Even Semester (2019)



**BINUS UNIVERSITY**



**BINUS INTERNATIONAL**



**Assignment Cover Letter**

**(Group Work)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | | |  |  |  |  |
| **Student Information**: | **Surname** | | |  |  | **Given Names**  **David**  **Livander**  **Muhammad**  **Zefanya** | **Student ID Number**  **2201797304**  **2201796384**  **2201797052**  **2201796870** |
| 1.  2.  3.  4. | **Amadeo**  **Surya**  **Erizky**  **Gedalya B. L. T** | |
|  |  |  |  |  |  |  |  |
| **Course Code** | **: COMP6340** |  |  |  |  | **Course Name** | **: Analysis of Algorithms** |
| **Class** | **: L3AC** |  |  |  |  | **Name of Lecturer(s)** | : **Maria Seraphina Astriani** |
|  |  |  |  |  |  |  |  |
| **Major** | **: CS** |  |  |  |  |  |  |
| **Title of Assignment**  (if any) | : **Zalgo Interpreter** |  |  |  |  |  |  |
| **Type of Assignment**    **Submission Pattern** | **: Final Project** | |  |  |  |  |  |
| **Due Date** | **: 29 - 10 - 2019** | |  |  |  | **Submission Date** | **: 29 - 10 - 2019** |

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer’s instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

# Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

# Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

**II. Table of Contents**

1. **Cover**
2. **Table of content**
3. **Background & Problem description**
4. **Implementation**
   1. **Formal description of the problem**
   2. **Design of algorithm**
   3. **Proof of correctness**
   4. **Complexity analysis**
5. **Evaluation**
   1. **Theoretical analysis of the algorithms**
   2. **Implementation details**
   3. **Data sets**
   4. **Results**
6. **Conclusion and recommendation**
7. **Program manual, how to execute (with screenshots)**
8. **Link to the application demo video (with max. length of 2 minutes)**
9. **Link to GIT website**
10. **Sources**

**III. Background & problem description**

We have categorized industrial milestones with stages. From the industrial revolution 1.0 all the way to the current revolution, Industry 4.0. Where we have smart cities, smart manufacturing, smart factory, smart devices and so many more on the way, where everything is online. Which means, the demand for these field is getting higher and higher. It has become a fact that people who live in this era will have an interaction in one way or another, with these “smart” systems.

It is the reality that now, in the world we live in, everything is a “smart” system and it will be literally everywhere. Integrated to everything. Now, it has become more apparent that the ability to understand, create and manipulate these systems in a deeper level is highly desirable. Especially for those in the teen to young adult range.

We believe that the first thing a coder should learn is not the code or syntaxes directly. But the first thing to learn is to have the ability to design a program, form logical thinking and framework of coding. They should be able to have the skills to design, think logically and be able to think like a coder. Because these skills are the those which are necessary to make any type of program or system, regardless of the language they use.

So, with Zalgo we hope to give a starting point for those who would like to study coding. Where the primary goal of using Zalgo is to make and design the backbone of the program.

**IV. Implementation**

**i. Formal description of problem**

Now that we have established the background, we can dive deeper into the core of the problem. We have seen drastic technological advancement through out the years and especially more to come in the future, we can see a larger demand for those who can create and manipulate these systems and moreover the interest will peak for these fields because of the increasing demands and the way society has shaped the world around us on technologies.

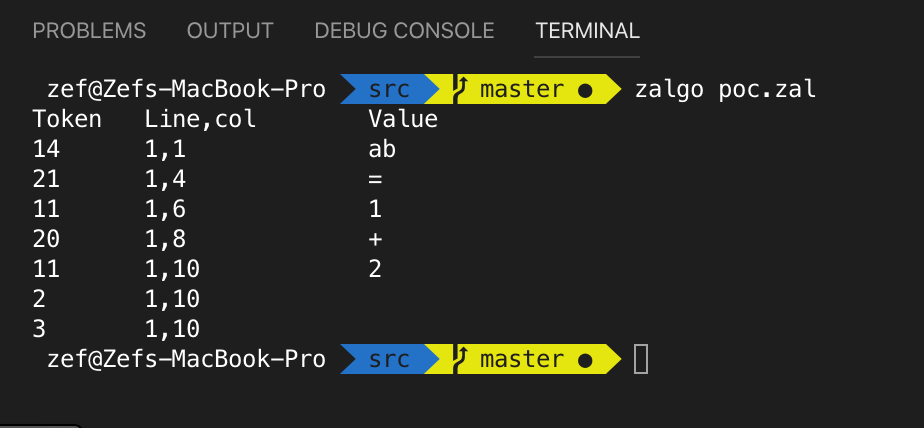
Hence with this, with new things coming up, new problems also arise and with Zalgo, we are able to tackle some of them. With Zalgo, we are able to provide a starting point for those people who would like to learn coding. No longer will there be the confusion of where to start to study coding. Next, there is also the problem of where those who are studying coding directly dive into various programming languages. Trying different languages such as Java, Python and many others while what we consider as more important is making sure these people are able to have these 3 important skills.

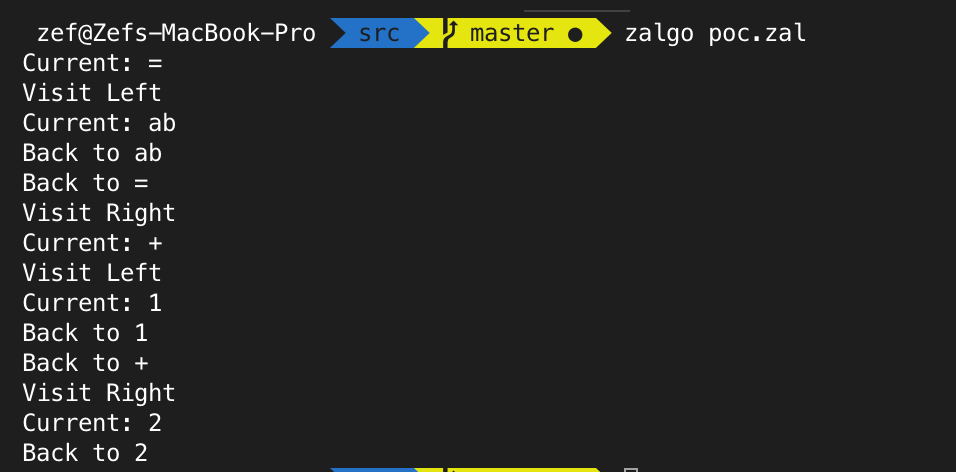
1. Designing a program
2. Logical thinking
3. Coding framework

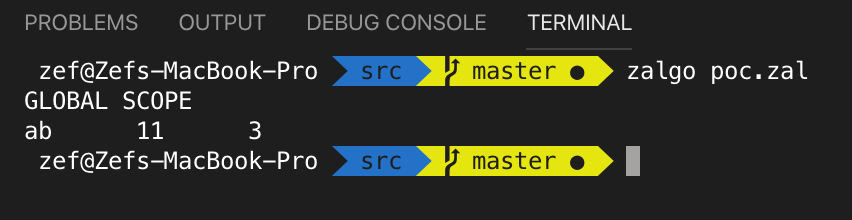
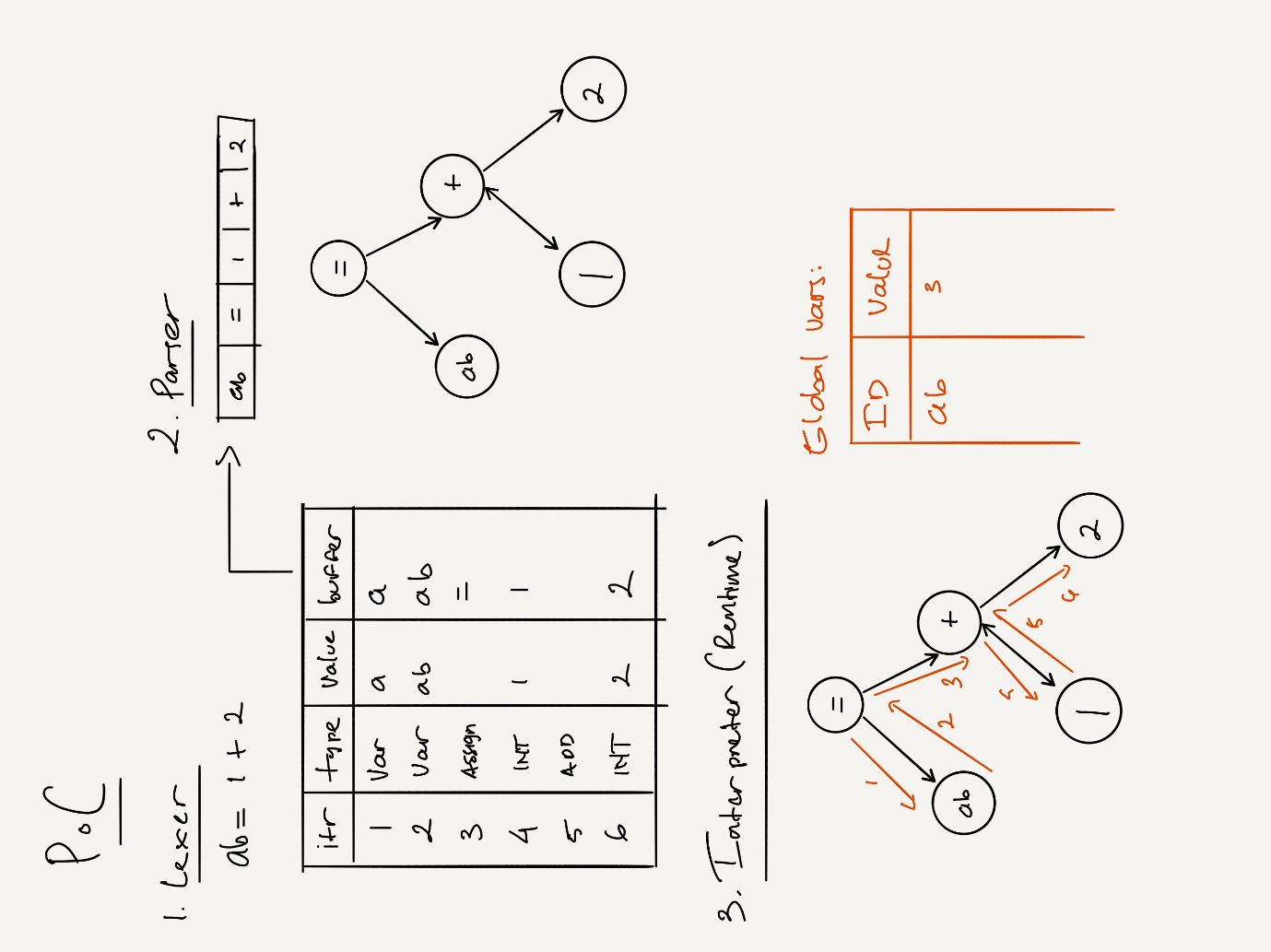
**ii. Design of algorithm**

For Zalgo, we create our own algorithm. So, for the first step we created our own Lexer to execute the lexical analysis – to convert the input code to a string of tokens which will be used for the second algorithm that we have created, the parser which analyses the syntactical structure of the given input. Checks whether the input is in the correct syntax of the programming language. The output of the parser would be in a form of a parse tree. After that, we created another third algorithm, interpreter, that translates the parse tree and executes the lines of instructions accordingly.

**iii. Proof of correctness**

**Lexical Analyzer (Lexer)  
**

**Parser  
**

**Interpreter  
  
  
**

**iv. Complexity Analysis**

To calculate the time complexity of Zalgo, we compared it against C++ and Python. We used a while loop with a simple print instruction. For every loop, 1000 to 5000 with 1000 intervals, we execute 5 times and calculate the average. Then we plot the result against a graph in microseconds.

**V. Evaluation**

**i. Theoretical analysis of the algorithms**

For Zalgo, we used to 3 algorithms.

First one is Lexical analysis or tokenization. Process of converting a sequence of characters into a sequence of tokens. Programs that performs lexical analysis are called lexer or tokenizer. Lexers are most often used for compilers and can be divided into two stages: the scanning, which segments the input string into syntactic units called lexemes and categorizes these into token classes; and the evaluating, which converts lexemes into processed values.

Lexers are generally quite simple, with most of the complexity deferred to the parser or [semantic analysis](https://en.wikipedia.org/wiki/Semantic_analysis_(compilers)) phases. However, lexers can sometimes include some complexity, such as [phrase structure](https://en.wikipedia.org/wiki/Lexical_analysis#Phrase_structure) processing to make input easier and simplify the parser, and may be written partly or fully by hand, either to support more features or for performance. A lexical token or simply token is a [string](https://en.wikipedia.org/wiki/String_(computer_science)) with an assigned and thus identified meaning. It is structured as a pair consisting of a token name and an optional token value. The token name is a category of lexical unit.

Common tokens are:

* identifier: names the programmer chooses;
* keyword: names already in the programming language;
* separator (also known as punctuators): punctuation characters and paired-delimiters;
* operator: symbols that operate on arguments and produce results;
* literal: numeric, logical, textual, reference literals;
* comment: line, block

The second one is parser. Parsing is the process of analyzing a string of symbols, either in [natural language](https://en.wikipedia.org/wiki/Natural_language), [computer languages](https://en.wikipedia.org/wiki/Computer_languages) or [data structures](https://en.wikipedia.org/wiki/Data_structure), conforming to the rules of a [formal grammar](https://en.wikipedia.org/wiki/Formal_grammar).

A parser is a software component that takes input data (frequently text) and builds a [data structure](https://en.wikipedia.org/wiki/Data_structure) – often some kind of [parse tree](https://en.wikipedia.org/wiki/Parse_tree), [abstract syntax tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree) or other hierarchical structure, giving a structural representation of the input while checking for correct syntax. The parsing may be preceded or followed by other steps, or these may be combined into a single step. Parsing is complementary to [templating](https://en.wikipedia.org/wiki/Templating_language), which produces formatted output.

The input to a parser is often text in some [computer language](https://en.wikipedia.org/wiki/Computer_language), but may also be text in a natural language or less structured textual data, in which case generally only certain parts of the text are extracted, rather than a parse tree being constructed. An important class of simple parsing is done using [regular expressions](https://en.wikipedia.org/wiki/Regular_expression), in which a group of regular expressions defines a [regular language](https://en.wikipedia.org/wiki/Regular_language) and a regular expression engine automatically generating a parser for that language, allowing pattern matching and extraction of text. In other contexts, regular expressions are instead used prior to parsing, as the lexing step whose output is then used by the parser.

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

* [Top-down parsing](https://en.wikipedia.org/wiki/Top-down_parsing) - Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for [parse trees](https://en.wikipedia.org/wiki/Parse_tree) using a top-down expansion of the given [formal grammar](https://en.wikipedia.org/wiki/Formal_grammar) rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate [ambiguity](https://en.wikipedia.org/wiki/Ambiguity) by expanding all alternative right-hand-sides of grammar rules. This is known as the primordial soup approach. Very similar to sentence diagramming, primordial soup breaks down the constituencies of sentences.
* [Bottom-up parsing](https://en.wikipedia.org/wiki/Bottom-up_parsing) - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. [LR parsers](https://en.wikipedia.org/wiki/LR_parser) are examples of bottom-up parsers. Another term used for this type of parser is [Shift-Reduce](https://en.wikipedia.org/wiki/Shift-reduce_parser) parsing.

The third algorithm is the interpreter.

Interpreter theory says that decision-making, judgment, perception, and virtually everything else that takes place in the brain is unconscious, and that what we understand as conscious thought is a distinct process that after the fact generates explanations for our actions and our experiences. Similar claims sometimes presented are that consciousness is an illusion or that consciousness is out of the loop in decision-making.

**ii. Implementation details**

The first algorithm, lexer, reads the input character by character from the start to the end. Reading every character, when it detects a keyword – a symbol or a syntax, it will be inserted into a queue as a token. If it detects a set of characters that do not match with a keyword, the lexer can predict what type of keyword it would be, but it could also act as a stop point meaning the previous character that matches a keyword or variable will be put into the queue as a token.

The second algorithm, parser, accepts strings of tokens generated by the lexer inside a queue. Parser checks every token then forms a parse tree from the tokens that later will be used by the interpreter to be executed.

The third algorithm, interpreter, executes the code from the parsing tree.

**iii. Data sets**

For the data sets used in the input, we can use anything from simple mathematical expressions to different loops in the code.

For example:

a = 1 + 1 or a=1+1  
print(“eris”\*100)  
a = input "Enter name: "  
print "Hello " + a + "!"  
a = 3  
b = a \* 2  
if a equals 1 then  
 print "a " \* 10  
else if a equals 2 then  
 print "b " \* 10  
else  
 print "c" \* 10  
end if  
i = 0  
max = input "Enter number: "  
while i < max then  
 print "Eris " + i  
 i = i + 1  
end while

# print 10 ^ 2  
print "DONE!”

**iv. Results**

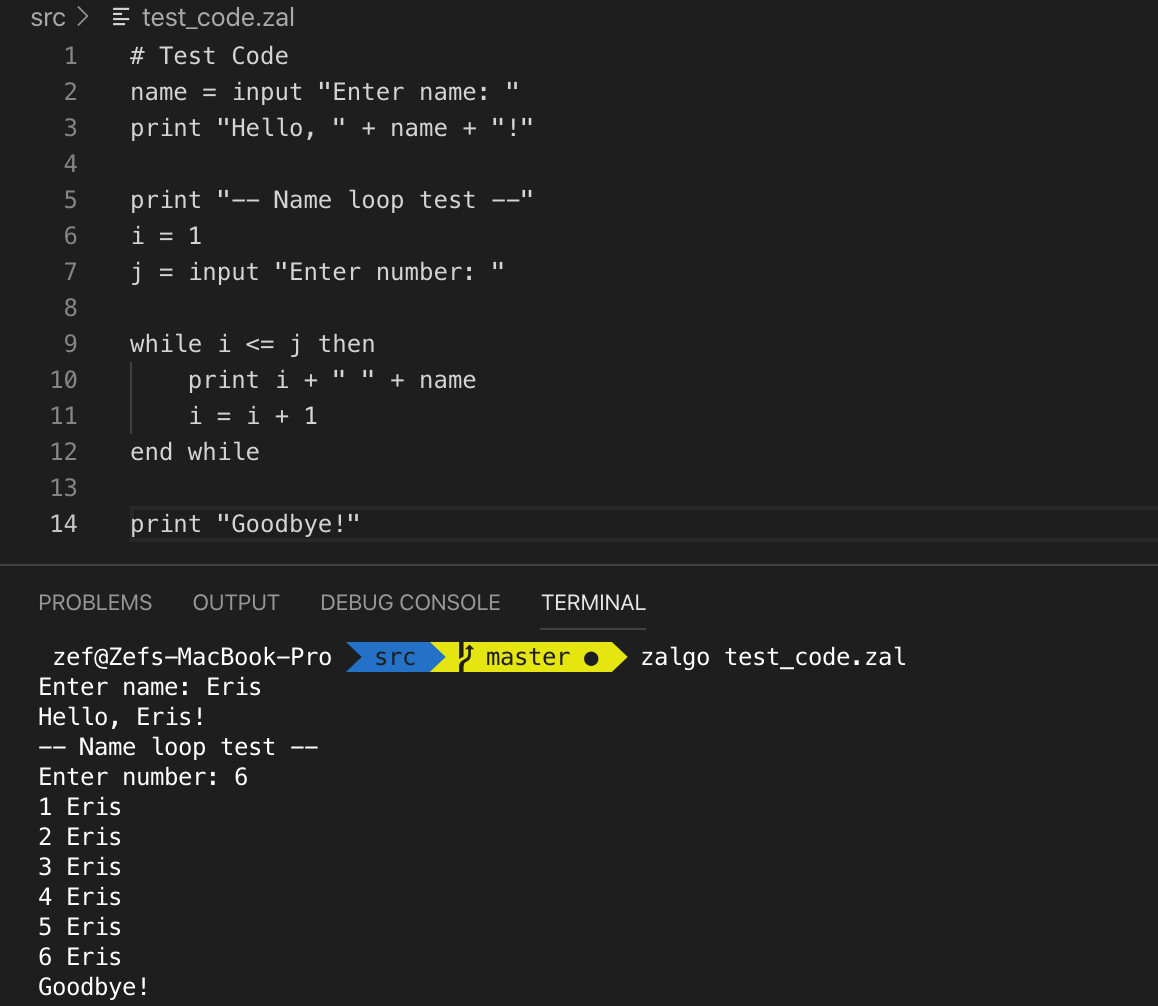
We have created an interpreter that is able to run a set of instructions with a certain set of grammar rules that is already pre-determined.

**VI. Conclusion and recommendation**

To create an interpreter, at least 3 different algorithms is required. We need to be able to read word by word, syntax by syntax and converting it into a token for the first algorithm, lexical analysis. Next, we have to be able to create a grammar rule for the interpreter where the second algorithm, parser, will check whether the tokens are the correct syntaxes. After that, the interpreter has to be able red execute the program from the parsing tree, the output result of the parser.

We would recommend designing and plan everything from the beginning to the end so that when creating the algorithm, the next steps are clear. We have to plan every grammar rule and all the syntaxes.

**VII. Program manual, how to execute (with screenshots)**



**VIII. Link to the application demo video (with max. length of 2 minutes)**

<https://drive.google.com/drive/folders/1DBSmw-zfzho4tOhHqHb8XetYenCmrcEV?usp=sharing>

**IX. Link to GIT Website**

<https://github.com/zefryuuko/zalgo>

**X. Sources**

<https://en.wikipedia.org/wiki/Lexical_analysis>  
<https://en.wikipedia.org/wiki/Parsing#Parser>  
<http://humancond.org/analysis/mind/interpreter_theory>