CS 3513 – Programming Languages
Programming Project 01
Report

Group 25:

210119K: Dhanawardhana G.A.C.G

210275H: Karunarathna K.C.A.P.A.

Problem Description:

We were asked to implement a lexical analyzer and a parse for the RPAL language. Output of the parser should have to be abstract syntax tree (AST) for the given program and then we had to implement an algorithm to convert the syntax tree in to standardize tree (ST) and to implement cse machine.

Lexical Analyzer

Overview:

Lexical analyzer is the first phase of the RPAL compiler, responsible for breaking down the input source code into tokens. The lexical analyzer transforms a stream of characters into meaningful units called tokens, facilitating subsequent stages of compilation. RPAL tokens include identifiers, keywords, operators, literals (integers, strings), and delimiters.

Approach:

The lexical analyzer involves iterating through the input program, character by character, while applying lexical rules to identify and classify tokens. The approach taken in the implementation of the RPAL lexical analyzer follows a top-down, recursive strategy, where the input stream is parsed incrementally, and tokens are generated based on predefined patterns and rules.

Implementation:

Token controller class

The TokenController class manages tokens during parsing and implements the singleton pattern.

Instance Management:

Implements the singleton pattern to ensure only one instance exists throughout the application's lifecycle and provides a static method ‘getInstance()’ to access the singleton instance.

Token Storage:

Maintains a vector tokens to store tokens extracted from the input program. The ‘setTokens()’ method populates the tokens vector by retrieving tokens from the lexer until an end-of-file token is encountered.

Lexer Integration:

Stores a pointer to the lexer (CustomLexer) to retrieve tokens from the input program. The ‘setLexer()’ method initializes the lexer and populates the tokens vector.

Token Access:

Provides methods ‘top()’ and ‘pop()’ to access tokens from the tokens vector. ‘top()’ returns a reference to the current token at the top of the tokens vector. ‘pop()’ retrieves and removes the current token from the tokens vector, advancing the current position.

Instance Destruction:

Provides a static method ‘destroyInstance()’ to clear resources associated with the singleton instance.

CustomLexer Class

The CustomLexer class implements the lexical analyzer for the RPAL language.

Tokenization Logic:

Implements the lexical analysis logic to tokenize the input source code. The ‘getNextToken()’ method scans the input string character by character, identifying and categorizing tokens based on language rules.

Token Types:

Recognizes keywords, operators, identifiers, integers, strings, and other token types defined by the RPAL language grammar.

Whitespace Handling:

Skips whitespace characters in the input stream to improve tokenization efficiency.

Operator Symbols:

Defines a method ‘isOperatorSymbol()’ to determine whether a character is an operator symbol.

Parser

Overview:

The parser is the next phase of the compiler or interpreter, responsible for transforming the stream of tokens generated by the lexical analyzer into an abstract syntax tree (AST). In the case of RPAL, the parser analyzes the sequence of tokens to enforce syntactic correctness and build a structured representation of the input program, facilitating subsequent semantic analysis and code generation phases.

Implementation:

Parser Class:

The Parser class encapsulates parsing functionality. It contains a ‘NodeOfStack’ vector to store pointers to nodes during parsing.

‘parse() ‘Function:

This static function initiates the parsing process. It retrieves tokens from the TokenController, a singleton class managing tokens. It handles the end of file condition and sets the root of the abstract syntax tree (AST) when parsing is complete.

Parsing Functions (e.g., E(), Ew(), T(), etc.):

These functions implement the parsing logic for different syntactic constructs defined by the grammar. They manipulate the token stream using methods like ‘pop()’ and handle syntax errors with appropriate error messages.

‘build_tree()’ Function:

This function constructs tree nodes with given labels, number of children, and optional values. It's used by parsing functions to build the AST by popping nodes from the stack and pushing new nodes onto it.

Token Handling:

Tokens are retrieved and manipulated using methods from the TokenController. The parser examines the current token to decide the parsing strategy based on the grammar rules.

Error Handling:

Syntax errors are detected and reported using ‘runtime_error()’ statements with descriptive error messages.

AST Construction:

Nodes are constructed and pushed onto the NodeOfStack vector during parsing. Non-terminal nodes are constructed by popping child nodes from the stack and pushing the parent node onto it.

Abstract Syntax Tree (AST) to Standardized Tree (ST)

Tree.h:

This file contains the declaration of the Tree class, which manages the abstract syntax tree (AST) and the symbol table (ST). It includes functions for setting and getting the root nodes of the AST and ST, as well as for releasing memory associated with them. The ‘generate()’ function initiates the generation of the ST from the AST.

TreeNode.h:

This file defines the ‘TreeNode’ class, representing a node in the tree. It includes functions for adding, removing, and accessing children nodes, as well as for setting and getting the node's label and value. It also provides functionality for memory management, allowing nodes and their children to be recursively released. Additionally, it defines ‘InternalNode’ and ‘LeafNode’ classes as derived classes of TreeNode, representing internal and leaf nodes in the tree, respectively. LeafNode prevents adding children to leaf nodes.

CSE Machine

Overview:

The CSE (Common Subexpression Elimination) machine is a computational model used for optimizing and executing expressions in functional programming languages. It operates by constructing control structures from a provided syntax tree representation of

program code. These control structures encapsulate the expression evaluation process and facilitate efficient execution. The machine iterates through these structures, executing expressions and optimizing performance by identifying and eliminating redundant subexpressions.

Implementation:

CSENode Class:

This class represents nodes in CSE machine. It has several constructors to handle different types of nodes. It stores information such as node type, node value, environment index, control structure index, bound variables, etc.

ControlStructure Class:

This class represents a control structure in CSE machine. It stores a collection of CSENode objects.

Stack Class:

This class represents the stack in CSE machine. It also stores a collection of CSENode objects.

Env Class:

This class represents the environment in CSE machine. It stores variables, lambdas, and lists in unordered maps. It has methods to add and find variables, lambdas, and lists as —findVariable, findLambda, and findList.

CSE Class:

The CSE machine, implemented through the CSE class, operates on a provided TreeNode representation of a program's code to optimize, and execute expressions efficiently. The class features methods tailored for constructing control structures based on the provided syntax tree, crucial for the subsequent evaluation process. Notably, the ‘createCS’ method dynamically generates

control structures by recursively traversing the syntax tree, populating them with appropriate CSENode representations derived from the program's code.
Furthermore, the utility function ‘isOperator’ plays a crucial role in the CSE machine's operation by determining whether a given string label represents an operator.

- Run the Programme

Setting up Environment:
If `make` is not installed, you need to install it. You can either use Chocolatey on Windows or download GNU Make directly.
To install Make using Chocolatey:

choco install make

To download GNU Make: Go to the GNU Make website and download the appropriate version for your system.

Downloading Graphviz:
You can download Graphviz from their official website: graphviz.org/download/
Alternatively, you can use Chocolatey to install Graphviz:

choco install graphviz

Compilation:
Assuming your program is in a directory called `programme`, open a command prompt in that directory.
If `myRpal.exe` doesn't exist:

make

If `myRpal.exe` already exists, no need to run `make`.

Running the Program:

To run an RPAL file:

./myrpal.exe <filename>

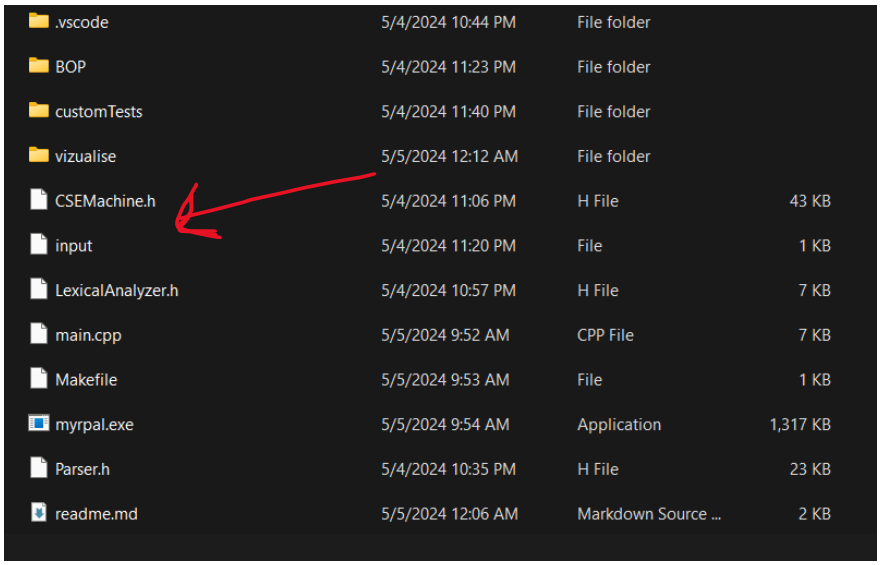
```
PS C:\Users\Gayan\Desktop\210119K-210275H> .\myrpal.exe input
Output of the above program is:
15
```

To visualize the abstract syntax tree (AST):

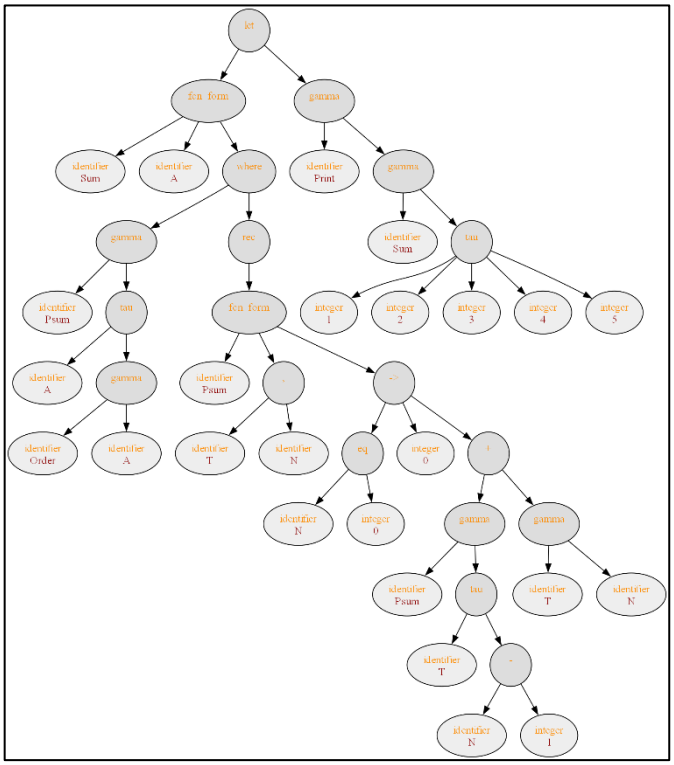
./myrpal.exe <filename> -ast

```
PS C:\Users\Gayan\Desktop\210119K-210275H> .\myrpal.exe input -ast
Abstract Syntax Tree:
let:
    fcn_form:
        identifier: Sum
        identifier: A
        where:
            gamma:
                identifier: Psum
                tau:
                    identifier: A
                    gamma:
                        identifier: Order
                        identifier: A
            rec:
                fcn_form:
                    identifier: Psum
                    ,:
                        identifier: T
                        identifier: N
                    ->:
                        eq:
                            identifier: N
                            integer: 0
                        integer: 0
                        +:
                            gamma:
                                identifier: Psum
                                tau:
                                    identifier: T
                                    -:
                                        identifier: N
                                        integer: 1
                            gamma:
                                identifier: T
                                identifier: N
    gamma:
        identifier: Print
        gamma:
            identifier: Sum
            tau:
                integer: 1
                integer: 2
                integer: 3
                integer: 4
                integer: 5
Also you can find the ast.png in the vizualise directory.
```

This command will generate an AST both in the command line and as `**ast.jpg**` in a directory folder named `visualize`.



open ast.jpg file



These instructions should help you set up the environment and compile/run the program successfully.