Workings of a Programing Language Interpreter

Narinder Singh

San Jose State University

Abstract— This report deals with the theory and implementation of Interpreter. The implementation of Interpreter requires the creation of Tokens by Lexer, parsing of those tokens to Abstract Syntax Tree(AST) by a parser according to the defined formal grammar rules. The AST is then visited by the interpreter to perform the required operations on different types of nodes.

Keywords— lexer, parser, interpreter, grammar, AST, compiler

I. INTRODUCTION

A. Background: Compilation vs. interpretation

Computers do not understand human languages, they require a middle man to convert English or any other characters to 0's and 1's. Compiler is such a middleman that takes human readable instructions and converts them to binary code which the computer processor can process and produce the desired results. As computer technology improves the programming languages get better as well. In a way improvements in processing power and programming languages go hand in hand. During the 1980's and 1990's when processing power was limited the programmers had to write programs in some kind of Assembly Languages based on the instruction set of the intended processor, which was not only less productive but also challenging to maintain

and debug. The job of a compiler is to translate a program written in higher language to some other lower level code and execute it. However, on the other hand an Interpreter relies on the mechanism of executing the code directly with the help of language it is written in without having to translate it to another type of code.
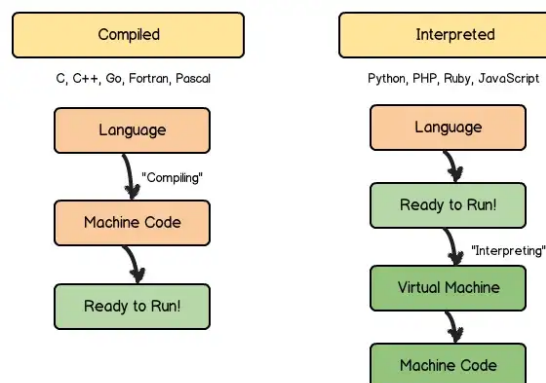

Fig. 1. Compiled vs Interpreted languages. (image source: quora.com)

B. Languages in the Real World

With more processing power programming languages have evolved to be written in human friendly code and compiled or interpreted to generate the results. Here is the list of different types of programming languages that are either compiled or interpreted or both that came into existence in the last few decades.

• C is usually compiled to machine code by GCC.

• Java is usually compiled to JVM bytecode by Javac, and this bytecode is usually

interpreted, although parts of it can be compiled to
machine code by JIT (just in time compilation).
• JavaScript is interpreted in web browsers.
• Unix shell scripts are interpreted by the shell.
• Haskell programs are either compiled to machine code using GHC,
or to bytecode interpreted in Hugs or GHCI.
Notice: Java is not an "interpreted language" - but JVM is!

II. PROJECT SPECIFICATION

This is a implementation of
a simple interpreter in Python that executes the mathematical expression for four basic operations: addition, subtraction, multiplication and division. The expression for the operation will be provided by the user.
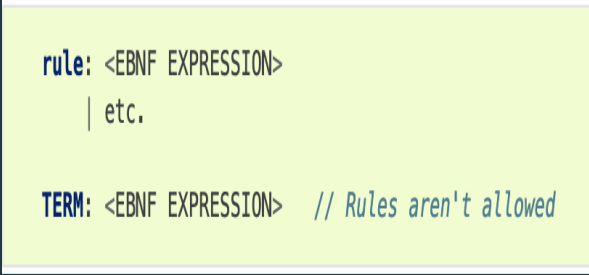
The project is based on creating of grammar in a text file. In order to build one I referred to Python's built in parser library called Lark that provides a structure called Lark grammar to define grammar rules. Once the grammar rules established we can then implement a lexer and a parser that would generate a data structure for the interpreter to process.

The implementation of the lexer, parser and the interpreter will be covered in the following section.

III. IMPLEMENTATION

A. Grammar

Every programming language has a grammar, which is a set of instructions on how to write valid statements for that language. So, for this project a valid syntax for the mathematical expressions was first implemented. It was done using Python's Lark grammar (Fig. 2), a design that defines a grammar using rules and terminals.


```
rule: <EBNF EXPRESSION>
    | etc.


TERM: <EBNF EXPRESSION>   // Rules aren't allowed
```

Fig. 2. Lark grammar design

Terminals here can be thought of as alphabets and rule a definition for the structure of a language. Rules and terminals are written as Extended Backus Naur Form (EBNF) expressions.

Once the grammar rule was constructed on a text file (Fig. 3), a test was run on it using Python's Lark lexer and parser. Larks lexer and parser were able to successfully process a given mathematical expression into a sequence of tokens and generate an abstract syntax tree (AST). After this success, to better understand how a lexer and a parser really worked I implemented my own lexer and parser versions in Python.

```
?start: expr

?expr: term
   | expr "+" term  -> add
   | expr "-" term  -> sub


?term: factor
   | term "*" factor  -> mul
   | term "/" factor  -> div

?factor: NUMBER        -> number
   | "(" expr ")"



%declare STR INT
%import common.NUMBER
%import common.FLOAT
%import common.WS_INLINE

%ignore WS_INLINE
```

Fig. 3. grammar.txt file

B. Lexer

Lexer is a part of an interpreter that turns a sequence of characters into tokens. In the project our lexer function was implemented in Python using two classes - a Token class and a Lexer class.

A Lexer object in the program takes in a user input in a form of mathematical expression. It breaks the input expressions into sequence of characters saved as cur_tok as shown below.

```python
class Lexer(object):
    def __init__(self, input):
        self.input = input
        self.pos = 0
        self.cur_tok = self.input[self.pos]
```

Fig. 4. Lexer class constructor in Python

The characters are then mapped to its token values when the lexer object calls the getTokens() method. The getTokens() (Fig. 6.) method and the token (Fig. 5.) values are demonstrated below.

```
# Tokens
PLUS, MINUS, MUL, DIV, LPAREN, RPAREN, NUMBER, EOF = ('add','sub','mul','div','(',')','number', 'EOF')
```

Fig. 5. Tokens

```python
def getTokens(self):
    while self.cur_tok is not None:
        if self.cur_tok.isdigit():
            return Token(NUMBER, self.number())
        if self.cur_tok == '+':
            self.nextToken()
            return Token(PLUS, '+')
        if self.cur_tok == '-':
            self.nextToken()
            return Token(MINUS, '-')
        if self.cur_tok == '*':
            self.nextToken()
            return Token(MUL, '*')
        if self.cur_tok == '/':
            self.nextToken()
            return Token(DIV, '/')
        if self.cur_tok == '(':
            self.nextToken()
            return Token(LPAREN, '(')
        if self.cur_tok == ')':
            self.nextToken()
            return Token(RPAREN, ')')
        if self.cur_tok.isspace():
            self.ignoreWS()
            continue
        self.error()
    return Token(EOF, None)
```

Fig. 6. getTokesn() method

C. Parser

The job of Parser is to analyze the tokens generated by the lexer and build concrete parse tree which is then converted into AST with the help any of the several parsing algorithms such as: LL(k) parsing also called recursive descent parsing (Example: the parser combinators of Haskell.); CYK Algorithm; Earley's Algorithm; LR Parsing Algorithm; Packrat Parsing Algorithm; Pratt Parsing Algorithm to name a few. The parser used in

this is the Look-Ahead LR parser (LALR).

```python
def printAst(text):
    grammer_file = open("pure_basic_grammar.txt", "r")
    grammar = grammer_file.read()
    grammer_file.close()
    parser = Lark(grammar, parser='lalr')
    ast = parser.parse(text)
    print(ast.pretty())
```

Fig. 7 printAst() uses Lark parser to print the AST.

In the program, the parser object after receiving a token from lexer calls the parse() method to produce AST following the rules of the grammar. The two figures below (Fig. 8 and Fig. 9) presents the Parser class that implements the parse() method which in turn calls the expr(), term() and factor() method to generate a tree data structure as defined by the grammar.

```python
        return node
    def term(self):
        """ ?term: factor
        | term "*" factor   -> mul
        | term "/" factor   -> div
        """
        node = self.factor()
        while self.curr.type in (MUL,DIV):...
        return node

    def expr(self):
        """ ?expr: term
        | expr "+" term    -> add
        | expr "-" term    -> sub"""

        node = self.term()
        while self.curr.type in (PLUS, MINUS):
            token = self.curr
            if token.type == PLUS:
                self.matchToken(PLUS)
            elif token.type == MINUS:
                self.matchToken(MINUS)
            node = BinaryOp(left=node, op=token, right=self.term())
        return node

    def parse(self):
        return self.expr()
```

Fig. 8 Parser class methods

```python
class Parser(object):

    def __init__(self, lexer):
        self.lexer = lexer
        self.curr = self.lexer.getTokens()


    def error(self):
        raise Exception('Invalid Syntax')

    def matchToken(self, type_):
        if self.curr.type == type_:
            self.curr = self.lexer.getTokens()

        else:
            self.error()
    def factor(self):
        """ ?factor: NUMBER    -> number
        | "-" atom         -> neg
        | NAME             -> var
        | "(" sum ")"""
        token = self.curr
        if token.type == NUMBER:
            self.matchToken(NUMBER)
```

Fig. 9 Parser class.

To AST view in a tree form we used Lark library which is general purpose lexer and parser for this project.

D. Abstract Syntax Tree

An Abstract Syntax Tree is a tree data structure, generated when a parser computes a sequence of token based on the grammar defined. AST represents an abstract syntactic structure of the language construct. This basically mean that in the tree structure the parent nodes represent the operator and the child nodes represent the operands given in a mathematical expression (user input).

It is critical for the abstract syntax tree be implemented correctly because interpreter uses this data structure to carry out its execution of the code. Since the interpreter in

the program is designed to execute mathematical binary operation, it is important to take into account the precedence of the operators in the expression.

From the very start, the grammar is constructed in a manner that it takes care of the binding of the expressions through rules defined. The grammar makes the parser place the operators with higher precedence at the bottom of nodes of the tree structure, AST such that when interpreter traverses through the nodes, they get executed first . The figure (Fig. 10) in the following Testing section demonstrates the correct precedence of operators being applied.

The figure below is a representation of AST.
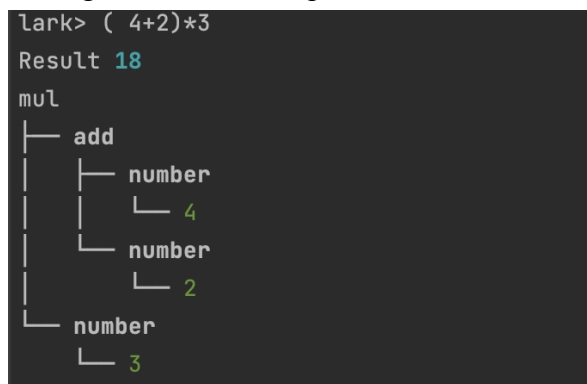
Fig. 9 Abstract Syntax Tree

E. Interpreter

Interpreter is the final step in interpreting a given expression of the required grammar. In the implemented Interpreter it calls the visit_BinaryOp() method to visit the different nodes of the tree and returns the result to the user. visit_BinaryOp() is responsible for calling the required operation for a given token.

Fig. 10. Interpreter function

IV.   TESTING

The program allows for three options at the startup:
- When selected 1: Calculator open and the user can give in expressions with one digit to get the results.
- When selected 2: Users can pass in mathematical expressions and Lark parser will print the AST.
- When selected 3: User can view the AST for hard coded expressions.

```
elif text == "3":
    pure_basic.printAst("1+1")
    pure_basic.printAst("1+2-3")
    pure_basic.printAst("1+2-3*2")
    pure_basic.printAst("12/2+2*3")
    pure_basic.printAst("1+2-3-(3)/(23)+2+2")
```

```
own> 1 + 1
Result 2
own> 2 - 3
Result -1
own> 4/3
Result 1.3333333333333333
own> 5 * 3
Result 15
own>  (4 + 4) * 3
Result 24
own>  4 + ( 4 * 3 )
Result 16
own>
```

Fig. 9 . Test Results

V.   CONCLUSION

This project helped me understand the basics of how different programming languages work behind the scenes. Interpreter makes it so much easier for the user to write a huge number of statements and execute them without having to compile first. This project serves as a base for implementing a more complex programming language which can include features like error handling, branching, functions and variable assignments which we look forward to exploring and implementing.

[VI]. REFERENCES
[1].https://ruslanspivak.com/lsbasi-part1/
[2]https://github.com/davidcallanan/py-myopl-code/commit/d640ca8e82eeac3cf9b0df4e9b6b3f57d94c57b5

[3]https://lark-parser.readthedocs.io/en/latest/examples/calc.html