SWIFT INTERPRETER TEAM 5

(Written by studying “Crafting Interpreters” book)

***SECTIONS*:**

- *Features/Constructs*
- *Explanation of code*
- *Dependencies Required*
- *Challenges*
- *What We Learnt*
- *Contributions*

***FEATURES/CONSTRUCTS*:**

1. *Variable Assignment: Assignment of Strings, Integers and Floating Point Numbers to Variables.*
2. *Arithmetic Prowess: Arithmetic Operators(+, -, *,/) on numerical expressions with operator precedence.*
3. *Comparison Mastery: Comparison Operators(>, <, >=, <=, ==, !=) on numerical expressions.*
4. *Conditional Control: If-Else if-Else statement/ladder.*
5. *Repetitive Tasks: While loop.*

6. *Output Mastery: Print statement, which can print a combination of Strings, Numerical Expressions and variables(their value).*

CODE EXPLANATION:

The code is structured into three subparts:

Scanner:

The Scanner class in the provided Swift interpreter code is responsible for converting the source code into a sequence of tokens , which are the basic building blocks of the Swift language.

The Scanner class processes the source code character by character, recognizing lexemes and converting them into tokens. It handles various types of tokens, such as identifiers, keywords, literals, and punctuation symbols, based on the Swift language grammar. The resulting list of tokens is then passed to the parser for further interpretation.

Parser:

The Parser class in the provided Swift interpreter code is responsible for parsing a list of tokens generated by the Scanner class and constructing an abstract syntax tree (AST). The AST represents the syntactic structure of the Swift code and is used as

an intermediate step before interpreting or executing the code.

The Parser class navigates through the list of tokens and applies recursive descent parsing to construct the AST. It defines methods for parsing various grammar rules, such as expressions, statements, and declarations. The resulting AST is a hierarchical representation of the Swift code's syntactic structure, which is then used for further interpretation or execution.

Interpreter:

The Interpreter class in the provided Swift interpreter code is responsible for executing or interpreting the abstract syntax tree (AST) generated by the Parser class. It traverses the AST and performs the necessary actions for each type of statement or expression encountered.

The Interpreter class implements the visitor pattern for both expressions (Expr.Visitor<Object>) and statements (Stmt.Visitor<Void>). It provides methods for interpreting a list of statements, executing individual statements, and evaluating expressions. The class maintains environments for variable scope, and it interacts with the Environment class for variable resolution and

assignment. The Interpreter class plays a crucial role in the execution phase of the Swift interpreter, ensuring that the semantics of the Swift language are properly implemented.

Resolver:

The Resolver class in the provided Swift interpreter code is responsible for resolving variable bindings and handling scope during the interpretation process. It performs a static analysis pass over the abstract syntax tree (AST) generated by the Parser class to determine the scope of variables and functions.

In summary, the Resolver class helps in determining the scope of variables, checking for variable use before declaration, and handling variable resolution during the interpretation process. It collaborates with the Interpreter class to ensure that variable bindings are appropriately resolved at runtime.

BNF:

The fundamental BNF of a numerical expression is:

<expr>::= <term>|<expr>+<term>|<expr>-<term>

```
<term>::=
<factor>|<term>*<factor>|<term>/<factor>|<term>%<factor>

<factor>::= Integer | Float | Variable | <unaryop><factor> | (<expr>)
```

Similarly, the BNF for a comparison operation is:

```
<comparison>::= <expr><comparison_op><expr>

<comparison_op>::= >|<|>=|<=|==|!=
```

Now, let us shift gears and go to the BNF of an entire program in our subset of the SWIFT language:

```
<program>::= <compound_statement> EOF

<compound_statement>::= <statement_list>

<statement_list>::= <statement>|<statement><statement_list>
```

```
<statement>::= LET <assignment_statement> SEMI| VAR
<assignment_statement> SEMI| <assignment_statement> SEMI
|<print_statement> SEMI| IF<if_statement>| WHILE

<while_statement>|empty
```

```
<assignment_statement>::= variable=<expr> SEMI |
variable=string SEMI
```

```
<print_statement>::= LPAREN <printable_list> RPAREN SEMI

<printable_list>::= <printable>|<printable>COMMA<printable_list>
```

```
<printable>::= string| <expr>
```

```
<while_statement>::= LPAREN <comparison> RPAREN <block>
```

```
<block>::= LCURL<block_statement_list>RCURL
```

<block_statement_list>::=<statement>|<block_statement_list><block_statement> <block_statement>::= LET <assignment_statement> SEMI| VAR<assignment_statement> SEMI| <assignment_statement> SEMI |<print_statement> SEMI

<if_statement>::= LPAREN <comparison> RPAREN <block> <else_and_else_if_tree>

<else_and_else_if_tree>::= empty| <else_if_tree> | <else_if_tree> ELSE <block>

<else_if_tree>::= empty| ELSE IF <block> | <else_if_tree> ELSE IF <block>

Note: Although SEMICOLON(;) is optional for Assignment and Print Statements in SWIFT, we decided to enforce them in our subset so that the structure of the program looks neat.

DEPENDENCIES:

Java Runtime Environment(JRE)

(Note: The test cases must be in the appropriate file for the interpreter to work)

CHALLENGES:

Java's use of the JVM and its memory management model introduces additional complexities:

- *JVM Compatibility: The interpreter needs to be compatible with the JVM and its runtime*

environment, ensuring that generated bytecode is executed correctly.

- *JVM Memory Management: Java's garbage collection mechanism must be integrated into the interpreter to handle memory allocation and deallocation for Swift objects.*
- *Closures: Closures are nested functions that can access variables from their enclosing scope, requiring the interpreter to handle scope resolution and variable lifetime.*
- *Difficulty in typing conditional statements*
- *Error Handling and Recovery: The interpreter needs to effectively handle unexpected input and error conditions, providing meaningful error messages and recovery mechanisms.*

These challenges that we overcame.

***WHAT WE LEARNT*:**

1. *We learnt about the steps involved in the interpretation process(we already learnt it in the theory part, but now we got a practical feel of it)*
2. *We learnt how BNF is a powerful weapon in terms of creating parsers and parse trees. We learnt how to write BNF for a (subset of a) PL and how to use/convert that BNF into*

functions/methods that can check for syntax errors.

3. *We learnt how breaking down a system into modular subcomponents(in this case, into lexer, parser, interpreter etc.) can be very useful.*
4. *We learnt the syntax of the SWIFT programming language and how it is similar/different from common programming languages like Python, C, Java etc.*
5. *Understanding Language Design Principles: By comparing and contrasting Swift and Java, we gain insights into the fundamental principles of language design. We appreciate how different languages make trade-offs between expressiveness, performance, and safety.*
6. *Appreciating Language Evolution: As Swift and Java evolve, we need to adapt the interpreter accordingly. We gain insights into the process of language standardization and version control.*

CONTRIBUTIONS:

K.Akhil Solomon(CS22B032):

1.Participation in Group/Team discussions

2.The Classes written are "Interpreter","Parser"," Swift".

3.Contributed for readme part.

L.Sampath(CS22B033):

1. *Participation in Group/Team discussions.*
2. *The Classes written are "Scanner","Token","TokenType".*
3. *Contributed for readme part.*

I.Kalyan Anudeep(CS22B025):

1.Participation in Group/Team discussions

2.The Classes written are "Stmt","Expr"," Resolver".

3.Contributed for readme part.

D.A.Raghuveer(CS22B019):

1. *Participation in Group/Team discussion.*
2. *The Classes written are "SwiftCallable","SwiftInstance","Swiftfunction" , "SwiftClass"*
3. *Contributed for readme part.*

B.Lachiram Nayak(CS22B012):

1. *Participation in Group/Team discussions*
2. *The Classes written are "Environment","Return","Runtimeerror"*
3. *Contributed for readme part.*

###THE END###