Compiler Documentation

Francisco Prestamo

October 4, 2023

Contents

1 Introduction

2 Lexer

The lexer component of the compiler performs lexical analysis, which involves breaking the input source code into individual tokens. The lexer reads characters from the input and identifies tokens based on predefined rules. It eliminates any unnecessary whitespace and comments while extracting the meaningful tokens.

2.1 Tokenization Process

Figure 1: Working of a Lexer

2.2 Token Types

The following are the token types defined in the HulkPL:

2.2.1 Special Tokens

WhiteSpaceToken
OpenParenToken
CloseParenToken
BadToken
NumberToken
IdentifierToken
StringTypeToken
NumberTypeToken
EOF

2.2.2 Keyword Tokens

VarToken
IfToken
ElseToken
ForToken
WhileToken
DoToken
SwitchToken
CaseToken
BreakToken
ContinueToken
DefaultToken
ReturnToken
TrueToken
FalseToken
NullToken
LetToken
InToken

2.2.3 Operator Tokens

AdditionToken
SubtractionToken

MultiplicationToken
DivisionToken
ModulusToken
EqualityToken
InequalityToken
GreaterThanToken
GreaterThanOrEqualToken
LessThanToken
LessThanOrEqualToken
LogicalAndToken
LogicalOrToken
LogicalNotToken
SemicolonToken
LeftParenthesisToken
RightParenthesisToken
AssignmentToken
FunctionToken
LeftBraceToken
RightBraceToken
PrintToken
QuestionMarkToken
ColonToken
CommaToken
StringToken
TwoDotsToken
AssignmentTwoDotsToken
ArrowToken
ArrobaToken
PowToken

2.3 Implementation

The Lexer class is responsible for tokenizing input text into a list of tokens. It uses regular expressions to match different token patterns. The class contains the following methods:

2.3.1 Lex

```
public static List<Token> Lex(string input)
```

The `Lex` method is the entry point of the lexer. It takes an input string and returns a list of tokens. It iterates through the input string character by character, building potential tokens until a match is found. If no match is found, it creates a token based on the current token string and moves to the next character. Finally, it adds an end-of-file (EOF) token to mark the end of the input.

2.3.2 Match

```
private static Tuple<TokenType, string> Match(string input)
```

The `Match` method is a helper method used by the lexer to match the current token string against the defined regular expressions. It returns a tuple containing the token type and the matched string. It iterates through the token regexes dictionary, attempts to match the input string with each regex, and returns the corresponding token type if a match is found.

2.3.3 GetVariableTypeToken

```
private static TokenType GetVariableTypeToken(string input)
```

The `GetVariableTypeToken` method is a helper method used to map a variable type string (e.g., "String" or "Number") to the corresponding token type. It returns the token type based on the input string.

2.3.4 GetKeywordToken

```
private static TokenType GetKeywordToken(string input)
```

The `GetKeywordToken` method is a helper method used to map a keyword string (e.g., "if", "else", "function") to the corresponding token type. It returns the token type based on the input string.

2.3.5 GetOperatorToken

```
private static TokenType GetOperatorToken(string input)
```

The `GetOperatorToken` method is a helper method used to map an operator string (e.g., "+", "-", "*", "/") to the corresponding token type. It returns the token type based on the input string.

2.4 Lexing Example

Consider the following example...

3 Parser

The parser component of the compiler performs syntax analysis on the tokens produced by the lexer. It checks whether the sequence of tokens conforms to the grammar rules of the programming language. The parser uses a parsing technique such as recursive descent parsing or LALR(1) parsing to build a parse tree or an abstract syntax tree (AST) representing the structure of the program.

3.1 Syntax Analysis

[Explain how the parser analyzes the syntax of the input program.]

3.2 Abstract Syntax Tree

[Discuss the creation and structure of the abstract syntax tree (AST).]

3.3 Implementation

3.3.1 Constructor

```
public Parser(List<Token> tokens)
```

The constructor initializes the fields:

- `tokens` is set to the passed in list of tokens
- `currentTokenIndex` starts at 0
- `precedence` is initialized with the precedence values for each binary operator

3.3.2 Parse

```
public MainProgramNode Parse()
```

The `Parse` method is the main entry point. It parses all the tokens and builds the AST:

- It initializes a list to hold the top level statements
- It calls `ParseStatement` in a loop to parse each statement, adding non-null statements to the list
- It creates the `MainProgramNode` root node with the statement list and returns it

3.3.3 ParseStatement

```
private Node ParseStatement()
```

The `ParseStatement` method parses a single statement based on the current token:

- `PrintStatement` - Parses a print expression statement
- `IfStatement` - Parses an if statement
- `WhileStatement` - Parses a while loop statement
- `FunctionDeclaration` - Parses a function declaration
- `VariableDeclaration` - Parses a variable declaration statement
- `LetStatement` - Parses a let statement
- `Expression` - Parses a standalone expression statement
- `VariableAssignment` - Parses a variable assignment statement

It returns the parsed statement node or null.

3.3.4 ParseBlock

```
private List<Node> ParseBlock()
```

The `ParseBlock` method parses a block statement delimited by . It loops parsing statements until reaching the end brace.

3.3.5 ParseExpression

```
private Node ParseExpression()
```

The `ParseExpression` method is the main entry point for parsing expressions. It delegates to `ParseBinaryExpression`.

3.3.6 ParseBinaryExpression

```
private Node ParseBinaryExpression(int minPrecedence)
```

`ParseBinaryExpression` recursively parses left-hand side expressions and binary operators based on precedence.

3.3.7 ParseUnaryExpression

```
private Node ParseUnaryExpression()
```

`ParseUnaryExpression` checks for unary - and parses the expression.

3.3.8 ParsePrimaryExpression

```
private Node ParsePrimaryExpression()
```

`ParsePrimaryExpression` parses primitive expressions like literals, grouping.

3.3.9 Helper Parsing Methods

Additional helper methods like `Consume`, `Match`, `Check`, `Advance` help with parsing logic like verifying expected tokens and advancing the parser.

4 Conclusion