# CS251: Proposal

3 Idiots
160050005
160050008
160050028

October 29, 2017

# Contents

# EYE, The Interpreter that Visualises

## 1 ABSTRACT

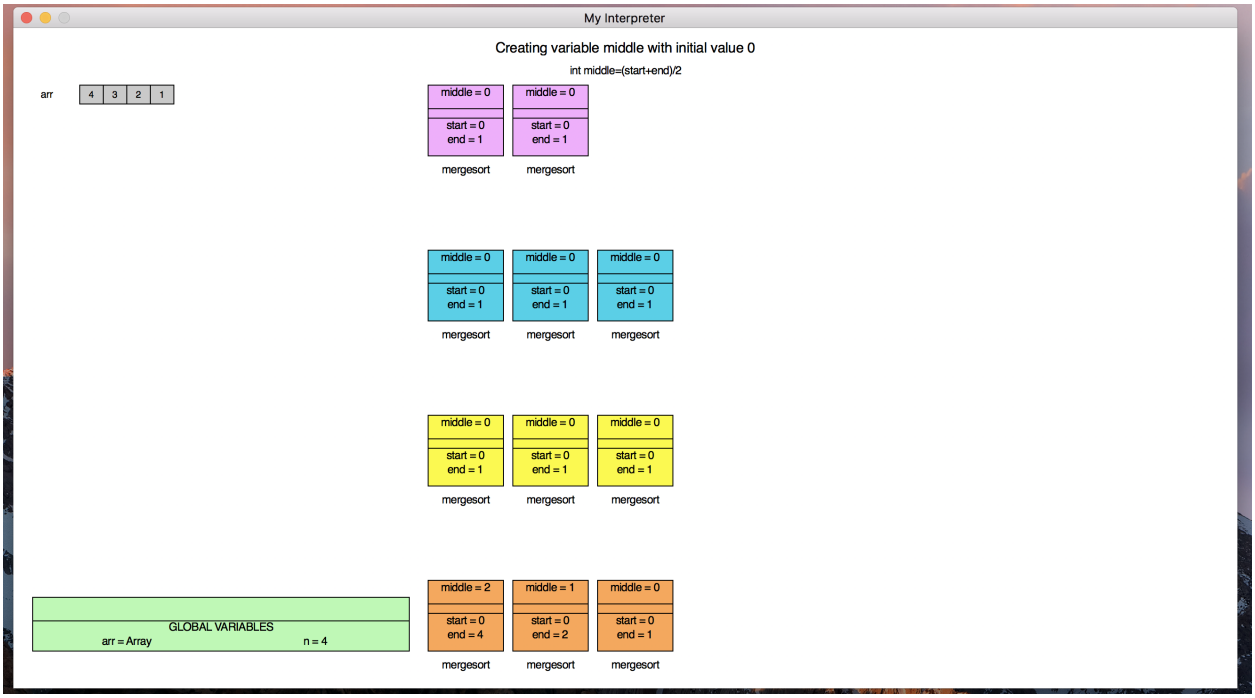
- We aim to create an **Interpreter** for our own language (syntactically similar to C++) and provide a visual representation of the *"behind-the-scenes"* **memory allocation** and other processes executed by the interpreter
- We would also develop graphical representations of the **execution stack** of the program as it is being executed as well as the **environment** in which functions are called and executed
- There would be special structures to visually represent common **data structures** like queues, trees, linked lists, stacks and others on a canvas, which would be updated every time a method is called
- Essentially, this would be a *comprehensive tool* to visualise program execution in **real time**

## 2 INTRODUCTION AND MOTIVATION

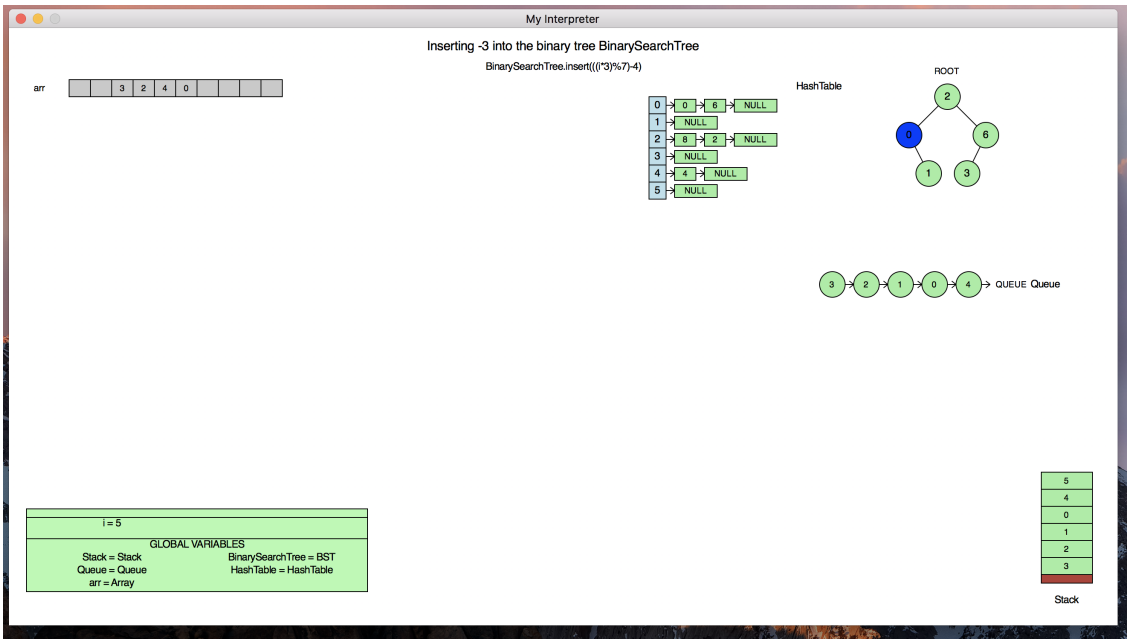
- The difficulties faced by us in our Algorithms class inspired us to create a tool that would give a very *intuitive* and *in-depth* visualisation and **feel of the algorithms** we were learning about.
- We realised that being able to see an algorithm execute (for example, seeing the state of an array after every iteration of the **insertion sort algorithm**, or the probe sequence in a binary search tree) would go a long way in making us *proficient* in developing our own algorithms.
- This can also prove to be a good tool to teach **introductory courses** on programming to those without much prior experience (for example, in the *CS101* course) as it would really help develop a deep understanding of what it means to code and how the code you write is actually being executed by the compiler. Having knowledge of the functioning of a compiler is of great use in any field related to computer science.
- Another motivation behind this project was to develop a **debugging** tool that would not be as non-descriptive as a segmentation fault and not need as much effort to debug as raw C++ source code. We can **trace the execution** of the code until we come across the error to find memory leaks, to catch logical shortcomings or corner cases.

## 3 The Final Product

The final product we developed contains an interpreter which takes a file (with eye-styled code) as input and executes it step by step, while simultaneously ensuring changes in the graphics and visualisations. Comprehensive instructions regarding the software requirements and running instructions can be found in the user manual. Here is a screenshot of the panel that is displayed while the program is being executed:




In the screenshot one can see that at the top we have a line which tells us what task the interpreter is carrying out currently. Below that we have a code snippet displayed, which shows the actual code that the user has written which is being currently executed. On the bottom left we have the execution stack which represents the differnent scopes as well. We see that at the bottom we have the global variables and as we move up we have different rectangle representing the different activation frames, with the second lowest one denoting that of the main program.
Everything else represents the heap memory, that is the various arrays, data structures, etc that are assigned memory in the heap. We can see that at present, we have arrays both of which are declared in the heap, namely "arr". At the bottom, to the right of the execution stack we have the function stacks, which denote the activation frame of all the functions that are active.
So, the screenshot denotes the output that we get when we run a program that implements mergesort on an array of length 4 with values 4,3, 2 and 1. However, this code has a small bug and this causes a very un-descriptive run-time error ("segmentation fault") on executing in C++. However, in this screenshot, we see that the frame with the same "start" and "end" values has been made numerous times and this means that an infinite number of function frames are being created. This is because the base case has been written incorrectly and this error is very easy to detect using Eye, but would be difficult to debug in C++.




In this screenshot, we can see the various data structures that we have implemented to the right. They are all defined in the heap and we have developed representations for binary search trees, hash tables, linked lists and stacks.

# 4 The Code

We have documented the code well and we have divided it into folders as well which represent one module each. We have the parser directory that contains the two files - parsingClasses - which defines the classes we use to store each of the programming constructs and parsingRules - which defines all the rules for parsing. The directory called lexer contains the rules for lexing the user code. The dataStructures directory contains the graphics library we used and our implementations of all the data structures, execution stack, etc.

# 5 LITERATURE SURVEY AND PRIOR ARTWORK

- Extensive research has been done on the field of interpreters previously. There are many interpreters for the language C++ like Cint, Cling etc. Though **Cint and Cling** have most of the functionality related to C++, none of them implement *all* of them.
- Attempts have been made to create graphical visualizations of programs. People have always been interested in this field and we found a research paper[1] dating right back to 1983.

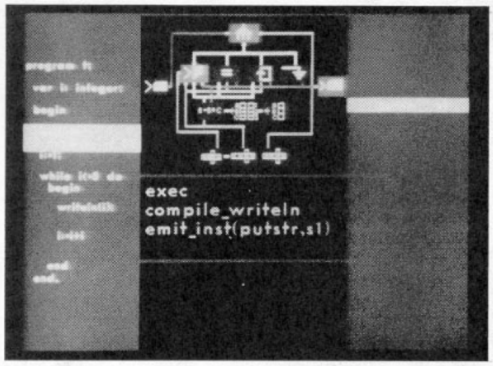



Figure 1. Overview of a running compilation. (Animated mock-up.) Source code is on left, object code appears on right as it is output. Highlights on compiler structure diagram indicate active routines, which are named in the stack trace below the structure diagram.

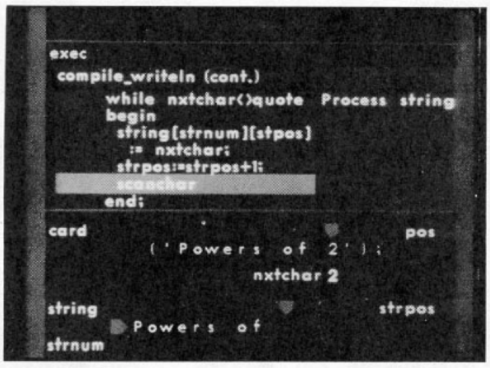



Figure 2. Detailed view of a running compilation. (Animated mock-up.) As highlights move through code, graphical index pointers advance across arrays and the box for the NXTCHAR variable moves along under the CARD array to the appropriate position. The break in the left hand part of the CARD array is used to indicate the existence of initial entries in the array that do not appear in the current display.

- As for the **graphical representation** of code,there exist some sites (*https://visualgo.net/en/sorting?slide=1*, *https://www.cs.usfca.edu/ galles/visualization/Algorithms.html*) where you can visualize basic data structures for already decided algorithms.
  There are not many platforms that allow visualisation of user-input code. However, we found this link on the internet: *http://www.pythontutor.com/cpp.html#mode=edit* which is a visualisation environment which does similar things but for python. The counterpart for C++ is still in the experimental phase.

# 6 PROBLEM STATEMENT

Develop an interpreter for a small rule-based language of your own using Python3 and its libraries. The interpreter must depict, in real time, the execution environment, data structures, variables and memory allocation to give the user a comprehensive overview of the interpreter working and code execution.

# 7 SOFTWARE AND HARDWARE REQUIREMENTS

## 7.1 Python3

The entire code will be written in python3 and we will be using its libraries extensively, some of which are as follow:

- **rply**: we will be utilizing its following classes:
  - **LexerGenerator** – for lexical analysis of the user's code
  - **BaseBox** – for defining our own rules for the abstract syntax trees (These are trees that define the grammar of any language) which would be utilized for interpreting
  - **ParserGenerator** – to do the most important and involved task... parsing!
- **graphics.py**: is an open source library. We will be using it for the following
  - To build basic **geometrical shapes** like rectangles, circles, lines, and much more, which would in turn be used to build structures to visualise common data structures.

– To add **animation** to the objects on the canvas to animate execution of functions and make it easier to understand what is happening.

### 7.2 Doxygen and Latex

We will utilize these tool for two purposes:

- to maintain documentation of our code as we expect it to become extremely large
- to develop a reference manual for Eye, enlisting all its features, standard libraries and member functions.

### 7.3 HTML5/CSS

We plan to build a simple web-based user interface for our project.

### 7.4 Git and GitHub

We will use this extensively to maintain the code base and allow different team members to simultaneously work on different modules conveniently.

## 8 IMPLEMENTATION

The implementation of our project can be broken down into the following modules:

- Lexer
- Parser
- Interpreter
- Graphics

$$code \longrightarrow \boxed{\text{Lexer}} \longrightarrow \boxed{\text{Parser}} \longrightarrow \boxed{\text{Interpreter}} \longrightarrow output$$

#### 8.0.1 Lexer

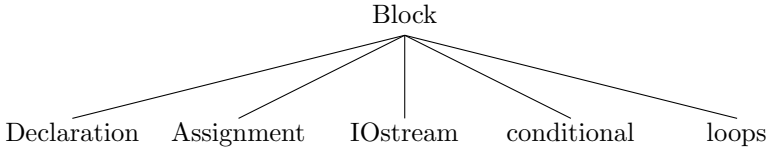
- We used the **rply** library to tokenise each element of the user's code.
- The **lexergenerator** class' **add() method** allows us define rules to allot tokens to different parts of the code.
- Using the **ignore() method**,we excluded irrelevant elements like whitspaces.
- The **build() method** builds all the rules and calling the lexer.lex() function with the user code as input, we get a **lexerstream** object.
- In Figure 1, we define rules to tokenise the input string, like assigning the token 'assignment' to '='. Whitespaces have been ignored.
  The **next() method** of the lexerstream object created on tokenising the string '5 =5', gives us the tokens in order.


Figure 1: Sample Lexer

#### 8.0.2 Parser

- We defined classes for all major programming constructs and the aim of parsing is to convert the tokenised user code into instances of these classes. Examples of a few of these classes are assignment, for loops, while loops, cin and cout, functions, etc.
- Each class has data members to store the necessary information and functions that are specific to the programming construct. This enabled us to execute the code eventually. The parser will detect keywords and create instances of these classes.
- Consider the example of a 'for' loop. It has the following four data members-
  - Initialization- block object
  - Condition- expression object
  - Updation- block object
  - Actual code to be executed repeatedly- block object

  Now, an expression is basically code that can be evaluated to produce one single value. A block object is a list of one or more statements and a statement itself can be defined as:




- Every datatype in our language is stored in a corresponding datatype in python. For example, a string in C++ is stored in a string in python and complex data structures like stacks, queues and trees will be stored in classes defined by us.
- As another example, the class used to represent the declaration statement has two data members:

  - Type of variable being referred
  - Assignment type object which itself will contain
    * Name of the variable stored as a string
    * Value to be assigned stored as an expression object

- Similarly for all other constructs, this table depicts the data members and what each class represents.

| **Parser Class** | **Its Data Members** | **What it Represents** |
|---|---|---|
| Identifier | letter(letter—digit)* | Name of a variable |
| Variable | identifier—identifier[expression] | Variable or array element |
| Term | raw data—variables—(expression) | A value |
| Expression | (unary operator)* term (binary operator term)* | Anything that can be evaluated |
| Statement | declaration—assignment—cout/cin—loops | Constructs of different kinds |
| Block | statement(statement)* | One or more statements |
| Assignment | variable and expression | Statements with an "=" sign |
| Declaration | variable type and assignment type | Variable declarations |
| Cout/cin | List of expressions/variables | Input and output objects |
| Conditionals | Expressions for conditions and blocks to execute | If – else if – else constructs |
| For loop | Intialisation, updation, condtion and block | For loop constructs |
| While loop | Condition and block | While loop constructs |
| Functions | Name, argument list and block | Function Declarations |
| Function Calls | Function name and list of arguments passed | Actual execution of functions |

- In Figure 2, we see the rules for parsing all sorts of expressions. Once the expression has been parsed and stored in the aforementioned classes, evaluating them only required us to call the corresponding python operators.

#### 8.0.3 Interpreter

- Before we can actually begin parsing, we must store all the variables and functions that the user has defined in a 'symbol table'. The symbol table for functions is simply a dictionary, where the key is the function name and the value will be an instance of the class used to store functions.

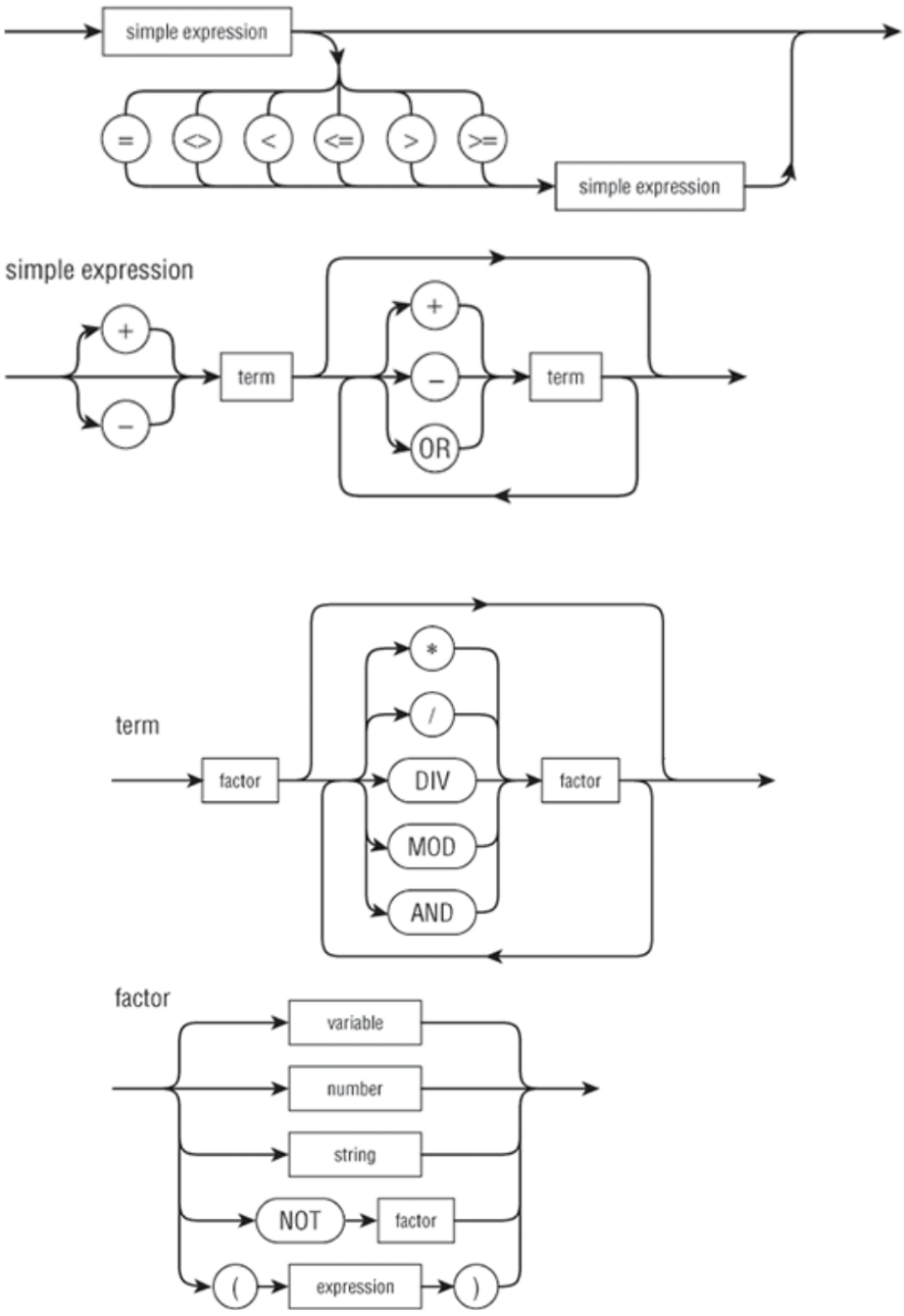



Figure 2: Rules for parsing and evaluating all sorts of expression. This is from a reference book [2] we found on interpreters and compilers

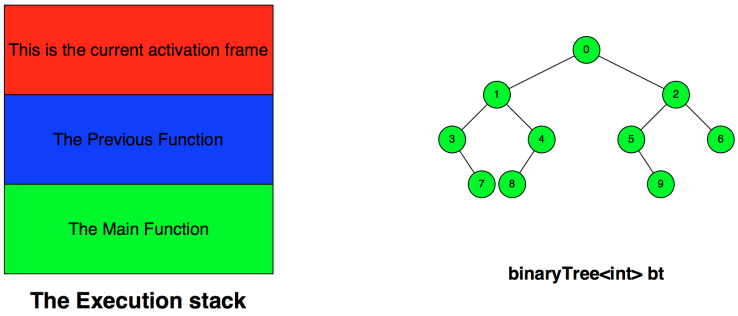
- Now, to store variables, we used a list of dictionaries where the key is the name of the variable and the value is a python object of the same kind as that in the user's program. For example, if the user defines a string called myString with value "SSL", the value associated with the key 'myString' will be a python object of type string with the value 'SSL'. To maintain scope of variables, the last entry in the list contains all variables defined only in the current scope, while the first element of the list

contains all the global variables. To find the value of a variable, we start searching from the last entry in the list and work our way backwards.

- When we encounter a '}', the last entry of the list is deleted.
- Whenever a function is called, a new list is created which has the same global variables and the parameters of the function are at position 1.
- We are now set to execute the code using appropriate rules for executing for loops, if loops, etc.

## 8.1 Graphics

For every common data structure, the standard member functions have been modified to implement an intuitive visualisation of the data stored. However, to prevent over-crowding and ensure ease-of-understanding, we limited the size and number of such data structures.
Also, we represented the execution stack and environment in which each line is executed and evaluated. We represent a particular scope as a rectangular block and display in it the names and values of every variables at each stage. Figure 8.1 shows a crude visualisation model.




# 9 FEASIBILITY

This platform has great feasibility with respect to real-world scenarios. Eye is a great tool for understanding the working of basic algorithms and for debugging simple programs. This would also be a phenomenal teaching tool for courses like CS213 and CS101 to demonstrate common problems and algorithms like the railroad problem, LPT algorithm in machine scheduling and sorting algorithms. It will give students insight into the basic working principles of an interpreter and help them develop a deep and thorough understanding of algorithms.
However, we realise that it is impractical to implement all the varied features of C++ and hence, we will not include concepts like pointers, inheritance and polymorphism.
We strongly believe that this can serve as a template to develop more functionality to make debugging possible for complex programs. This will help simplify the task of all programmers significantly and save a lot of time, energy and frustration.

**Based on the inputs given by Prof. Kavi Arya after our presentation in class, we are adding the following sections**

# 10 Proposed Deliverables

Our deliverables would include the following:

1. Build lexer, parser and interpreter modules to handle the following programming constructs:
   (a) Mathematical expressions including the +, -, *, / and % operators
   (b) Boolean expressions including the or, and and not operators
   (c) Assignment statements
   (d) Arrays
   (e) Declaration statements
   (f) If, If-else and If-else if-else blocks
   (g) For loops
   (h) While loops
   (i) Functions
2. Develop implementations for the following non-primitive data structures which would include visualisations for their common member functions:
   (a) Linked Lists-doubly and singly linked
   (b) Stacks
   (c) Queues
   (d) Hash Tables
   (e) Trees
   (f) Heaps
3. Develop a comprehensive, easy-to-understand interface to depict the execution environment and the scope of different variables. The state of all variables and classes will be maintained at all times.
4. Eye will be able to visualise simple algorithms and how the data evolves on running the algorithms. For example, we will represent the state of the array after each iteration of a sorting algorithm.

# 11 What we Actually Accomplished

We are proud to be able to say that we have been able to implement the entire project and actually develop all that was mentioned in the deliverables. We have built the entire lexer and parser modules as well as developed visualisations of all common data structures. We have not only been able to meet the goal that Professor Kavi Arya had set for us (of showing a simple bubble sort or insertion sort program), but have been able to accomplish much more (like complex recursive sort methods eg. mergesort).
The deliverables we have delivered are as follow:

1. Here is a list of the programming constructs that Eye can lex, parse and execute:
   (a) Mathematical expressions including the +, -, *, / and % operators
   (b) Boolean expressions including the or, and and not operators
   (c) Assignment statements
   (d) Arrays
   (e) Declaration statements

(f) If, If-else and If-else if-else blocks

(g) For loops

(h) While loops

(i) Functions

2. We have also implemented the following abstract data types with their common methods (along with intuitive and comprehensive visualisations) to understand their functions:

   (a) Linked Lists-doubly and singly linked

   (b) Stacks

   (c) Queues

   (d) Hash Tables

   (e) Trees

   (f) Heaps

3. We have built a console using the graphics library which depicts the execution stack, the variables that are currently present in the execution environment along with their scopes as well as visualisations for common data structures as has been described in the third section.

4. We have also been able to depict comprehensively and in a very intuitive fashion how the data evolves when an algorithm is executed. It is very easy to understand the various sorting routines using Eye.

# 12 Proposed Timeline

We will build the project in three iterations and in each iteration, we will include some more constructs and complete all the related modules. This means that after a given iteration, the program will be able to tokenise, parse, execute and visualise all statements belonging to a particular programming construct.
We will have 3 iterations, each lasting one week. We will aim to including the following constructs in each construct:

1. Mathematical and Boolean expressions, assignments and declarations

2. For loops, if loops, while loops

3. Non-primitive data structures

We have kept a week at the end to iron out bugs, optimise and document the code and complete parts that might not have been completed on schedule.

# 13 The Timeline we Actually Followed

We pretty much followed the timeline that we had proposed based on the suggestions and feedback we received in class after our proposal presentation. Here is the final timeline:

- Week 1: Mathematical and Boolean expressions, assignments and declarations
- Week 2: For loops, if loops (conditionals) and while loops
- Week 3: Non-primitive data structures (stacks, trees, hash tables, etc.)
- Week 4: Functions, documenting the code, removing bugs in the code and developing the user manual

# References

[1] Richard T. Carling Christopher F. Herot David Kramlich, Gretchen P. Brown. Program visualization: Graphics support for software development. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1585640&tag=1.

[2] Ronald Mak. *Writing Compilers and Interpreters*. Wiley Publishing, Inc., 2009.