CIE461

Basic Programming Language Implementation

Phase 1 Report

Project Overview

The project involved the implementation of a programming language for the "automata and compiler design" course. The language was designed to support basic programming constructs like variable declaration, assignment statements, conditional statements, and loops. The primary objective of the project was to implement the lexer and parser phase of the compiler, which would parse the input source code and generate an Abstract Syntax Tree (AST) representation of the program. The AST representation would later be used in the subsequent phases of the compiler, such as semantic analysis, code generation, and optimization.

Tools and Technologies used

For the implementation of the lexer and parser, the project used the Julia programming language, a high-level, high-performance dynamic programming language designed for numerical and scientific computing. Julia was chosen for its ease of use, flexibility, and compatibility with other programming languages. Julia is an open-source language with a large community of developers, making it an ideal choice for the project.

The project also utilized two external packages, PEG and Parsercombinator, to implement the parser. PEG (Parsing Expression Grammar) is a parsing technique that allows for the creation of simple, yet powerful parsers. It is a parsing technique that uses a formal grammar to specify a set of rules that dictate how input data should be parsed. PEG was used in the implementation of the assignment statement, if statement, and while statement.

Parsercombinator, on the other hand, is a Julia package that provides a set of combinators for building parsers. Combinators are functions that take parsers as input and return a new parser as output. The package was used to implement the parsing of mathematical expressions in the programming language.

Finally, the project utilized the enum data type in Julia for the declaration of variables and constants. The enum type in Julia allows for the creation of a set of named values, which are treated as constants. This made it easier to define and manage the different data types used in the programming language.

Tokens and Their description

INT : Integer data type

FLOAT : Float data type

PLUS : Addition operator

MINUS : Subtraction operator

MULTIPLY : Multiply operator

DIVIDE : Divide operator

LPAREN : Left Parenthesis

RPAREN : Right Parenthesis

EQUAL : Equal

EQUALE : Equal equal oprator (==)

ST : Smaller than operator (<)

STE : Smaller than or equal operator (<=)

GT : Greater than operator (>)

GTE : Greater than or equal operator (>=)

IF : If statement

ELSE : Else statement

For : For loop

FUNCTION : function declaration

OR : OR operation for 0s and 1s

AND : And operation for 0s and 1s

LSL : Logical Shift Left

LSR : Logical Shift Right

POWER : Power operation (Exponent)

WHILE : While loop

SWITCH : Switch statement

CASE : Case of the switch

Break : to break the switch

Default : the default case for switch

CURLYLB : Left Curly Brackets

CURLYRB : Right Curly Brackets

EOF : End of File

Quadruples and Their description

Quadruple	Description
“ + ” , V1, V2, Res	Res = V1 + V2
“ - ” , V1, V2, Res	Res = V1 - V2
“ * ” , V1, V2, Res	Res = V1 * V2
“ / ” , V1, V2, Res	Res = V1 / V2
“ - ”, V1, “ ”, Res	Res = - V1

Results

- ## Quadruples

```
julia> expression = "1+2*3/4"
"1+2*3/4"

julia> parsed = parse_one(expression, all)
1-element Vector{Any}:
 Sum(Any[Prd(Any[1.0]), Prd(Any[2.0, 3.0, Inv(4.0)])])

julia> if isempty(parsed)
           error("Invalid expression")
       end

julia> parse_tree = generate_parse_tree(parsed[1])
ParseNode("sum", "", ParseNode[ParseNode("prd", "", ParseNoc
), ParseNode("value", 3.0, ParseNode[]), ParseNode("inv", "'

julia> quadruples = generate_quadruples(parse_tree)
4-element Vector{Quadruple}:
 Quadruple("/", "1", 4.0, "t2")
 Quadruple("*", 2.0, 3.0, "t3")
 Quadruple("*", 3.0, "t2", "t3")
 Quadruple("+", "t1", "t3", "t4")

julia>
```

The Input: 1+2*3/ 4

The outputs: are 4-element Vector{Quadruple}:

Quadruple("/", "1", 4.0, "t2")

Quadruple("*", 2.0, 3.0, "t3")

Quadruple("*", 3.0, "t2", "t3")

Quadruple("+", "t1", "t3", "t4")

• Symbol Table

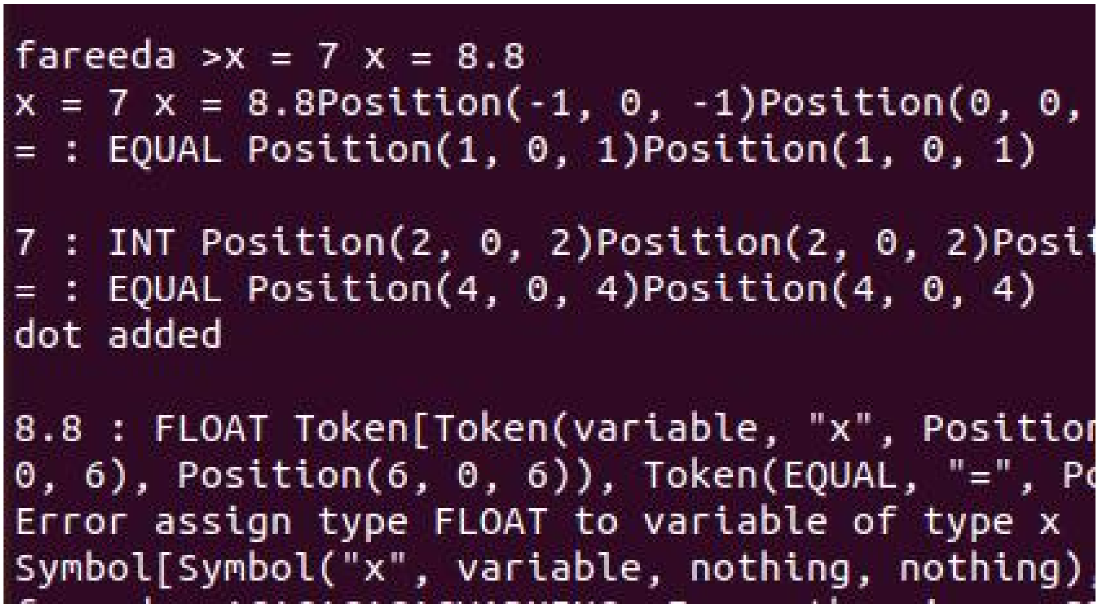
```
fareeda >c {v {g }y }k {o }
c {v {g }y }k {o } Position(-1, 0, -1)Position(0, 0, 0)Position(0, 0, 0)
{ : Left Curly Bracket Position(1, 0, 1)Position(2, 0, 2)Position(2, 0, 2)
{ : Left Curly Bracket Position(3, 0, 3)Position(4, 0, 4)Position(4, 0, 4)
} : Right Curly Bracket Position(5, 0, 5)Position(6, 0, 6)Position(6, 0, 6)
} : Right Curly Bracket Position(7, 0, 7)Position(8, 0, 8)Position(8, 0, 8)
{ : Left Curly Bracket Position(9, 0, 9)Position(10, 0, 10)Position(10, 0, 10)
} : Right Curly Bracket Position(11, 0, 11)Token[Token(variable, "c", Position(11, 0, 11)
on(11, 0, 11), Position(11, 0, 11)), Token(CURLYLB, "{", Position(11, 0, 11), Position(11
, Position(11, 0, 11)), Token(variable, "y", Position(11, 0, 11), Position(11, 0, 11)), T
1, 0, 11)), Token(CURLYLB, "{", Position(11, 0, 11), Position(11, 0, 11)), Token(variable
oken(EOF, "", nothing, nothing)]
///////////////////////////////////////////////
Symbol[Symbol("g", variable, nothing, nothing)]
///////////////////////////////////////////////

///////////////////////////////////////////////
Symbol[Symbol("v", variable, nothing, nothing), Symbol("y", variable, nothing, nothing)]
///////////////////////////////////////////////

///////////////////////////////////////////////
Symbol[Symbol("o", variable, nothing, nothing)]
///////////////////////////////////////////////
Symbol[Symbol("c", variable, nothing, nothing), Symbol("k", variable, nothing, nothing)]
```

The Input: c {v {g} y }k {o}

The outputs: it generated 4 symbol tables one for each scope

1. “g” in symbol table
2. “V ” and “y” in a symbol table
3. “O” in a symbol table
4. “C” and “K” in the global symbol table

- ## Error



The Input: x = 7; x = 8.8

The Output: “Error assign type FLOAT to variable x”

As expected there is an assignment error assigning x to float after assigning it to a type INT