Project 1 – CS3513 – Programming Languages
Designing a compiler for RPAL

Group Name → "PL Group"

Group Members →

1) Weerasinghe M.S.S. – 210690B
2) Jayasinghe C.H - 210247B

Introduction & Project Objective →

- This project aimed to develop a compiler for the RPAL language. The primary objectives were to implement a lexical analyzer, a parser to generate an Abstract Syntax Tree (AST) from the input RPAL program, and an algorithm to convert the AST into a Standardized Tree (ST) representation. Additionally, a CSE machine was required to be implemented to execute the RPAL program and produce the desired output.

- The lexical analyzer was responsible for breaking down the input RPAL program into a sequence of tokens, which were then fed into the parser. The parser analyzed the token stream and constructed the corresponding AST based on the RPAL language grammar rules. The AST served as an intermediate representation of the program, capturing its syntactic structure.

- Next, an algorithm was implemented to transform the AST into an ST, a specialized data structure designed for efficient execution. The ST represented the program in a standardized form, facilitating optimizations and code generation.

- Finally, the CSE machine was implemented to execute the ST and generate the correct output for the given RPAL program. The CSE machine performed common subexpression elimination, an optimization technique that avoids redundant computations by reusing previously calculated values.

Main.cpp file →

- The `main.cpp` file serves as the entry point of the RPAL compiler. It is responsible for handling command-line arguments, reading the input file, and initializing the parser component. The main function performs the following tasks →

1) **Argument Parsing**
   The function first checks the number of command-line arguments provided. It expects at least one argument (the input file path) and optionally a second argument `-ast` (flag for printing the Abstract Syntax Tree). The function validates the arguments and extracts the file path and the `-ast` flag (if present).

2) **File Reading**
   After obtaining the file path, the program reads the contents of the input file into a string using the `std::ifstream` class from the C++ standard library. If the file cannot be opened or does not exist, an error message is printed, and the program exits with an error code.

3) **String to Character Array Conversion**
   The contents of the input file, stored in a string, are converted into a character array to be passed to the parser component.

4) **Parser Initialization and Execution**
   An instance of the `parser` class is created, and the character array representing the input program, along with its length and the `-ast` flag, is passed to the constructor. The `parse()` method of the `parser` instance is then called to initiate the parsing process.

5) **Error Handling**
   If an incorrect number of command-line arguments is provided, appropriate error messages are printed, and the program exits with an error code.

The `main.cpp` file serves as the bridge between the command-line interface and the compiler's internal components, ensuring proper argument handling, file reading, and initialization of the parsing process.

Parser.h file →

The `parser.h` file defines the `parser` class, which is responsible for parsing the input RPAL program and constructing the Abstract Syntax Tree (AST) representation. The `parser` class works in conjunction with other components, such as the scanner (lexer), grammar rules, parameters, and data structures for representing the AST and Standardized Tree (ST).

Class Members

1. `isASTFlagged`: A boolean flag indicating whether the AST should be printed or not.

2. `params`: A pointer to a `parameters` object, which holds various parameters and data structures used by the parser.

Constructor

The `parser` constructor takes the following arguments:

1. `characterArray`: A character array containing the input RPAL program.

2. `fileLength`: The length of the input program.

3. `isFlagged`: A boolean indicating whether the AST should be printed or not.

Member Functions

1. `parse()`: This function is the **entry point for the parsing process**. It performs the following tasks:

- Retrieves the first token from the input program using the `scan` function.
- Calls the `procedure_E` function, which starts the recursive descent parsing process from the start symbol of the grammar.
- Checks if the end of the input program is reached by comparing the current index with the file length.
- Prints an error message if the end of the file is not reached.
- Retrieves the root of the AST from the syntax tree stack.
- -If the AST flag is set, it prints the AST by calling the `printTree` function on the root node.
- Calls the `makeStandardizedTree` function to convert the AST into a Standardized Tree (ST) representation.
- Builds control structures for the CSE machine by calling the `buildControlStructures` function.
- Prints the created control structures for the CSE machine.

So this is class is responsible for coordinating the different components of the compiler, such as lexical analysis, syntax analysis, and intermediate code generation (AST and ST). It follows the

grammar rules defined in `grammerRules.h` to parse the input program and construct the corresponding AST and ST representations.

Tree.h file

The `tree.h` file defines a `tree` class that represents a node in the Abstract Syntax Tree (AST). The class has the following members:

Private Members

- `value`: A string representing the value of the current node.
- `type`: A string representing the type of the current node.

Public Members:

- `left`: A pointer to the left child node.
- `right`: A pointer to the right child node.

Member Functions

- `string getType()`: Returns the type of the current node. It doesn't take any parameters.
- `string getValue()`: Returns the value of the current node. It doesn't take any parameters.
- `void setType(string nodeType)`: Sets the type of the current node. It takes a `string` parameter `nodeType` representing the new type.
- `void setValue(string nodeValue)`: Sets the value of the current node. It takes a `string` parameter `nodeValue` representing the new value.
- `static tree* createNode(string value, string type)`: A static function that creates a new tree node with the given value and type. It takes two `string` parameters: `value` representing the node's value, and `type` representing the node's type.
- `static tree* createNode(tree* x)`: A static function that creates a new tree node from an existing node. It takes a `tree*` parameter `x` representing the existing node to be copied.
- `void printTree(int dotsCount)`: Prints the syntax tree by recursively traversing the nodes and indenting based on the depth

The `tree` class represents a single node in the AST, containing the value, type, and pointers to its left and right children. The `createNode` functions are used to create new nodes, and the `printTree` function is used to print the entire AST in a hierarchical format.

standardizeTree.h file

The `standardizeTree.h` file contains a function `makeStandardizedTree` that takes an AST (represented by a `tree` node) and converts it into a Standardized Tree (ST) representation. The ST follows a set of transformation rules defined for the RPAL programming language.

Function:

- `makeStandardizedTree(tree *treeNode)` : This function recursively traverses the AST and applies specific transformation rules based on the node values and structures. The transformation rules are as follows.

```
let[ =[ X E ] P ] ⇒ gamma[ lambda[ X P ] E ]
and[ =++[ X E ]] ⇒ =[ ,[ X++ ] tau[ E++ ]]
where[ P =[ X E ] ] ⇒ gamma[ lambda[ X P ] E ]
within[ =[ X1 E1 ] =[ X2 E2 ] ] ⇒ =[ X2 gamma[ lambda[ X1 E2 ] E1 ]]
rec[ =[ X E ] ] ⇒ =[ X gamma[ Ystar lambda[ X E ] ] ]
fcn_form[ P V+ E ] ⇒ =[ P +lambda[ V .E ]]
lambda[ V++ E ] ⇒ ++lambda[ V .E ]
@[ E1 N E2 ] ⇒ gamma[ gamma[ N E1 ] E2 ]
```

The function recursively applies these transformation rules to the AST, modifying the structure and creating new nodes as needed. The resulting tree represents the ST form of the input program.

Token.h file

The *token.h* file defines the token class, which is used to represent individual tokens in the lexical analysis phase of the compiler. A token is a basic unit of the input source code, such as a keyword, identifier, operator, or literal value.

The token class has the following members:

Private Members:

- type: A string representing the type of the token (e.g., keyword, identifier, operator).
- val: A string representing the actual value of the token (e.g., the text of an identifier or the numerical value of a number).

Public Member Functions:

- setType(const string &sts): Sets the type of the token to the provided string sts.
- setValue(const string &str): Sets the value of the token to the provided string str.
- getType(): Returns the type of the token as a string.
- getValue(): Returns the value of the token as a string.
- operator!=(token t): An overloaded operator function that allows for comparing two `token` objects for inequality.

The `token` class encapsulates the type and value of a token, providing a structured way to represent and manage tokens throughout the different stages of the compiler. The member functions allow for setting and retrieving the type and value of a token object.

By creating an object of the `token` class, the scanner component of the compiler can store the information about each token extracted from the input source code. These token objects can then be passed to later stages of the compiler, such as the parser and code generator, for further processing.

CseEnvironment.h file

The `cseEnvironment.h` file plays a crucial role in managing variable scopes and environments within the context of the programming language implementation. It defines the `environment` class, which is responsible for creating and maintaining environments that store variables and their corresponding values.

Class Members

- `string envName` : This member variable stores the name of the current environment. It can be used for identification purposes or for debugging and logging.

- `environment *prevEnv` : This pointer member variable holds a reference to the previous environment. It allows for a hierarchical structure of environments, where each nested scope has access to the variables defined in its parent scope.

- `map<tree*, vector<tree*>> variablesMap` : This is a map data structure that associates variables (represented as trees) with their corresponding values The use of trees as the data structure for variables and values

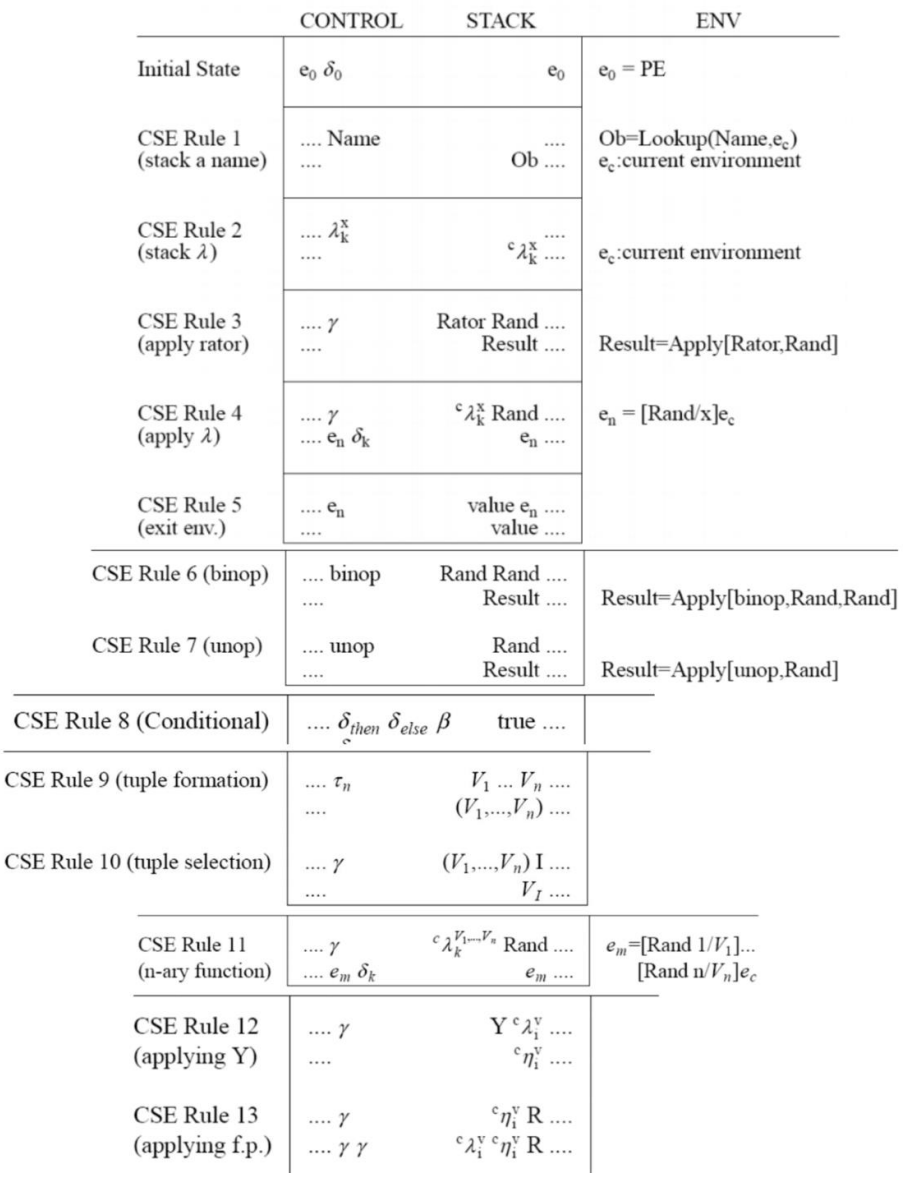
Constructors

- `environment()` : The default constructor initializes a new environment with a `prevEnv` set to `NULL` (indicating no parent environment) and an `envName` of `"env0"`. This constructor is likely used for creating the global or top-level environment.

- `environment(environment *prev, string name)` : The parameterized constructor takes a pointer to the previous environment (`prev`) and a string representing the name of the new environment (`name`). This constructor is useful for creating nested environments within parent environments, such as local scopes within functions or blocks..

Appendix

1. CSE Machine Rules



2. RPAL Grammer

```
# Expressions ########################################

E    -> 'let' D 'in' E                          => 'let'
     -> 'fn'  Vb+ '.' E                         => 'lambda'
     ->  Ew;
Ew   -> T  'where' Dr                           => 'where'
     -> T;

# Tuple Expressions ##################################

T    -> Ta ( ',' Ta )+                          => 'tau'
     -> Ta ;
Ta   -> Ta 'aug' Tc                             => 'aug'
     -> Tc ;
Tc   -> B '->' Tc '|' Tc                        => '->'
     -> B ;

# Boolean Expressions ################################

B    -> B 'or' Bt                               => 'or'
     -> Bt ;
Bt   -> Bt '&' Bs                               => '&'
     -> Bs ;
Bs   -> 'not' Bp                                => 'not'
     -> Bp ;
Bp   -> A ('gr' | '>' ) A                       => 'gr'
     -> A ('ge' | '>=') A                       => 'ge'
     -> A ('ls' | '<' ) A                       => 'ls'
     -> A ('le' | '<=') A                       => 'le'
     -> A 'eq' A                                => 'eq'
     -> A 'ne' A                                => 'ne'
     -> A ;

# Arithmetic Expressions #############################

A    -> A '+' At                                => '+'
     -> A '-' At                                => '-'
     ->   '+' At
     ->   '-' At                                => 'neg'
     -> At ;
At   -> At '*' Af                               => '*'
     -> At '/' Af                               => '/'
     -> Af ;
Af   -> Ap '**' Af                              => '**'
     -> Ap ;
Ap   -> Ap '@' '<IDENTIFIER>' R                 => '@'
     -> R ;

# Rators And Rands ###################################

R    -> R Rn                                    => 'gamma'
     -> Rn ;
Rn   -> '<IDENTIFIER>'
     -> '<INTEGER>'
     -> '<STRING>'
     -> 'true'                                  => 'true'
     -> 'false'                                 => 'false'
     -> 'nil'                                   => 'nil'
     -> '(' E ')'
     -> 'dummy'                                 => 'dummy' ;

# Definitions ########################################

D    -> Da 'within' D                           => 'within'
     -> Da ;
Da   -> Dr ( 'and' Dr )+                        => 'and'
     -> Dr ;
Dr   -> 'rec' Db                                => 'rec'
     -> Db ;
Db   -> Vl '=' E                                => '='
     -> '<IDENTIFIER>' Vb+ '=' E                => 'fcn_form'
     -> '(' D ')' ;

# Variables ##########################################

Vb   -> '<IDENTIFIER>'
     -> '(' Vl ')'
     -> '(' ')'                                 => '()';
Vl   -> '<IDENTIFIER>' list ','                 => ','?;
```