the extensibility of a given piece of software;
- Principles of Programs Execution (3^{rd} year, S_5): This course describes how a computer execute a program, which is the essence of this project. It will also discuss data representation in memory. The final lab sessions of this course are dedicated to the definition of an assembly language on top of Arduino electronic boards, which is the natural continuation of this project on a real-life processor;

- Procedural programming (3^{rd} year, S_5): This course uses C as the implementation language. The preprocessing mechanisms available in C are very close to the one you will implement in the *macro* module;
- Formal languages (3^{rd} year, S_6): This part of the "Theoretical computer science #2" course targets the definition of formal languages, as well as the classification of languages into several classes, based on their properties;
- Compilation (4^{th} year, S_7): In this course, you will learn how to compile a set of real-life languages, as what we will see in this project can only be seen as a quick and dirty introduction to compilation, considering a very restricted subset of programming languages;
- Complexity & Computability (4^{th} year, S_7): This course will define what a Turing-machine is, which is actually the theoretical model underlying the BRAINF∗CK language. It will also discuss what it means to "compute" something, what is computable and what is not, as well as the relationship between these questions and the Turing model.

2 The BRAINF∗CK virtual machine

This section describe the virtual machine specifications. These information must be respected *by the book*, and considered as the definition of an existing standard. Your product will be assessed using programs you did not write yourself, and as a consequence it must respect this contract.

2.1 The computational model

The computational model is very simple. The BRAINF∗CK machine has a finite ordered set M of $30,000$ memory cells denoted as c_i and indexed by zero: $M = [c_0, c_1, \dots, c_{29999}]$. Each cell can contain up to 8 bits of unsigned data (denoted as d_i), *i.e.*, $\forall i \in [0, 29999], d_i \in [0, 2^8 - 1]$. Storing a value that is out of bounds in a given cell will broke the machine, *i.e.*, immediately stop the current execution and output an error code. Initially, all the cells are initialized with a 0. The machine is connected to an input stream denoted as in, and an output stream denoted as out. By default, these streams are the standard ones.

A program is defined as a word of the BRAINF∗CK language, defined as a sequence of instructions indexed by zero $P = [i_0, i_1, \dots, i_n]$. There is no restriction on the length of a program (denoted as $|P|$), meaning that the empty program $empty = []$ is valid, as well as a program of *very large* size. The execution context of a program P is a tuple $C_P = (M, p, i)$, where *(i)* M is the previously defined memory, *(ii)* p is a pointer to the memory cell currently used by the program ($p \in [0, 29999]$, initially $p = 0$) and finally *(iii)* i is a pointer to the next instruction to be executed in the program (initially, $i = 0$). After the execution of each instruction (excepting for the conditional ones), the execution pointer is automatically incremented by one. The execution of a program P stops naturally when the execution pointer i goes after the latest instruction (*i.e.*, $i = |P|$). When the machine stops, it prints on the standard output the contents of its memory (excepting cells containing zeros).

The BRAINF∗CK language defines an alphabet of 8 instructions, divided in 4 categories. Each instruction interacts with the current context and produces a new one. The execution semantics associated to the instruction set is formalized as the following:

- Manipulating the value stored in the current memory cell:

  i_1: *Increment*. This instruction increment the pointed memory cell by one, triggering an error if the values goes out of bound (*i.e.*, $d_p = 255$):

$$\text{Let } C_P = (M, p, i),\ Increment(C_P) = (M', p, i + 1),\ d'_p = d_p + 1$$

i_2: *Decrement*. This instruction decrement the pointed memory cell by one, triggering an error if the values goes out of bound (*i.e.*, $d_p = 0$):

$$\text{Let } C_P = (M, p, i),\ \mathit{Decrement}(C_P) = (M', p, i+1),\ d'_p = d_p - 1$$

- Moving the memory pointer:

  i_3: *Left*. This instruction moves the memory pointer to the left, triggering an error if the pointer goes before the first instruction (*i.e.*, $p = 0$).

  $$\text{Let } C_P = (M, p, i),\ \mathit{Left}(C_P) = (M, p', i+1),\ p' = p - 1$$

  i_4: *Right*. This instruction moves the memory pointer to the right, and trigger an out of memory error if the pointer goes after c_{29999}.

  $$\text{Let } C_P = (M, p, i),\ \mathit{Right}(C_P) = (M, p', i+1),\ p' = p + 1$$

- Interacting with the external world using side-effects on input and output streams:

  i_5: *Out*. This instruction prints out the value stored in the currently pointed memory cell (*i.e.*, d_p) as an ASCII character[2]. For example, if $d_p = 97$, the machine will print "`a`" (without the quotes) on the output stream.

  $$\text{Let } C_P = (M, p, i),\ \mathit{Out}(C_P) = (M, p, i+1) \wedge \mathit{out} \leftarrow d_p$$

  i_6: *In*. Read a byte on the input stream, and store it inside the currently pointed cell, erasing its previous contents.

  $$\text{Let } C_P = (M, p, i) \wedge \mathit{read}(\mathit{in}) = v,\ \mathit{In}(C_P) = (M', p, i),\ d'_p = v$$

- Moving the execution pointer to support loops and conditional instructions. These instructions work as couples associated to one another. One cannot define a *Jump to* without an associated *Back to* (such a program is invalid). Nested definitions are supported, as each *Jump to* is bound to the corresponding *Back to*. The binding that exists between two instructions is modeled by a boolean function named $\mathit{bound}(i, j)$ that returns true if the instruction located at i is associated to the one located at j, false elsewhere.

  i_7: *Jump to*. This instructions is silently ignored if $d_p \neq 0$. On the contrary, it moves the execution pointer i to the instruction right after its associated *Back to*, denoted as $i' > i$.

  $$\text{Let } C_P = (M, p, i),\ \mathit{Jump\ to}(Cp) = \begin{cases} d_p = 0 & \Rightarrow \quad (M, p, i'+1) \wedge \mathit{bound}(i, i') \\ d_p \neq 0 & \Rightarrow \quad (M, p, i+1) \end{cases}$$

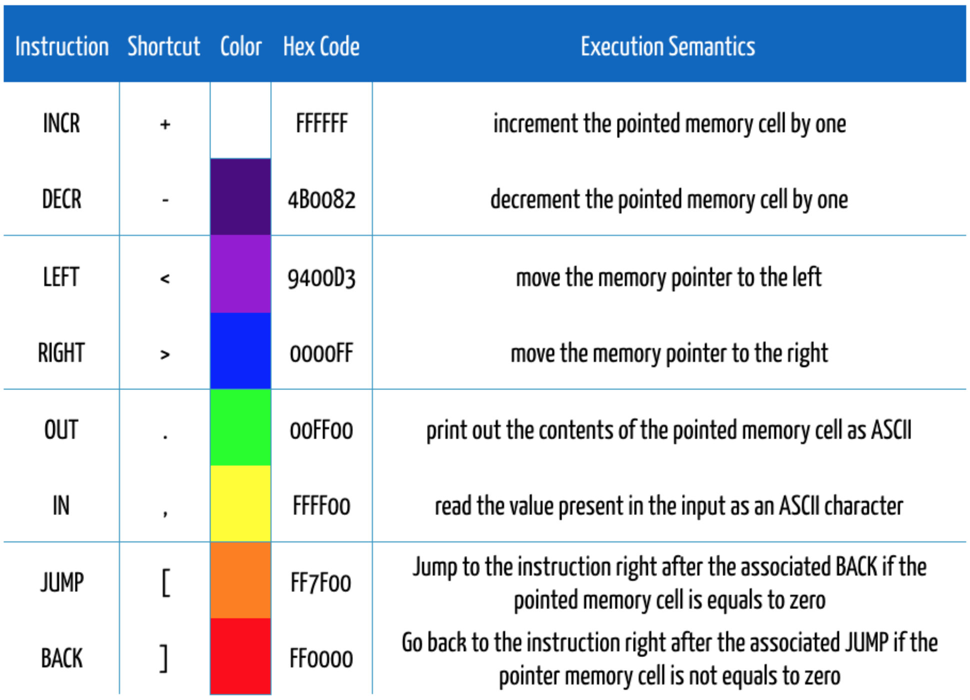
  i_8: *Back to*. This instructions is silently ignored if $d_p = 0$. On the contrary, it moves the execution pointer i to the instruction right after its associated *Jump to*, denoted as $i' < i$.

  $$\text{Let } C_P = (M, p, i),\ \mathit{Back\ to}(Cp) = \begin{cases} d_p = 0 & \Rightarrow \quad (M, p, i+1) \\ d_p \neq 0 & \Rightarrow \quad (M, p, i'+1) \wedge \mathit{bound}(i', i) \end{cases}$$

2.2 The concrete syntax of the BRAINF∗CK language

The concrete syntax of the BRAINF∗CK language is represented in FIG. 2. For each instruction previously described, the table defines its associated keyword (*Instruction* column) and a short text that describes the execution semantics in natural language. In addition, it defines for each instruction a shortcut notation and a color code to be used when reaching level two in the next section.

[2]As defined in the ISO/IEC 646 standard (`http://en.wikipedia.org/wiki/ISO/IEC_646`).



Instruction	Shortcut	Color	Hex Code	Execution Semantics
INCR	+	⬜	FFFFFF	increment the pointed memory cell by one
DECR	-	🟪	4B0082	decrement the pointed memory cell by one
LEFT	<	🟪	9400D3	move the memory pointer to the left
RIGHT	>	🟦	0000FF	move the memory pointer to the right
OUT	.	🟩	00FF00	print out the contents of the pointed memory cell as ASCII
IN	,	🟨	FFFF00	read the value present in the input as an ASCII character
JUMP	[	🟧	FF7F00	Jump to the instruction right after the associated BACK if the pointed memory cell is equals to zero
BACK	]	🟥	FF0000	Go back to the instruction right after the associated JUMP if the pointer memory cell is not equals to zero

Figure 2: Set of instructions available in the BRAINF∗CK language

Keep Calm and Take a Step Back

What are the limitations of the language? Can you already identify what will be difficult and what will be easy when implementing this project? Implement and interpret by hand short programs using this formalism, for example a program to add two integers. How can you represent in memory numbers greater than 2^8? What are the limitation of this computation model?

3 Compliance level expected by the customer

The customer who ordered the product defined 4 levels of compliance for the delivered products. These levels are composed of different development *slices*, already identified. A level is considered completed and delivered when all the slices inside this very level are accepted by the customer. It does not add any value to your product to deliver a slice of level L_i is level L_{i-1} is not totally delivered. After the first demonstration, the customer will deliver a *"compliance benchmark"*, defining several programs, the associated inputs and the expected outputs.

3.1 Level 1: Initiate

$$L_1 = S_0 \bullet S_1 \bullet S_2 \bullet S_3 \bullet S_4$$

The first level of compliance is composed of 5 business-driven slices. From a non-functional point of view, the product has to be delivered as a command-line tool, invoked using the `bfck` command. It will read the program to execute in a file (specified with the `-p` argument, using a non-existing file will trigger an error), and will print at the end of the execution the contents of the (relevant) memory on the standard output and an exit code of zero. If an error occurs, the error is printed on the standard error stream, and the systems returns an exit code greater than zero. Keywords are entered line by line, one keyword per line.

S_0: Supporting the empty program. Considering a file named `empty.bf` that is empty, the execution of the product should not produce any output or error.

```
azrael:~ mosser$ touch empty.bf
azrael:~ mosser$ ./bfck -p ./empty.bf
azrael:~ mosser$ echo $?
0
```

S_1: Supporting the `INCR` keyword. After the completion of this slice, the product is able to interpret a program that only contains the `INCR` keyword. For example, considering a file that contains 7 times the `INCR` keyword (one by line), the interpreter will output that c_0 contains the value 7. If one consider a file that contains the `INCR` keyword 256 times, the associated execution should trigger an error as the cell overflows. In this very case, the exit code of the command must be equals to one.

```
azrael:~ mosser$ ./bfck -p ./incrementC0by7.bf
C0: 7
```

S_2: Supporting the `DECR` keyword. This slice is very similar to the previous one, implementing the opposite operation. It triggers an error if trying to decrement a cell that contains 0, using the same exit code as the previous one (memory overflow).

S_3: Supporting the `RIGHT` keyword. This slice allows the program to move the memory pointer to the right. The importance here is to only print the memory cells that are relevant (*i.e.*,, $\forall i \in [0, 29999], c_i \neq 0$). Moving the pointer to the extreme right of the memory will trigger an out of memory error (exit code: 2).

S_4: Supporting the `LEFT` keyword. This slice is very similar to the previous one. It triggers an error if one tries to go left of the first cell (exit code: 2).

⏮ Keep Calm and Take a Step Back (L_1) ⏭

Think about the way the specifications of the BRAINF∗CK virtual machine were *sliced*. Can you explain why we started by this subset of the specifications? Was the implementation order important? Does it make sense to use another way of cutting this specifications into pieces? Can you define and justify another way of slicing/cutting the specifications? If yes, proceed. If not, what concepts are hidden in these choices? How would you proceed in another project?

3.2 Level 2: Padawan

$$L_2 = L_1 \bullet S_5 \bullet S_6 \bullet S_7 \bullet S_8 \bullet S_9 \bullet S_{10} \bullet S_{11}$$

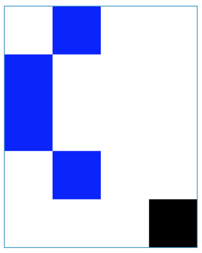
Considering that you are now an initiate, the slices will now be a little bit more thick. Reaching this level of compliance will give a BRAINF∗CK language interpreter that will support shorter programs using an alternative syntax, support input/output streams and loops, and allows one to store a program inside an image.

S_5: **Supporting shortened syntax.** Until now, we were just supporting one instruction per line, using the long keyword associated to each instruction. We want now to be able to support the "short" syntax associated to the BRAINF∗CK language (actually the one defined by the official specifications). This syntax relies on symbol encoded using one single character, and multiple symbols can be used on the same line. It is important to note that one can mix both syntaxes in the very same program.

S_6: **Rewriting instructions.** The interpreter now support both short and long notations. However, the short syntax is way more efficient in terms of storage. The `bfck` tool will now accept a parameter `--rewrite` that will print on the standard output the shortened representation of the program given as input (and not execute it).

```
azrael:L1 mosser$ ./bfck --rewrite -p s3_sample.bf
+>>>>+
```

S_7: **Using images to represent programs.** Instead of a textual representation, we can use an original way of storing our programs, *i.e.*, storing a program as an image. We consider each instruction encoded using a 3×3 square of the color defined in FIG. 2, and the final image is stored according to the BITMAP file format[3]. To fill missing pixels (if necessary), we use the *black* color. The first instruction is written in the top-left corner, and goes right and then down. We assume here that the file extension is `.bmp`. Examples of such programs are depicted in FIG. 3.


(a) P_1

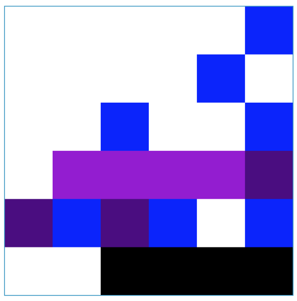

(b) P_2

Figure 3: Example of programs stored as *Bitmap* files

S_8: **Translating textual programs as images.** The BRAINF∗CK interpreter should now be able to translate a textual representation of a program into an image-based one, using the `--translate` argument. The translation does not interpret the program. It creates a squared image, with enough pixels to store the contents of the program given as parameter.

S_9: **Supporting inputs and outputs.** We still need to support the `IN` and `OUT` instructions of the BRAINF∗CK language. By default, the streams associated to the virtual machine are the standard ones. One can override this behavior using the `-i` and `-o` arguments to specify file names. The input file, if given, must exists (error code of 3 elsewhere).

[3]The Java Image API knows how to deal with it.

S_{10}: **Well-formed program.** Before starting to interpret the looping instructions, one might want to be sure that a given program is well-formed. The `--check` option if the interpreter will analyze the input program (but not execute it), and exit silently if the program is well-formed, *i.e.*, each `JUMP` instruction is bound to a `BACK` one. This problem is known as the "*well-parenthesized word*" problem, considering each `JUMP` as an opening parenthesis and each `BACK` as a closing one. If the checker encounters an error, it must exit with an error code (4).

S_{11}: **Executing loops.** Considering a well-formed program (the interpreter should exit with the same error code as the previous slice if the program is not well-formed), we should now be able to interpret loops in a BRAINF∗CK code. At this step, we consider a linear approach to move from one marker to the associated one: when a `JUMP` instruction is encountered and must actually jump to the associated `BACK` one, the execution pointer will move one instruction at a time until it reaches the right instruction.

⏮ **Keep Calm and Take a Step Back (L_2)** ⏭

What was the impact of adding the I/O instructions to your code? What was the impact of adding a new representation for programs, using images? Discuss about the modularity of your code, to what extent is it easy to extend it? What about the rewriting mechanism, is it injective, surjective, bijective? Does it really matters? The well-formed property is very important for a program. Describe a recursive and an iterative implementation of this algorithm, and discuss the differences between the two approach: it does the same thing, but how does it impact your code? Do you foresee any issues relative to the linear approach used to execute loops in the interpreter?

3.3 Level 3: Jedi Knight

$$L_3 = L_2 \bullet S_{12} \bullet S_{13} \bullet S_{14} \bullet S_{15}$$

Congratulations, you are now an achieved *Padawan*. To reach the *Jedi Knight* level, you will have to instrument your code to support developers, by computing metrics, optimizing the execution engine and supporting more complex constructions in the input programs.

S_{12}: **Introducing metrics.** To measure the different performances associated to alternate implementation of the very same program, one wants to retrieve metrics about the execution of a program after its execution. These metrics are:

- *PROG_SIZE*: the number of instructions in the program;
- *EXEC_TIME*: the execution time of the program, in milliseconds;
- *EXEC_MOVE*: the number of times the execution pointer was moved to execute this program;
- *DATA_MOVE*: the number of time the data pointer was moved to execute this program;
- *DATA_WRITE*: the number of time the memory was accessed to change its contents (*i.e.*, `INCR`, `DECR` & `IN`) while executing this program;
- *DATA_READ*: the number of times the memory was accessed to read its contents (*i.e.*, `JUMP`, `BACK`, `OUT`)

S_{13}**: Logging steps to trace execution.** It is (very) difficult to debug a BRAINF∗CK program[4]. To support the developer, we propose to write a trace of everything that happen during the execution of a given program in a companion file. When invoked with the `--trace` option, the interpreter will create a file named `p.log` (considering the execution of `p.bf`) that contains the following information: the execution step number (starting at one), the location of the execution pointer after the execution of this step, the location of the data pointer at the very same time, and a snapshot of the memory.

S_{14}**: Supporting code comments & indentation.** Still wanting to help the developers, we need to add some support to the concrete syntax of the language. It is important that one can indent loops to figure out what is happening ins a given program. Code comments will also help. A comment starts with a `#` and ends at the end of the current line. One can use whitespaces or tabulation to support code indentation.

S_{15}**: Code macros.** The concrete syntax is an immediate translation of the computational model capabilities. However, as the model is simple, it is not really user-friendly to write program using this syntax. Without changing the underlying model, we want now to add higher-level instructions. For example, each time a user enters an integer value using the input stream, we need to subtract 48 to it (as 48 is the ASCII code for 0, 49 for 1, . . .). Instead of writing 48 times the `DECR` instruction, one might want to write `TO_DIGIT`. A macro is considered as a *long* instruction (alone on a single line).

S_{16}**: Parameterized macro and recursive expansion.** The `TO_DIGIT` macro is actually a macro on top of another one, a macro that decrements the current memory cell by $n \in \mathbb{N}$. The macro expansion should first reduce `TO_DIGIT` into `MULTI_DECR 48`, and then reduce `MULTI_DECR 48` into the right BRAINF∗CK instructions to actually decrement the current cell 48 times.

S_{17}**: Interpretation optimization.** The linear approach we used to find a closing (or opening) marker when a jump is needed is not really efficient. We propose to slightly adapt the interpreter engine to store a *jump table*. This table implements the *bound* function defined in the formal specifications, and can give for an execution pointer location related to a *loop* instruction. Using the jump table, the execution engine will then literally *jump* to the next location instead of moving index one by one and testing the symbol each time. The jump table has to be computed before starting the execution of the program.

⏮ Keep Calm and Take a Step Back (L_3) ⏭

What were the pros and cons of using the object-oriented paradigm for this project? What is the role of the macros in the language specifications? How can you empirically measure the optimization benefits? What will happens in case of recursive macros? What was the impact of introducing the metrics in your code? What does the jump table cost to your interpreter? Does it worth it? Some metrics can be statically computed in given cases (*e.g.*, no conditional instructions). Does it make sense to separate the two implementations? Do you see other things one can optimize in the virtual machine?

[4]Actually the language itself was designed to be difficult to use, on purpose.

3.4 Level 4: Jedi Master

You are now an accomplished Jedi Knight. To master the final level, you will have to demonstrate your capability to take initiative when confronted to difficult problems. The specifications for this level are fuzzy on purpose. You must now be able to take decisions, and explain why these decisions were the right ones.

S_{18}: **From macro expansion to procedure calling.** Macros are limited by the fact that the code of the macro is replicated each time the macro is used. This can lead to very verbose programs at the execution level. the opposite way of implementing behavioral reuse is to consider procedure. Contrarily to a macro, a procedure is written only once in the program, but executed several times, each time one call it. Imagine a way to declare procedures in your programs, and to call such artifacts. You can reuse the `TO_DIGIT` example from slice S_{15}, among others.

S_{19}: **Procedures with parameters.** Improve your virtual machine to support procedure with parameters. How to specify procedure arguments, how to send these arguments to the procedure?

S_{20}: **From procedures to functions.** The next step is to improve your virtual machine to allow the computation of return value from your procedures, transforming such artifacts into functions.

S_{21}: **Code Generation.** For now, we are interpreting the program directly inside the virtual machine. It should be possible to generate code in another language (*e.g.*, C, Java, Python) that might be way more efficient than the interpretation of the language.

⏮ Keep Calm and Take a Step Back (L_4) ⏭

What are the assumptions you made in your product to support procedures and functions. What is the impact of these assumptions with respect tot he expressiveness of your language? Can you build the dependency graph of the different slices you delivered? Does it make sense?

References

[Mül93] Urban Müller. Brainfuck, an eight-instruction turing-complete programming language. *Wikipedia*, *`http://en.wikipedia.org/wiki/Brainfuck`*, 1993.