Broca : Final Report

Anwar Mujeeb, *Student, SJSU* , Russell Semsem, *Student, SJSU,* and Saptarshi Sengupta, *Professor, SJSU*

***Abstract*—Broca is a simple implementation of a functional programming language powered by Java**

***Index Terms*—Lexer, Parser, Abstract Syntax Tree, Interpreter, REPL**

I. Introduction

BROCA is a minimal functional programming language that has various functionalities including computing basic arithmetic, declaring variables, using logical statements, functions, and even higher-order functions. This language was implemented to illustrate how languages are made into tokens, organized, and interpreted.

II. Overview

We have finished our basic implementation of Broca, our language definition supports type inferences, functions, logical statements, and loops. In the following report, we will be illustrating the design of our language, the structure and mechanics of our lexer, parser, and interpreter, and our REPL environment that users can use to directly interact with our language. We will also be looking at features that we plan to add in the future such as implementation of lists and multi-line processing. Finally, we will discuss our experience with building Broca and what we have learned from its implementation.

A. Project Timeline

Our timeline throughout the semester for this project was primarily organized by order of priority, here is a simple breakdown:

1) Language Definition - Develop Grammar
2) Add Arithmetic and Variable Support - Create initial Lexer, Parser, Interpreter and REPL
3) Add Logical Operations and While, For Loops
4) Add Function Support, Higher Order Functions, Start List Implementation

B. Language Design

Broca's language design showcases a syntax that is both expressive and unambiguous, adhering closely to functional programming principles like pure functions and a declarative approach. It encompasses a range of features essential for fundamental functional programming and exhibits clear delineations to prevent common programming errors. The language design is important for the language processing system.

We have defined the following grammar in Backus-Naur Form (BNF) for our language design.

Anwar Mujeeb, Russell Semsem are with San Jose State University.
Manuscript received December 2023;

```
<expr> ::= "KEYWORD:VAR" "IDENTIFIER"
         "EQUAL" <arith-expr>
         |"IDENTIFIER" "EQUAL" <arith-expr>
         | <comp-expr>(("KEYWORD:AND"|
         "KEYWORD:OR") <comp-expr>)*

<comp-expr> ::= NOT <comp-expr>
              | <arthi-expr> (("EE"|"LT"|
              "GT"|"LTE"|"GTE")
              <arthi-expr>)*

<arith-expr> ::= <term> (("PLUS" |
               "MINUS") <term>)*

<term> ::= <factor> (("MUL" |
         "DIV") <factor>)*

<call-func> ::= <factor> ("LEFT_PAREN"
              (<expr> ("COMMA"
              "IDENTIFIER")*)?
              "RIGHT_PAREN")?

<factor> ::= ("PLUS" | "MINUS") <factor>
           | "INT" | "DOUBLE" | "IDENTIFIER"
           | "LEFT_PAREN" <expr> "RIGHT_PAREN"
           | <if=expr>
           | <for-expr>
           | <while-expr>
           | <func-def>

<if-expr> ::= "KEYWORD:IF" <expr>
            "KEYWORD:THEN" <expr>
            ("KEYWORD:ELIF" <expr>
            "KEYWORD:THEN" <expr>)*
            ("KEYWORD:ELSE" <expr>)?

<for-expr> ::= "KEYWORD:FOR" "IDENTIFIER:EQ"
             <expr> "KEYWORD:TO" <expr>
             ("KEYWORD:STEP" <expr>)?
             "KEYWORD:THEN" <expr>

<while-expr> ::= "KEYWORD:WHILE"
           <expr> "KEYWORD:THEN" <expr>

<func-def> ::= "KEYWORD:FN" "IDENTIFIER"?
         "LEFT_PAREN" ("IDENTIFIER"
         ("COMMA" "IDENTIFIER")*)?
         "RIGHT_PAREN" "ARROW" <expr>
```

C. Functionality

Broca is designed to handle a variety of essential programming features, making it versatile. The capabilities of Broca are the following:

- Basic Arithmetic
- Variables
- Logical Operations
- Conditional Statements
- While and For Loops
- Functions
- Higher-Order Functions

Broca can support basic arithmetic operations allowing for numerical computations. Variables enable the storage and manipulation of data values in future input statements. Logical operations and conditional statements allow complex decision-making and control flow on Boolean logic. For iterative processes, Broca can perform both 'While' and 'For' loops, providing the ability to repeat tasks until a certain condition is met.

Functions allow for code reuse and modular programming by encapsulating expression statements that perform specific tasks. Finally, Broca supports higher-order functions, an essential feature of functional programming. This is achieved by treating functions as variables or first-class functions. Thus, we can use functions as parameters and return values.

III. Language Processing System

Broca's primary components consist of the Read-Evaluate-Print Loop (REPL), Lexical Analysis (Lexer), the Syntax Analysis (Parser), and the Interpreter. Each component serves a distinct role in the process of transforming source code into executable instructions.

A. Read-Evaluate-Print Loop: REPL

The REPL serves as our execution environment, as it orchestrates the flow of data from the initial scanning of user input to the output of the interpreter's results. It begins by scanning the input, received as a string, and forwards it to the lexer. The lexer then deconstructs this input into a sequence of tokens, which it subsequently returns. These tokens are then conveyed by the REPL to the parser, which constructs the corresponding Abstract Syntax Tree (AST) and returns its root node. Following this, the REPL hands over the root node to the interpreter, which executes the necessary operations as it traverses through the AST. Finally, the REPL displays the interpreter's results and stands ready to process the next input from the user.

B. Lexical Analysis: Lexer

The lexer's main role is to create a list of tokens from the input string. To do this, we have defined the lexer program, the Token class, and the TokenType enumeration. The lexer program's scanning process iterates through each character of the input, initiating different Token creation processes. Creating tokens for characters such as '+', '-', or ')' are rather quick, as they are identified as TokenTypes PLUS, MINUS, and RIGHT-PAREN. In addition, the lexer knows how to differentiate characters such as '=' or "==" when creating an EQUAL token for assigning variables or an EE token (equals-equals) for logical comparisons. Finally, the lexer is capable of identifying longer substrings of the input such as numbers or variable names by finding the beginning and end index and creating their respective tokens such as TokenType IDENTIFIER, INT, or DOUBLE. These tokens are appended to a tokens list and returned to the REPL. The following is an example of tokens created from an input string:

```
Input: "VAR a = (3.52+4)"
Tokens:
    [TokenType: KEYWORD, Value: VAR]
    [TokenType: IDENTIFIER, Value: a]
    [TokenType: EQUAL, Value: =]
    [TokenType: LEFT_PAREN, Value: (]
    [TokenType: DOUBLE, Value: 3.52]
    [TokenType: PLUS, Value: +]
    [TokenType: INT, Value: 4]
    [TokenType: RIGHT_PAREN, Value: )]
```

C. Syntax Analysis: Parser

The parser's main role is to take the sequence of tokens from the lexer and create an Abstract Syntax Tree (AST). To do this, we have defined the parser program and ASTNode class with a variety of subclass nodes. Here, the grammar that we defined earlier plays a critical role as the parser program is a reflection of it. Thus, it is important to follow the flow of the grammar to ensure we create the correct AST. Certain sequences of tokens will create certain ASTNodes. For example, if a token has a TokenType of PLUS, MINUS, MUL, or DIV, it will know to create a binary operator node (BinOpNode). Keywords such as "FN" for functions will create a defining function node (DefFuncNode). Since some nodes can have fields of type ASTNode, we can grow our AST. For example, BinOpNode has two ASTNode types to represent its left and right sides. Once the parser has gone through all the tokens, the AST is completed, and the root node is returned to be used by the interpreter. The following are ASTs for basic arithmetic:

```
Arithmetic: 2+3*4
         BinOpNode(+)
             /\
            /  \
NumberNode(2)  BinOpNode(*)
                   /\
                  /  \
      NumberNode(3)  NumberNode(4)

Arithmetic: (2+3)*4
                BinOpNode(*)
                    /\
                   /  \
        BinOpNode(+)  NumberNode(4)
             /\
            /  \
NumberNode(2)   NumberNode(3)
```

The examples highlight the versatility of the parser in producing different abstract syntax trees according to the input and the tokens list.

D. Interpreter

The interpreter's main role is to take the root node of the AST from the parser and return a result. To do this, we have created an interpreter program and a Symbol Table. The interpreter begins to traverse through the AST, and depending on the node it visits, it will execute a certain function and its logical instructions to return a result. For example, when visiting the BinOpNode, the interpreter will visit the visitBinOpNode(BinOpNode node) and begin a set of instructions for extracting the numeric value of the left and right field ASTNodes and perform an arithmetic operation according to the node's operator token. The result is then returned. This is a similar process for performing other features such as logical statements, loops, and functions, as their ASTNodes have a corresponding interpreter function. The Symbol Table, similar to a HashMap, plays a critical role in the interpreter, as it stores the names of TokenType IDENTIFIER and its respective numeric values or instructions. The Symbol Table makes it possible to call and use variables and functions by their identifier names in future input statements and execute accordingly.

IV. Conclusion

We have created Broca to illustrate how languages such as functional languages are created and implemented, while Broca is not strictly functional (it has mutable objects and variables), it allows the user to get a deeper understanding of the functional paradigm without having to leave the conveniences of object-oriented paradigm, this makes it similar to a language such as Python. In conclusion, our language uses a REPL as the main entry point in our language, it takes in inputs one line at at time and then passes these lines to our Lexer which tokenizes the string and passes it to our Parser to build an Abstract Syntax Tree using the Token definition as guidelines, it then uses the Interpreter to translate these Abstract Syntax Trees into Java and execute and respond to the REPL.

A. Future Implementations

Our future implementations of Broca include the following:

- Lists, an essential component of most programming languages, can enhance Broca to be effective on large data sets and compute more complex functions. This includes doing comprehensions such as lazy evaluation and implementing Strings. To do this we will have to update our BNF definition to add more tokens and types.
- Multi-line interpreting, which means entire files should be able to be passed through Broca so that users can save their work and make workflows more effective. This can be done by creating a standalone interpreter that runs single files.

B. Learning Process

Through the creation of Broca, we have been fortunate to gain a deeper understanding of how languages, specifically interpreted languages work. This has been through applying materials that we learned in class directly to the implementation of our project.

- Functional Programming: we were able to gain a deeper understanding of functional programming languages as they are taught. This was due to the fact as we implemented our language we had to understand how functional paradigms would fit with our language definition and make adjustments so that functions are treated as first-class functions.
- Complex Data Structures and Use Cases: Throughout the various stages of our programming language we had to use complex data structures such as Graphs and HashMaps and others to implement concepts such as Abstract Syntax Trees and Symbol Tables. We had to use design patterns such as the Singleton Pattern, Polymorphism, and Interfaces to ensure the reproducibility of our Nodes and Token, which allowed us to scale our language to more features. This all in all gave us a deeper understanding of data structures and optimal ways to execute them.
- Language Workings: Understanding the various stages of a language's implementation and execution, allowed us to better understand existing languages such as Python and Java. Implementing our interpreter and parser gave us better debugging skills when it comes to working with more common languages as we now understand how they work under the hood.

References

[1] CodingWithAdam. (n.d.). Introduction to Lexers, Parsers and Interpreters with Chevrotain. Blog. Retrieved from https://dev.to/codingwithadam/introduction-to-lexers-parsers-and-interpreters-with-chevrotain-5c7b.

[2] GeeksforGeeks. (n.d.). Functional Programming Paradigm. Article. Retrieved from https://www.geeksforgeeks.org/functional-programming-paradigm/.

[3] Kundel, D. (n.d.). ASTs - What are they and how to use them. Blog. Retrieved from https://www.twilio.com/blog/abstract-syntax-trees.

[4] Grandinetti, P. (n.d.).What Is A Programming Language Grammar?. Article. Retrieved from https://pgrandinetti.github.io/compilers/page/what-is-a-programming-language-grammar/.