The Turkish Programming Language: BEREN

İsmet Eren Yurtcu

Berkin Yıldırım

Manisa Celal Bayar University Faculty of Engineering

ABSTRACT *The report explores the development of a simple programming language (PL), detailing its lexical analysis (Lexer) and parsing (Parser) mechanisms implemented in Java. The PL encompasses basic arithmetic operations, variable declarations, conditional statements (if-else), and loop constructs (while).*
The Lexer breaks down input text into tokens, identifying units like numbers, identifiers, and operators using a stream-based approach. It gracefully handles whitespace and unknown characters.
Meanwhile, the Parser interprets the token stream to grasp the program's structure and semantics. Employing recursive descent parsing, it constructs a parse tree representing the program's syntactic hierarchy, aiding subsequent analysis and interpretation.
The report delves into the design, implementation, and functionality of each component, emphasizing the pivotal role of lexical and syntactic analysis in fostering effective communication between developers and machines. It sets the stage for advanced language features and program comprehension.
By dissecting the PL and elucidating each component's role, the report provides insights into language design principles, parsing algorithms, and the complexities of program execution.

I. INTRODUCTION

Welcome to the intriguing journey into the creation of the BEREN programming language, where we unravel the intricacies of lexical analysis (Lexer) and parsing (Parser) mechanisms implemented in Java. BEREN, a language crafted with precision, encapsulates the essence of basic arithmetic operations, variable declarations, conditional statements (if-else), and loop constructs (while).
In our expedition, the Lexer stands as a vigilant sentinel, meticulously dissecting the input text into coherent tokens, identifying numerical entities, symbolic identifiers, and operational characters with finesse. Navigating through whitespace and unknown characters, it illuminates the path toward comprehension.
Meanwhile, the Parser assumes the mantle of understanding, deciphering the token stream to unravel the structure and semantics of BEREN. Through the artistry of recursive descent parsing, it constructs a majestic parse tree, a testament to the syntactic beauty inherent in BEREN, facilitating profound analysis and interpretation.
This report embarks on an odyssey through the design, implementation, and functionality of each component, illuminating the pivotal role played by lexical and syntactic analysis in bridging human intent with machine execution. It lays the foundation for advanced language features and profound program comprehension, promising a voyage of discovery into the heart of language design and parsing algorithms.

II. METHODS

Methods of Lexer Class:

- **public Lexer(String fileName): The constructor of the Lexer class takes a file name and creates a BufferedReader to read the file.**
- **private void readNextLine(): Reads the next line of the file and updates the Lexer's state.**
- **private void advance(): Moves the current character position to the next character.**
- **public List<Token> tokenize(): Breaks down the input text into tokens and returns the list of these tokens.**
- **private Token number(): Defines a number token.**
- **private Token identifier(): Defines an identifier token.**

- **private boolean isLetter(int c): Determines whether the given character is a letter.**
- **public void printTokens(): Prints the list of tokens to the console.**

Methods of Parser Class:

- **public Parser(List<Token> tokens): The constructor of the Parser class takes a list of tokens and initializes necessary data structures for processing.**
- **private Token currentToken(): Returns the current token.**
- **private void advance(): Advances the current token to the next one.**
- **private boolean match(TokenType type): Matches the type of the current token with the specified type and advances.**
- **public void parse(): Processes the list of tokens and analyzes the syntax of the program.**
- **private void statement(): Analyzes a statement.**
- **private void declaration(): Analyzes a variable declaration statement.**
- **private void assignment(): Analyzes an assignment statement.**
- **private int expression(): Evaluates an expression.**
- **private int term(): Evaluates a term.**
- **private int factor(): Evaluates a factor.**
- **private void ifStatement(): Analyzes an if statement.**
- **private void whileStatement(): Analyzes a while loop statement.**
- **private boolean condition(): Evaluates a condition.**
- **private void error(String message): Terminates the program with an error message.**

- **public void printParseTree(): Prints the parse tree in BNF format to the console.**

III. CODE EXPLANATION

The provided code represents an implementation of the BEREN programming language, featuring both lexical analysis (Lexer) and parsing (Parser) components. Here's a breakdown of the main functionalities:

Lexer:

The Lexer component serves as the initial stage of code interpretation. It takes raw input text and tokenizes it, breaking it down into individual units called tokens. These tokens represent fundamental elements of the programming language, such as numbers, identifiers (variable names), arithmetic operators, keywords (IF, ELSE, WHILE), and other symbols.

The Lexer scans through the input text character by character, identifying patterns and categorizing them into specific token types. It handles whitespace gracefully, disregarding it unless it separates meaningful tokens. Additionally, it recognizes unknown characters and categorizes them as such.

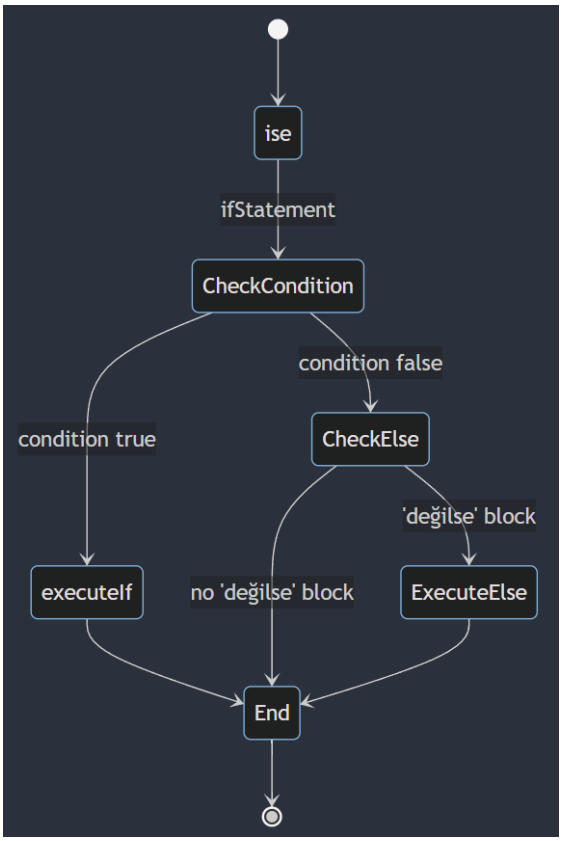
Parser:

Once the Lexer has generated a stream of tokens, the Parser component steps in to make sense of these tokens. It takes the token stream as input and analyzes the structure and syntax of the code. The Parser is responsible for constructing a hierarchical representation of the program's syntax, commonly known as a parse tree.

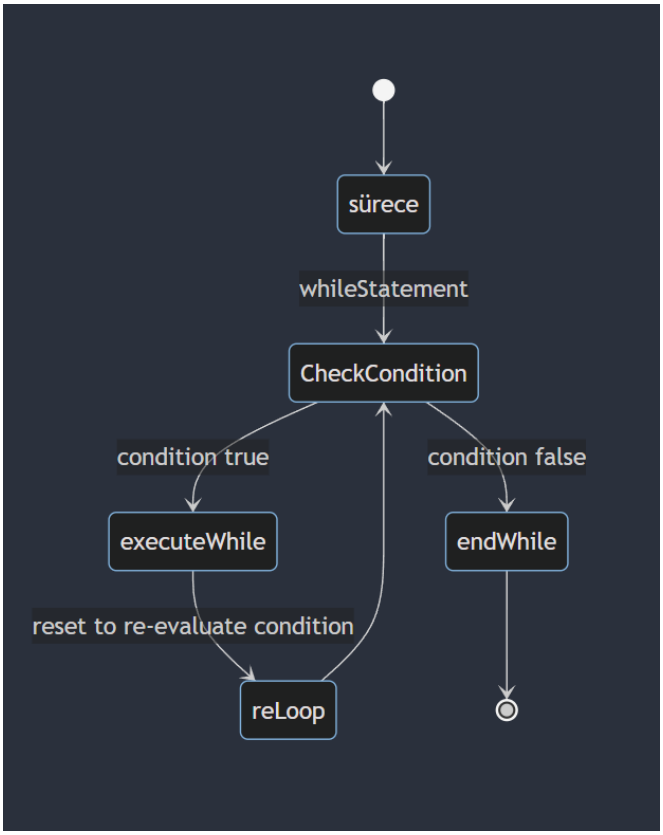
The Parser employs recursive descent parsing, a top-down parsing technique, to systematically analyze the token stream and build the parse tree. It traverses the token stream, identifying different types of statements, expressions, conditions, and control flow constructs (such as if-else statements and while loops). As it encounters tokens, the Parser matches them against expected patterns defined by the language grammar, ensuring syntactic correctness.

IV. STATE DIAGRAM

If Statement



While Statement



V. BNF

<program> ::= <degiskenBildirimi> <ifDeyimi> <whileDeyimi> EOF

<degiskenBildirimi> ::= SAYI ID ASSIGN NUMBER <degiskenBildirimi> | ε

<ifDeyimi> ::= IF LPAREN <kosul> RPAREN <ise> <değilse>

<ise> ::= <degiskenBildirimi>

<değilse> ::= ELSE <degiskenBildirimi>

<whileDeyimi> ::= WHILE LPAREN <kosul> RPAREN <dongu>

<dongu> ::= <atama> <dongu> | ε

<atama> ::= ID ASSIGN <ifade>

<ifade> ::= ID | ID PLUS NUMBER | ID MINUS NUMBER | ID MULTIPLY NUMBER | ID DIVIDE NUMBER

<kosul> ::= ID GREATER ID | ID LESS ID

<IDENT> ::= [a-zA-ZğüşöçĞÜŞÖÇ][a-zA-ZğüşöçĞÜŞÖÇ0-9]*

<INT_LIT> ::= [0-9]+

<TokenType> ::= SAYI | ID | NUMBER | ASSIGN | PLUS | MINUS | MULTIPLY | DIVIDE |
IF | ELSE | WHILE | LPAREN | RPAREN | GREATER | LESS | EOF | UNKNOWN

VI. OUTPUT

```
Okunmasini istediginiz text dosyasi numarasini yazin: 1
SAYI sayi
ID x
ASSIGN =
NUMBER 5
SAYI sayi
ID y
ASSIGN =
NUMBER 10
IF ise
LPAREN (
ID x
GREATER >
ID y
RPAREN )
SAYI sayi
ID z
ASSIGN =
NUMBER 20
ELSE değilse
SAYI sayi
ID z
ASSIGN =
NUMBER 30
WHILE sürece
LPAREN (
ID x
LESS <
ID y
RPAREN )
ID x
ASSIGN =
ID x
PLUS +
NUMBER 1
EOF
ID x
EQUAL =
<ifade> girildi
NUMBER 5
<ifade> cikildi
ID y
EQUAL =
<ifade> girildi
NUMBER 10
<ifade> cikildi
LPAREN (
<ifDeyimi> girildi
<ise> girildi
ID x
GREATER >
ID y
<ise> cikildi
<degilse> girildi
ID z
EQUAL =
<ifade> girildi
NUMBER 30
<ifade> cikildi
<degilse> cikildi
<ifDeyimi> cikildi
LPAREN (
<whileDeyimi> girildi
ID x
LESS <
ID y
ID x
EQUAL =
<ifade> girildi
ID x
PLUS +
NUMBER 1
<ifade> cikildi
ID x
LESS <
ID y
ID x
EQUAL =
<ifade> girildi
ID x
PLUS +
NUMBER 1
<ifade> cikildi
ID x
LESS <
ID y
ID x
EQUAL =
<ifade> girildi
ID x
PLUS +
NUMBER 1
<ifade> cikildi
ID x
LESS <
ID y
ID x
EQUAL =
<ifade> girildi
ID x
PLUS +
NUMBER 1
<ifade> cikildi
ID x
LESS <
ID y
<whileDeyimi> cikildi
ID x
EQUAL =
<ifade> girildi
ID x
PLUS +
NUMBER 1
<ifade> cikildi
BNF FORMAT
<degiskenBildirimi> -> x = <ifade> 5 <degiskenBildirimi>
<degiskenBildirimi> -> y = <ifade> 10 <degiskenBildirimi>
<ifDeyimi> -> ( <kosul> ) x > y <degiskenBildirimi> -> z = <ifade> 30 <degiskenBildirimi>
<ifDeyimi>
<whileDeyimi> -> ( <kosul> ) <dongu> x < y <atama> -> x = <ifade> x + 1 <atama>
x < y <atama> -> x = <ifade> x + 1 <atama>
x < y <atama> -> x = <ifade> x + 1 <atama>
x < y <atama> -> x = <ifade> x + 1 <atama>
x < y <atama> -> x = <ifade> x + 1 <atama>
x < y <whileDeyimi>
<atama> -> x = <ifade> x + 1 <atama>

BUILD SUCCESSFUL (total time: 1 second)
```

```
Okunmasini istediginiz text dosyasi numarasini yazin: 2
SAYI sayi
ID x
ASSIGN =
NUMBER 10
IF ise
LPAREN (
ID x
GREATER >
NUMBER 5
SAYI sayi
ID y
ASSIGN =
NUMBER 20
ELSE değilse
SAYI sayi
ID z
ASSIGN =
NUMBER 30
Error: Expected closing parenthesis at token SAYI sayi
EOF
ID x
EQUAL =
<ifade> girildi
NUMBER 10
<ifade> cikildi
LPAREN (
<ifDeyimi> girildi
<ise> girildi
ID x
GREATER >
NUMBER 5
C:\Users\ieren\AppData\Local\NetBeans\Cache\16\executor-snippets\run.xml:111: The following error occurred while executing this line:
C:\Users\ieren\AppData\Local\NetBeans\Cache\16\executor-snippets\run.xml:68: Java returned: 1
BUILD FAILED (total time: 3 seconds)
```