Milestone1

Abir Rajbongshi 210273
Aditya Bangar 210069
Pratham Sahu 210755

February 2024

1 Introduction and Running the Code

In this milestone, we have implemented a parser-lexer pair for the Python language. We use the flex and bison tools to implement the lexer and parser respectively. We have also implemented the Abstract Syntax Tree (AST) for the given input python file by adding the necessary code in the bison specification file.

To build the binary, we use the makefile. The makefile contains the necessary commands to build the binary from the lexer and parser files.

Run the following command to build the binary:

```
make
```

To run the binary, use the following command along with command line arguments:

```
./pycomp
```

The following commandline arguments are supported:

```
-- help : to display the help message
-- verbose : to display the verbose output, on the terminal
-- input : to specify the input file
-- output : to specify the output dot file
-- graph : to specify the output type for the AST, it will be of the same
         name as the output file. It can be either png or pdf.
```

To clean the binary, use the following command:

```
make clean
```

2 Features

The following list of features are supported in our implementation:

- Regular indentation used in python can be handled. This was implemented using multiple states in the lexer.
- Our implementation handles keywords, comments, identifiers, types and operators.
- Data types supported are String, Integer, Float, Imaginary, Boolean
- All basic keywords mentioned in the Milestone1 pdf.
- Basic Arithmetic, Relational, Logical, Bitwise and Assignment operators have been implemented.
- Basic error handling has been implemented (expounded on in the next section).
- It can handle implicit and explicit line joining.

3 Error Handling

We have imitated the actual python compiler error handling for our implementation

The different types of error handling we are carrying out in Milestone1 are

- For indentation error we print an indent error message along with the line number
- Missing opening and closing brackets are handled. The line number is printed along with the error message.
- For invalid tokens, we print an error message along with the line number.

4 Construction of the AST

We create the AST by adding program segments to the bison specification file. It is an S-attributed SDT.

Each node is a struct with the following fields:

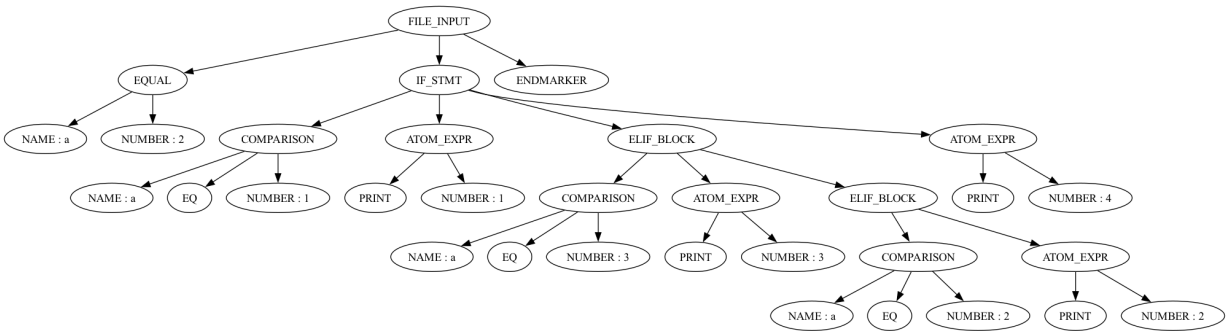
```
struct node_ast{
  string name;
  string lexval;
  vector<node_ast*> children;
};
```

The name field contains the name of the node, lexval contains the value of the token and children is a vector of pointers to the children of the node. A few examples of how the AST is portrayed are:

Example 1

In the following example, we show the AST for an if-elif-else block of statements. Inside the IF_STMT token, we have a comparison statement, the then block, the elif block, and the else block. The elif block is further presented as a tree.

AST

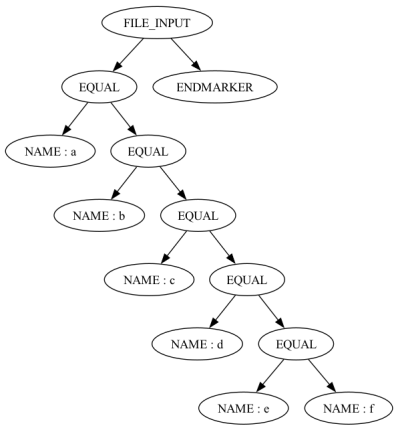


Program Segment:

```
a=2
if a == 1:
    print(1)
elif a == 2 :
    print(2)
elif a == 3:
    print(3)
else:
    print(4)
```

Example 2 In the following example, we show the Right Associative nature of assignment operator in the AST.

AST:

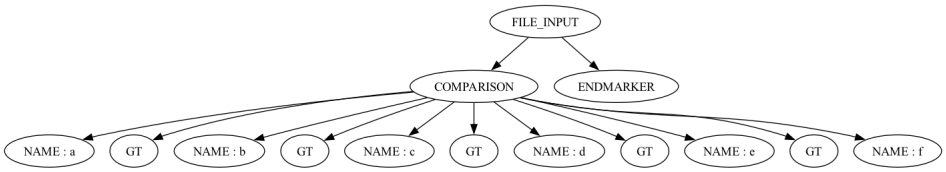


Program Segment:

```
a = b = c = d = e = f
```

Example 3 In Python, comparison operators do not follow any precedence. For the entire statement to be true, all the comparisons must be true, hence in the AST, all are at the same level.

AST:



Program Segment:

```
a > b > c > d > e > f
```

5 References and Testcases

- We have referred to the python 3.7 grammer specification to write the bison specification file.

The testcases we have used are

- test1.py - Dijkstra algorithm.
- test2.py - K Means Algorithm.
- test3.py - Implementation of university course enrollment system using classes.

- test4.py - Bellman Ford and number of paths.
- test5.py - Implementation of customer data management system using classes.
- test6.py - Rabin Karp and KMP pattern matching algorithms.