# "CMat: describing the language and its Python-implemented interpreter"

Final report

*Antonio M. Franques Garcia (franque2@illinois.edu)*

This is the final report of my term project for CS598, in which I (from now on I will refer as "we", for writing formality purposes) implement an interpreter for a blended subset of *Matlab*, *C* and *Cool*. The purpose of this project is to provide instructional material for a possible first-year Engineering/CS undergraduate class. This report is structured as follows: In **Section 1** we will first introduce and motivate the purpose of this project. **Section 2** will describe the final syntax of our designed language and break down the main ideas behind our implementation approach. In **Section 3**, we will get into more details on the *Python* implementation of our interpreter and also explain how and what has been accomplished in this project. In **Section 4** we will evaluate the performance of our language and interpreter relative to *C*, *Matlab* and *Python*, and also explain the reasons behind those numbers. Finally, **Section 5** will list what part of the project is subjected to possible future improvements.

## 1. Introduction

Oftentimes first year undergraduate students are unable to see the reasons why some topics are taught to them, therefore decreasing their interest and expectation towards that topic if it's not properly motivated beforehand. That is mainly due to the fact that, unlike their professors, they don't have enough experience as to see the whole picture and hence understand why those topics are so necessary later in their careers. In the case of Engineering and Computer Science studies, for example, this concept takes even a more relevant importance, since a considerable part of those students already have some basic knowledge about concepts and tools, like programming, which can create a false overconfidence on them if they think that what they know about it is almost enough as to be ready for the labor market.

The aim of this project is to provide enough instructional material for a possible first-year Engineering and Computer Science undergraduate class, which would properly motivate students to the need to learn some more advanced concepts in their future classes. In this new undergraduate class proposed, both kind of students (those who already have some basic knowledge about what programming is, and those who have not programmed ever before), would be at first introduced, in order to get more confident in basic programming, to a very limited but yet intuitive language, called ***CMat***, which would let them translate to a computer language some of the rudimentary math problems that they were taught to manually solve during their high-school studies (e.g. finding all prime numbers up to a certain one, obtaining the greatest common divisor of two numbers, calculate a Fibonacci sequence or even obtain the Mandelbrot set over a small area); all this under the obvious motivation that computers let us automate and accelerate procedures that otherwise human should manually and slowly solve over and over again. But not only this, right after the before mentioned introduction to basic programming through *CMat*, the students of this future possible class would go one step further by also learning what happens underneath programmer's view when they finish writing their source code and click that "compile & run" button on their programming environments, i.e. the interpreter/compiler. For that, they will first learn what a "token" is, and later, together with their intelligence and abilities, they will learn how to infer from a list of tokens representing a certain *CMat* source code (**Section 3.1**), what structure was actually written and meant by the *CMat* programmer. At the same time, in order to accomplish the just mentioned interpretation of tokens, students will go even one step further from that *CMat* syntax

learned at the beginning of the project and realize that they need some extra tools to fulfil their aim, since plain code will not provide them enough power as to construct the whole interpreter with it. Therefore, they will learn how to create and call methods and classes (**Section 3.2**); this will be their properly motivated introduction to Object Oriented Programming through a dynamic language like *Python*. All together will give as a learning experience the fact that programming in a dynamic language like *CMat* is easy but the performance (as we will see in **Section 4**) decreases a lot if its interpreter is not correctly implemented (this will motivate the future learning and use of ASTs for the interpretation process) or even more if we compare it to statically typed languages (this will motivate the future learning and use of *C*, *C++*, *Java*…). Due to this, students will understand why during their *Python* implementation of their *CMat* interpreter they needed to learn the concepts of methods and classes as an extra tool besides the well-known basic plain code (meaning by plain code: arithmetic operations, variable assignments, "`if-else`" structures, loops…), and in their future courses see why learning statically typed languages is so important, even if they are more tricky to learn and use. Also, they will now understand why compilers are a so important part of computer science, and why they will need to find a way better tool to implement it, rather than just arbitrarily process a list of tokens (as they did for this interpreter), if they want its performance to increase and get closer to the well-known languages like *Matlab* and *C*. For this, instead, they will have to learn a much better method next time: the Abstract Syntax Tree.

The rest of this report describes, into more detail and in the same order as it should be taught, the content of the so mentioned class material.

## 2. Description of *CMat*'s language syntax

As mentioned in the introduction, ***CMat*** is an out-of-the-box mixed subset of *Matlab*, *C* and *Cool* languages. Its final syntax can be described as:

```
expr | FLOAT
     | STRING
     | ID
     | ID <- expr
     | ( expr )
     | expr  * | / | + | -  expr
     | expr  < | <= | > | >= | = | != | and | or  expr
     | if ( expr ) { expr[, expr]* } [else { expr[, expr]* }]
     | for ( expr, expr, expr ) { expr[, expr]* }
     | while ( expr ) { expr[, expr]* }
     | function( expr[, expr]* )
```

(Additionally, comments in the code must start by "%". Any line or remaining of a line that follows a "%" symbol will be totally neglected until the beginning of the next line. There is no comment block special notation, however the user is free to put a "%" at the beginning of as many consecutive lines as desired)

This chosen syntax, at the same time that it can seem arbitrary, is based on the experience of what some engineering-oriented first-time programming learners described as intuitive, according to the way they usually break down, write and describe problems by hand. As a few examples to that:

- Assignments to an identifier (variable) are denoted by a left term (the identifier name) and a right expression (value to be saved), separated by an arrow (←) pointing to the left term (notation influenced by *Cool* language). The reason behind this syntax notation refers to the fact that in math problems we usually use an arrow to specify that a certain group of things (in this case the right expression of the assignment) "refers to" or "can be addressed by" some other term (in this case the identifier). Also, this way we avoid confusing between assignment and equality ("=", which is used as a Boolean operator).
- Arithmetic (`+`, `-`, `*`, `/`) and Boolean (`<`, `<=`, `>`, `>=`, `and`, `or`…) operators clearly follow the same standard notation as in regular math notation.
- Method dispatch (function call) also follows conventional math notation, e.g. if we write the mathematical expression `a←f(b,c)` we are saying that "`a`" is the result of a certain function "`f`" operating somehow over parameters "`b`" and "`c`", which is exactly the same notation used in *CMat*. It is relevant to highlight that also choosing the right names for function calls is important in order to provide an intuitive user experience. For this reason *CMat* functions receive the same names as those from *Matlab*; software that most likely all engineering students will use later on their careers.
- As it refers to loops and "`if-else`" structures, it has some room for discussion to either if it's better to use curly brackets or rather some other notation for delimitating the boundaries of its bodies, however given that *C*, *C++* and *Java* (the main languages that students would learn afterwards) use this notation, it seems logic to keep coherence with that, in order to ease the transition to them.

On the other hand, as it refers to the implementation of its interpreter, *Python* is the chosen language. These are some of the main reason behind this decision: *Python* is dynamically typed, hence easing the process of programming with it (by not having to specify and get restricted to a certain data type in variables, nor making definitions before their uses), *Python* has some native-implemented functions that will ease the process of tokenizing *CMat*'s source code, and last but not least, because it's important that students start getting used to a neat indented and well organized code, as it's mandatory to be in *Python*.

The strategy breakdown for tackling this *CMat* interpreter in *Python* is:

1) Read *CMat* source code as a string of characters
2) Text-match it according to a predefined list of regular expressions in order to get the original code tokenized
3) Interpret directly the whole vector of tokens (without an AST)

## 3. **Description of *CMat*'s interpreter implementation**

A more detailed description of the strategy breakdown just shown at the end of the previous section is given on the following 2 subsections.

### 3.1 Structure of the *Python* code and startup of the interpreter (steps 1 and 2 of the strategy breakdown)

All *Python* code is written in a single file, called `myinterpreter.py`, and it is divided into 4 main sections:

The first section contains the import of lexer's needed libraries as well as some global variables, such as:

- The constant translations of the token types for easiness of programming

```
FLOAT='FLOAT', WHILE='WHILE',…
```

- The regular expressions' definitions vector

```
regular_exprs = [ (r'\"[^\n]*\"',    STRING),
                  (r'\<-',           ASSIGN),
                  …]
```

- The symbol table (hash table in which we are going to store all the identifiers of the program and its values).

The second section of the code defines the class `Token`, which is going to be instantiated every time a token is found. Every `Token` object will contain the type of the token (`STRING`, `IF`, `FLOAT`, `LPAREN`…) and its value ("`mystring`" , "`IF`" , "`3.1415`" , "`(`"…). The second section also contains the definition of the lexer function, which receives as an argument the string of characters representing the whole *CMat* code and the vector of regular expressions' definitions (`regular_exprs`). This function (`lex`) makes use of some native *Python* libraries for text-matching the *CMat*'s source code string of characters according to the mentioned list of regular expressions. As a result, a `Token` object is created for every match, and appended to the tokens vector that will be returned as a final result of the `lex` function. It's worth mentioning here that such a tokens vector is actually subdivided into smaller subvectors, each one delimited to the next for every time that a semicolon is found. Such a division will ease the startup code defined in the `main` method (section 4 of the code).

Section 3 of the code contains the definition of the `Interpreter` class. An instance of this class will be generated every time an expression has to be interpreted as a whole, regardless of how small (e.g. a simple primitive) or big (e.g. "`if-else`" structure) it is. The initialization of `Interpreter` objects will be given by the input argument: a vector of tokens to be interpreted as a whole. The purpose of this class is to provide independence and recursion amongst the interpretation of different expressions. This way, an `object1` of the class `Interpreter` that is interpreting a certain `vector1` of tokens and at some point needs to independently interpret a subvector (`vector2`) of `vector1`, will recursively create another object (`object2`) of the same class `Interpreter`, interpret it without affecting the state of the `object1` interpretation, provide its result back to `object1` and resume `object1` interpretation. A further description of the class `Interpreter` will be given in Subsection 3.2.

Finally, section 4 of the code contains the `main` method of the *Python* program, which is the one that is going to be first called when we invoke our interpreter. In this method we specify the "startup" instructions that will be executed when `myinterpreter.py` is invoked from the *Python* shell. The format of the invocation, considering that we are in a Linux terminal, is: `python myinterpreter.py myCMatSourceCode`. This will run (actually interpret, because *Python* itself is also an interpreted language) `myinterpreter.py` with `myCMatSourceCode` as an input argument. This way, `main` method will save the path to the *CMat* source code into a variable, open and prepare it to be read as a string of characters, call the method `lex` (described in the second section of the code) in order to tokenize it, save the returning vector of tokens, create an `Interpreter` object

(according to section 3 of the code) for each subvector of tokens (for each expression delimited by a semicolon, as explained two paragraphs above) and finally call the main method of `Interpreter` class (`expr`) for each one of them. The call to the `expr` method of `Interpreter` class will automatically call any other necessary method of `Interpreter` class or even recursively create another instance of it if it's necessary to process any inner expression (like an expression inside parentheses or a predicate in an "`if-else`" statement).

3.2 Methodology of the `Interpreter` class (step 3 of the strategy breakdown)

Let's recall that an instance of the `Interpreter` class will be generated for every time that an expression wants to be independently interpreted from the rest (independently in terms of the vector of tokens, not in terms of the *CMat* variables; *CMat* variables are all global across the whole *CMat* program), producing as a result, the execution of it (e.g. an assignment or a method dispatch) and an optional return value (e.g. the result of some arithmetic operation, or the result of a method dispatch). So, what's the methodology followed when an `Interpreter` object is created and right after its `expr` method called?

First of all, at the moment of the object creation, the constructor of `Interpreter` sets some variables (shared between all methods of the same `Interpreter` object) like `tokens`, which will contain the vector of tokens to be interpreted; this is received as an argument either from the `main` method (the first time that an expression delimited by semicolon is requested to be interpreted) or from some method of the `Interpreter` class itself (every time that we need to recursively interpret part of the actual vector, e.g. when we want to evaluate the predicate of an "`if-else`" and then depending of its result proceed in one way or another for the rest of the "`if-else`" interpretation). Also, the constructor of `Interpreter` class sets three other variables, one that will always contain the `Token` (notice that we capitalize the first letter of `Token` to highlight the fact that we are referring not only to the concept of token itself but also to the fact that `Token` is an object of the class `Token`) of the vector that we are dealing with at a given time (initially set to the first `Token` of the vector), another that will always save the position of the vector (1st `Token`, 2nd `Token`, …) that we are dealing with at a given time (initially set to 0, for coherence with *Python*, because *Python* vector index starts at 0, not at 1) and another one which is a method dictionary (this hash table will call the right method of `Interpreter`, when a *CMat* method dispatch occurs). Right after creating an `Interpreter` object, we always call its `expr` method, which will call the rest of the methods of `Interpreter` class when needed, or even create a new `Interpreter` object, and return its result after its whole execution. When `Interpreter.expr` is called, which only occurs one time per expression to be interpreted (it can recursively call it again, but that would be with another expression, which although would be a part of the first it would be considered as an independent new one), it checks (using a long `if-elseif-elseif-else`…) which type of token it is.

*Notice from the syntax table described in Section 1 of this document that all different expressions start with a different type of token, therefore we'll run different schemes for different expressions depending on which is its first token type. The only exception to that is the case of the `ID`, which can either be an identifier call (loading the identifier value) or an identifier assignment "`ID <- value`" (overwriting the identifier value). This exception will be taken care of by checking if there is an assignment token following the `ID` token and distinguishing both procedures depending of the result of that check.*

On the following it's listed the different schemes executed (in the same order they appear in the code) depending on the first token type of the expression to be interpreted:

- If token is of type identifier (variable) and next token is of type assignment we evaluate the right part of the assignment (we create a new `Interpreter` object with the right part of the assignment expression as an argument) and save that value into the proper entry of the symbol table (we make a lookup with the token value of the left part of the assignment, which is the identifier name in which we want to store)
- If token is of type identifier (but without assignment afterwards), float, or left parenthesis it means that we are either in a Boolean or an arithmetic operation. Then, if the length of the expression is 1 we check if it's either an identifier or a float literal, and we return its value according to that, however if the length is longer than 1 we check if there is an "`and`" or an "`or`" outside parentheses, and in that case we evaluate the left part and the right part of it independently (we create two `Interpreter` objects) and "`and`"-ize or "`or`"-ize the results afterwards. If neither an "`and`" nor an "`or`" was found outside parenthesis we do the same with the "<", "<=", ">", ">=", "=", "`!=`" symbols, and if it doesn't find any of them neither we repeat the search again with "+" (addition), "-" (substraction), "*" (multiplication) and "/" (division). Notice that this way we are establishing precedence of division over multiplication, multiplication over subtraction, subtraction over addition, addition over comparison operators and comparison operators over Boolean-concatenation operators. Finally, if neither of the operators was found outside parentheses it means that we have to go one level deeper, therefore we remove the parentheses on both sides of the expression and evaluate its content (we create a new `Interpreter` object initialized with a vector containing all the tokens inside those parentheses).
- If token is of type string we simply remove the quotes on both sides of its value and return it
- If token is of type "`if`" we create a vector (let's call it `vector1`) containing all the vectors inside the predicate (we append until we find the uppermost right parenthesis) and then we evaluate it (as always, by creating an `Interpreter` object initialized with the vector of tokens that we want to interpret, in this case `vector1`, and executing its method `expr` afterwards). If the evaluation results on a "`true`" we will create an empty vector of vectors which will be filled by all the tokens inside the "`if`" body, delimited from vector to vector by the "," (coma) token (notice that expressions inside "`if-else`" body are separated by comas, not by semicolons), until the uppermost right curly bracket is found; once the whole body has been traversed we will interpret each one of the subvectors by creating an `Interpreter` object (and executing its `expr` method afterwards) for each one of them. If the predicate resulted to be false we won't neither save nor execute the "`if`" body, instead we'll check if there is an "`else`" token right after the "`if`" body, and if it's the case we'll save and interpret its body (analogously as we did with the "`if`" body). Finally, if neither the predicate was true nor there was an "`else`" statement (notice that the "`else`" statement is optional), we will not do anything else.
- Similarly as in the case of an "`if`", if token is of type "`while`" we evaluate the predicate by creating an `Interpreter` object with all the tokens inside the predicate limits (until we find the right-most parenthesis), executing its `expr` method, and supposing it results on a `true`, save all tokens inside its body, separate them in subvectors delimited by comas, independently

execute each one of them and finally reevaluate the original predicate; repeating the whole process over and over until the evaluation of the predicate results on a `false`.

- A "`for`" loop is somewhat a "`while`" loop but adding an assignment previous to the start of it and adding an assignment after the end of each iteration. In order to address this difference we check if the token is a "`for`", and in that case we separate the predicate into three subvector of tokens, each one delimited to the other by a coma, and the last one delimited to the body by the right-most parenthesis. Then, we evaluate the first subvector, which is the initial assignment, followed by the evaluation of the second subvector, which is the one that we are going to use during the next evaluations of the loop, and finally, at the end of each body iteration, we also run the the third subvector of the "`for`" predicate. This process (without the evaluation of the initial assignment) will repeat over and over until the "`for`" condition (second subvector) produces a false result on its evaluation.
- Finally, if none of the previous token types were matched, we do a final check in order to see if the token is actually referring to a method dispatch, therefore if the token is of type "`method`" we will do a lookup into a symbol table specially created for this purpose, and see if the value of the "`method`" token (which is the method name) matches with any of the entries of the table. If it does, its corresponding *Python* method is called, providing as a parameters, the vector of tokens contained within the parentheses of the call. The just called *Python* method (there's one for each of the *CMat* possible functions) will then properly deal with the received vector of tokens, separating it first into several subvectors delimited by comas, and then properly evaluating them, giving as a result, the expected method output (e.g. if we write `Mod(a,b)` in our *CMat* program, the interpreter will first lookup the token value "`Mod`" into the methods symbol table, then execute the method `Modulo` that corresponds to that entry of the symbol table, which is a *Python* method expressively created to deal with `Mod` calls, take the whole input vector of tokens, which will contain the arguments `a` and `b` of the `Mod` call, separate them into two subvectors delimited by the middle coma, evaluate them separately, call the *Python* operator "%" and use it over the two evaluations just obtained, finally return the result of that). Besides "`Mod`", another function has been similarly implemented: `Disp` (display), which allows the user to display as many identifiers, primitive values and strings, as desired, through the standard output (usually the screen); this unlimited list of displayed arguments has to be separated from one to another by comas (as it has been usual in our notation so far).

To sum up, the main features implemented in our interpreter are: tokenizing of the source code into its proper type and value, processing of comments at either the beginning or in the middle of a line, separation of *CMat* source code expressions (delimited from one to another by semicolon) and independent interpretation of each one of them, variable assignments, variable calls, parentheses precedence, Boolean evaluation of predicates (lower, lower or equal, greater, greater or equal, equal, not equal, and, or), arithmetic operations (addition, subtraction, multiplication and division) with precedence amongst different operators, "`if-else`" statement (with optional "`else`"), "`for`" and "`while`" loops, all with possibility of more than one expression in its body (including other inner "`if-else`", "`for`" or "`while`" statements) and finally, function call with variable number of arguments (`Modulo`, to calculate the remaining after dividing two numbers, and `Disp`, to display any combination of strings, numbers and identifiers through the standard output, which usually is the screen).

## 4. Evaluation of *CMat*'s interpreter performance

We evaluated the effectiveness of this *CMat* interpreter by measuring the execution time of four different scalar micro benchmarks over it, and then compare these results with the ones obtained by executing the same micro benchmarks over *C* and *Matlab*. The evaluation was performed on a machine with an Intel Core i7-3537U (1st Gen) processor at 2.5GHz and 8GB of memory. The chosen Operating System is Ubuntu 14.04 (64 bits). We used gcc4.8.4 with optimization 3 enabled to compile *C* benchmarks. For our *Python* implemented interpreter and benchmark evaluation, version 3.5 was used, and finally, for the case of *Matlab* benchmarks, version R2014b was the one chosen.

The measured suite of micro benchmarks included: finding a Fibonacci sequence (`Fib`), obtaining a list of prime numbers up to a certain one (`Prime`), calculating the Greatest Common Divisor between two values (`GCD`) and obtaining the Mandelbrot set for a certain squared surface (`Man`).

On the following, four figures are shown, one for each of the four micro benchmarks, evaluated for several input values (ordered from left to right, from lower to bigger input values), comparing their execution time slowdown over C:

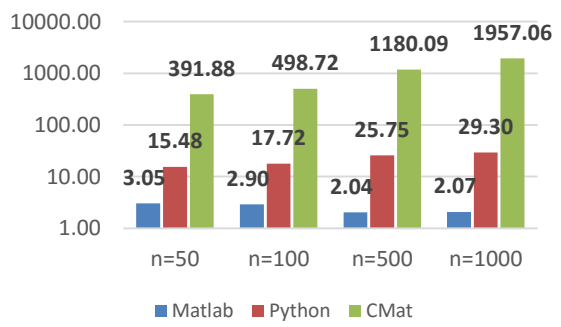


**Figure 1.** `Fib` micro benchmark

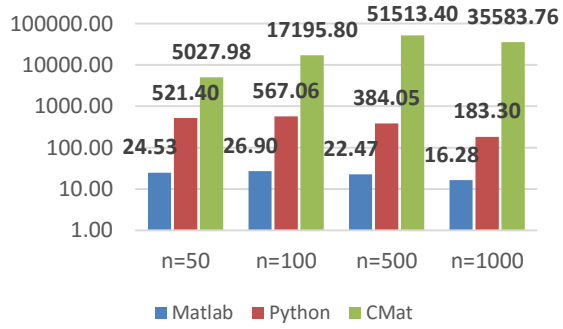


**Figure 2.** `Prime` micro benchmark

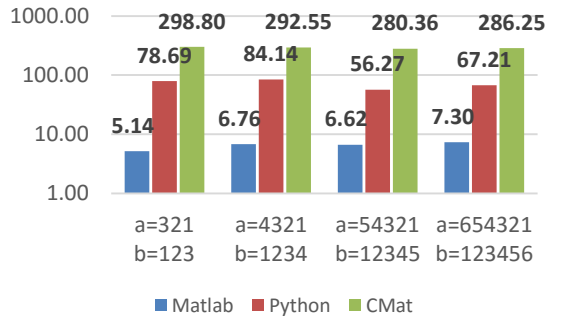


**Figure 3.** `GCD` micro benchmark

| | Matlab | | Python | | CMat | |
|---|---|---|---|---|---|---|
| | ET | SD | ET | SD | ET | SD |
| n=10 | 0.26 | 130.00 | 9.90 | 4951.5 | 7358.8 | 3.68E+06 |
| n=20 | 1.127 | 563.50 | 22.40 | 11203.5 | 32589.3 | 1.63E+07 |
| n=40 | 4.712 | 1570.67 | 57.81 | 19272.3 | 145884.5 | 4.86E+07 |
| n=80 | 19.264 | 6421.33 | 176.87 | 58957.0 | 577284.8 | 1.92E+08 |

**Figure 4.** `Man` micro benchmark (ET = Execution Time in miliseconds, SD = Slowdown relative to *C*)

First of all, it's important to highlight from these results that the contribution of *Python* itself is already quite high, therefore our interpreter (since it is Python-implemented) can't do nothing but to get worse. As expected, in Figures 1, 2 and 4 (`Fib`, `Prime` and `Man`), we observe that *CMat* experiences longer relative execution times as the input values get bigger (especially in the case of `Man`), this is because the bigger the input values are, the more iterations of the loop we have to run in order to obtain the final

value, therefore having to execute more times our poor structured interpreter. And because we have to go over the interpretation of every expression every time, the distance (in time) between our interpreter and our *Matlab*, *Python* and *C* references gets even bigger. In contrast, however, we see that even though it's quite high, the relative slowdown of *Matlab* and *Python* over *C* remains mostly constant (or even sometimes lower) over the execution of bigger input values, which demonstrates their great robustness, and shows us the way our interpreter should behave like.

As an exception to the behavior shown in Figures 1, 2 and 4, we have that in Figure 3 (finding the Greatest Common Divisor of two numbers), in which we deal with bigger numbers but not necessarily more iterations of the loop, also the relative slowdown of *CMat* remains constant over *C*. This actually does nothing but to show even better that the problem of our interpreter is not to deal with bigger numbers but to deal with more iterations (and thus, interpretations) of loops or structures, therefore having to go over our long `if-else-if-else…` checklist more times, in order to match the proper scheme and interpret it over and over again.

As a conclusion to this, we observe that our interpreter is mainly only useful for small scripts, in which we only want some easiness of programming or as an educational purpose, as in this case, to show that this is a poor implemented language, which gets worse and worse as the input values increases, whereas the other three (*C, Matlab* and *Python*) don't increase that much; see how the relative comparison between *C, Matlab* and *Python* doesn't almost increase as we increase the input values, whereas it increases a lot if we compare it with *CMat*. It is clear from here that if we want our performance to get closer (or at least relatively constant as we increase its demand) to that from other languages like *Matlab* or *Python*, we will have to drastically modify our implementation design, trying to find better-structured code and a better way to deal with the given source code. With all this, students are now able to see why they have to mainly learn on two strongly interconnected disciplines: programming languages and their interpreters/compilers.

## 5. Follow-up work

Not much can be done to improve the actual interpreter without drastically changing its whole structure, however, as an instructional way of keeping the students trying to figure out how to deal with a more complex expressions of *CMat*, at the same time than improving their skills on *Python*, it is suggested to add to the originally proposed language syntax, some extra features as the ones proposed on the following:

- Complex numbers notation, as in *Matlab*, through the use of the reserved word "`i`" (e.g. `3+4i`). This way, all arithmetic operations would change in a way such that they follow the main complex constraints. As a few examples (considering `x=(a+bi)` and `y=(c+di)`):
    - `x+y =(a+c)+(b+d)i`
    - `x*y =[(a*c)-(b*d)]+[(a*d)+(b*c)]i`
    - `x/y =` (a more complicated procedure)
    - `x<y =` (student will have to check if the *CMat* programmer is actually meaning the `|x|<|y|` operation of it instead it means to check if `a<c` or if `b<d`…)
- Array notation, through for example one-dimensional arrays (vectors) or multi-dimensional arrays (matrices and beyond). In order to achieve that, students would have to research the actual notation used in the common languages like *C*, *C++* or *Matlab* and see if it's intuitive to them, and therefore they want to use it in their *CMat* interpreter, or if instead they want to

change it for a better one. In either cases, they would have to now deal with the fact that identifier assignments and calls might be ambiguous, if the array position is not properly expressed by the programmer, and also learn and think which operators they can use and want to implement over them. Also, if they want to keep their *Python* code well structured, they should also take care to see how *Python* deals with arrays, given that after the interpretation of *CMat* translates to nothing else but to *Python* code.

- More functions, not necessarily only from *Matlab*, but also from other languages like *C*. By creating more *CMat* functions, students would have to learn how to deal with their internal operation, therefore increasing their *Python* and algorithms skills. As a few examples: modulus (to calculate the modulus of a complex number), rand (to produce a pseudo-random value), mean (to obtain the arithmetic mean out of a given vector of values), max/min (obtain the maximum/minimum value out of a given vector of values), sort (order in a certain order a given vector of values), plot (display a set of values into a certain map, through the standard output, according to some axis reference, which would let student learn how to deal with the graphics native libraries of *Python*)…
- Allow *CMat* programmers the possibility to create and call their own-programmed functions. For this, students would also learn how to modify the method's symbol table on the fly, automatically translate all the expressions into its body to *Python* code (according to some restrictions, of course) and describe the way to proceed with the input arguments on future calls of it.
- Scoping. Organize variables in a way such that they only live within their proper scope, e.g. in a method call, the method variables should only live within its body, unless declared as globals.

## Final note

Even though this report has been written in a paper-oriented format, I would like to spend these last lines expressing my personal opinion about what I learned with this project. Throughout the whole report I claimed that this project, due to its basic concepts and motivational conclusions, would be useful for first-year CS students, which is true. However, I would like to highlight that this also applies to other students like me that come from other fields (Telecommunications Engineering, in my case) and now want to integrate their knowledge with that of Computer Science. For this reason, excepting the part in which I use CMat to introduce students to programming, every step described above has been also a great learning experience to me. To be precise, these are the main concepts I learned through this project:

- On the language design: I realized of the important tradeoff between easiness of programming by the user and difficulty of implementation by the designer
- On the interpreter design:
    - I got better confidence in Python, and in general now with object-oriented languages, since I had to use classes, constructors, methods and objects for implementing my interpreter
    - I understood better the tokenizing process and its outcome, deciding that I wanted to keep all tokens representing the source code in a vector that I would use later on for the interpretation
    - Even when I was in the middle of the implementation I realized that the design that I was following (dealing directly with an ordered vector of tokens) was poorly structured and could not be easily scalable or reused for another language syntax. Therefore,

because of this and because of the performance results later obtained, I realized of the importance of a well-structured interpreter/compiler, starting by the use of an AST

- I learned how to properly evaluate the performance of my proposed interpreter versus other well-known languages (measuring only the time that the processor spends on this tasks, not considering the time that it's idle or assisting other tasks), also how to present these results. As a consequence of this I also realized how great and robust are languages like C, what also makes me understand the importance and tradeoff of statically versus dynamically typed languages
- I got better confidence on report writing and structuring

## References


[1] D. Padua. Set of lectures of CS 598 DHP Scripting Languages - Design and Implementation, fall 2015 (with a high relevance on Lecture 10 – Interpreters).

[2] Michael Dawson. *Python* Programming for the Absolute Beginner, 3rd Edition 3rd Edition.

[3] H. Wang, P. Wu, and D. Padua. Optimizing R VM: Allocation removal and path length reduction via interpreter-level specialization. In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, pages 295:295–295:305, New York, NY, USA, 2014. ACM.

[4] Terence Parr. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) 1st Edition.

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition).

[6] Ronald Mak. Writing Compilers and Interpreters: A Software Engineering Approach 3rd Edition.