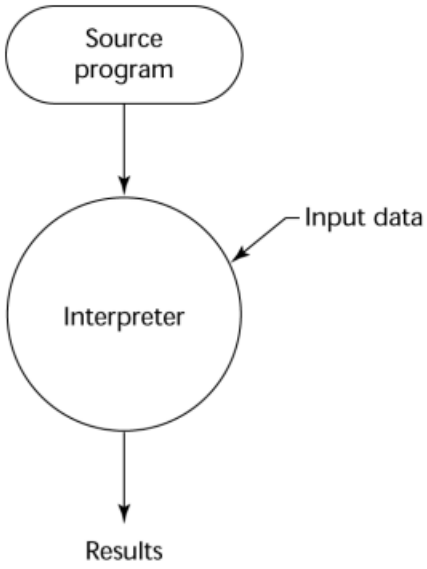
Interpreter Assignment Report
Team- 10

Introduction

Interpreter is a program which directly executes the given instructions in high level language, without a need to convert it to machine code.



It has its own advantages and disadvantages. Unlike compilers, interpreter takes a single line of code at a time, so it becomes easy to debug if we encounter errors. Also, interpreters have cross- platform functionality, meaning the source code can run in any system which has the interpreter code. But interpreters are slow and less secure compared to compilers.

Language Chosen

The language we chose for this project is **JULIA**. Julia is a high level, dynamically typed and a high-performance language. Julia is a functional language. It is created to give the speed of C/C++, while remaining as easy as Python to use.

Constructs Interpreted

- Assignment Statements
- Compound Statements
- Print Statements
- Conditional Statements
- Boolean Statements

Assumptions and Constraints

- The datatypes we considered in our interpreter- int, boolean and float.
- The print statement in our interpreter doesn't have format specifiers.
- We used semicolon (;) for "End of Line" functionality, which is not actually available in Julia.
- The boolean statements only support two operands.

Explanation

The code has three parts- Lexer, Parser and Interpreter.

Lexer

The lexer breaks down the code into stream of tokens and takes each token and categorizes it based on its type and assigns type and value for each token. This helps in performing operations of a specific type which are specific to the type of token. In Julia there are no predefined datatypes for variables and hence it takes the datatype of the assigned value.

For example: a=5, 5 is an integer hence a is an identifier with type integer and of value 5.

Every token has a type and a value.

```
class Token(object):
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __str__(self):
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()
```

Language has some reserved keywords which can’t be kept as variable names.

```
RESERVED_KEYWORDS = {
    'begin': Token('begin', 'begin'),
    'end': Token('end', 'end'),
    'if': Token('IF', 'if'),
    'elseif': Token('ELSEIF', 'elseif'),
    'else': Token('ELSE', 'else'),
    'println': Token('println', 'println'),
    'True': Token('TRUE', True),
    'False': Token('FALSE', False)
}
```

Parser

The functionality of parser is to perform syntax analysis. The tokens generated in lexer are taken as input in parser. The stream of tokens is now traversed sequentially and are linked based on their order of execution, i.e., a tree is generated with nodes having data of execution and using post-order traversal this data is implemented with the operations and the result is computed in the interpreter. A class is defined for node creation in each category.

BNF and Parse Tree of constructs used

- **Assignment Statements**-

  BNF-

  <assign_stmt> → <var> EQUALS <expr>

  <expr> → <term> | <expr> PLUS <term> | <expr> MINUS <term>

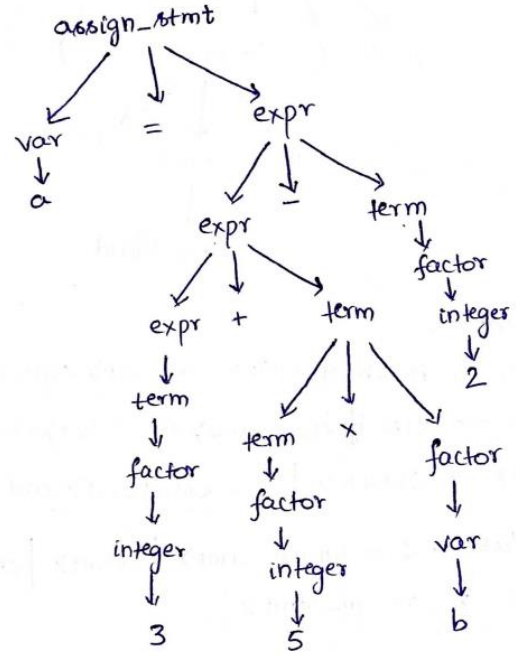
  <term> → <factor> | <term> MUL <factor> | <term> DIV <factor>

  <factor> → INTEGER | FLOAT | <var>

  <var> → a | b | c | d | e | .....

  Parse Tree-

  Example → a = 3 + 5 x b - 2



- **Print Statement**

  BNF-

  <print_stmt> → print LPAREN <to_print> RPAREN

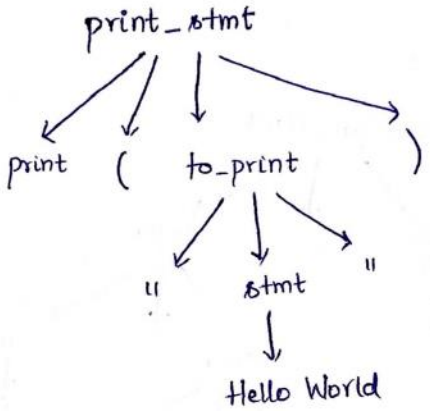
  <to_print> → <factor> | QUOTES stmt QUOTES

  <factor> → integer | float | <var>

  <var> → a | b | c | d | e - - - - - - - - -

Parse tree-

Example → print ("Hello World")



- **Conditional Statements-**

<conditional_stmt> → if <bool_stmt> <stmt_list> <block>

<block> → else if <bool_stmt> <stmt_list> <block2> | end | NULL

<block2> → end | <block> | else <stmt_list> end

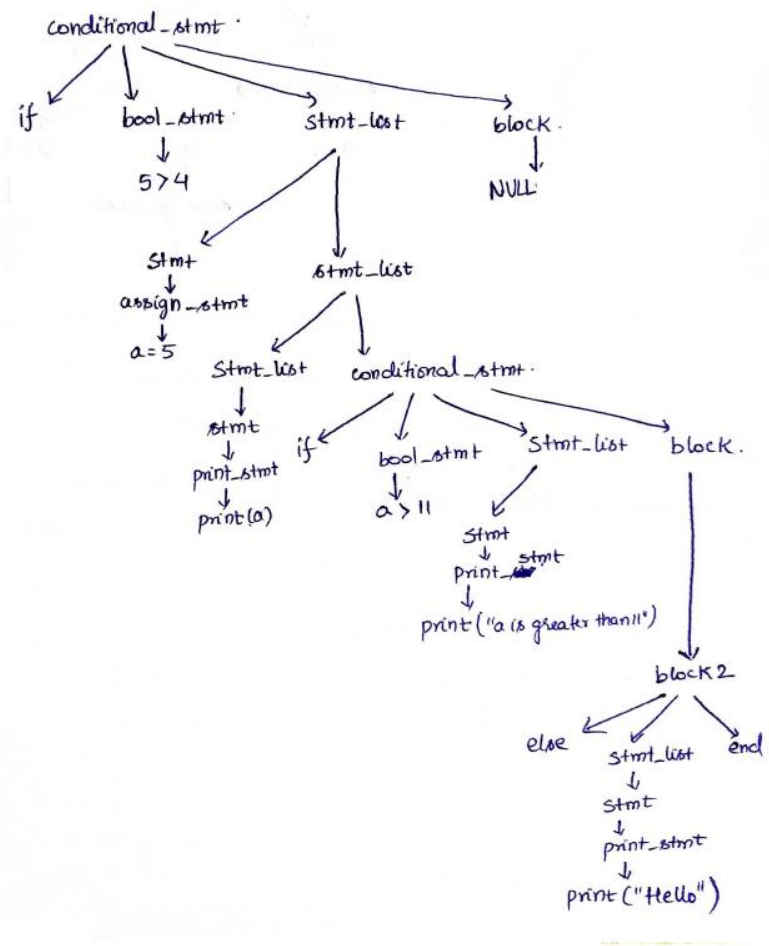
<stmt_list> → <stmt> | <stmt> <stmt_list> | <stmt_list> <conditional_stmt> | <conditional_stmt>

<stmt> → <assign_stmt> | <print_stmt>

Parse Tree-

Example :→

```
if(5>4)
  a=5
  print(a)
  if(a>11)
    print("a is greater than 11")
  end
else print("Hello")
end
```



Using the above BNF, code of parser is written. This includes creation of node and generating parse tree.

Interpreter

Interpreter takes the parse tree generated by the parser as input. The generated parse tree is traversed and the result is computed. This is done by creating a generic function of format “visit_nodename” and the final result is computed.

If the syntax of the program is wrong anywhere, parse is not generated completely, hence the system shows error

Challenges and Learnings from this project-

Learnings-

To build the interpreter was the main aim of this project, hence for the aforesaid purpose we have had a close understanding of the steps of compilation and interpretation. This has cleared our doubts regarding BNF grammar.

The interpreter has shown how to convert a given piece of grammar into an equivalent code and also helped is to enhance our OOPs concepts, especially in python.

Challenges-

New line token for demarcation was not found, we had to use semicolon for that purpose, even though Julia doesn't have semicolon for demarcation.

Learning a new language (Julia) and its basic constructs was challenging.

- Done by Team 10

Team Members and their contributions: -

CS21B040- Pagolu Pavan –
Parser, Interpreter and Test case File

CS21B054- Vinnakota Charvy –
Parser and Interpreter

CS21B042- Reddy Sai Krishna –
Parser and Interpreter

CS21B052- Vakalapudi Nikhil Chowdary –
Lexer, BNF, Report

CS21B029- Kotapati Bhargava Chaitanya –
Lexer, BNF

CS21B024- Joolakanti Pranay Bhasker Reddy –
Lexer, Report