CS 3513 - Programming Languages
RPAL interpreter

Group Members:

- 210372D Manawathilake K.C.K.
- 210067X Bandara.W.G.W.M.

Problem Description:

The project requirement was to implement a compiler for the rpal language. The grammar and the lexical rules are already provided to us. We had to implement the scanner, screener, parser, a tree standardizer and finally the cse machine to complete the compiler. After the completion, the program is able to read a file, compile it and execute to give a suitable output. For this project the compiler was coded in C plus plus programming language.

Structure of the Program

Myrpal.cpp is the main file that coordinates all the functions and data structures in the programmer. Below are the main functions called within this file

- readFileToString(string)
- scanner(string)
- screener()
- parser()
- standardizer()
- cse()

These functions are coded in several different files. Below is a description of all the files and their content

- Struct.h : Contains the data structures for the nodes of the abstract syntax tree, data structure for the tokens of the lexical analyser, suitable arrays and stacks to store the created derivation trees and the syntax tree and the print functions
- Vocabulary.h : Contains the functions that can identify the alphabet categories
- Lex.h : Contains the functions of the lexical analyzer, which includes the scanner and the screener.
- Grammar.h : Contains the grammar rules of the language.
- Standardize.h : Contains the functions needed to standardize the AST.
- Cse_structs.h : This contains the data structure of the building units of the cse machine. This also includes the stacks required to implement the cse machine, build in functions and other auxiliary functions required.
- Cse.h : Contains the main content of the Control Stack Environment Machine.

Process

In order to run a program in our compiler we first need to create a text file with the required program written in it. This file should be saved in the same directory as the compiler (myrpal.exe). The programme can be then compiled using the following terminal commands:

./myrpal file_name

This would run the program. And if it is required to obtain the AST of the program, an -ast flag should be added to the end of this command:

./myrpal file_name -ast

Following of this document will explain how a text file provided to this programme will be compiled step by step

Step 1: Reading the String from the text file

The file is read using the readFileToString() function in myrpal.cpp
The file is read using the fstream cpp library. Using this library the file location of the given file_name is read. If there are any errors it will be output in the terminal. If the reading process was successful the input file will be stored as the variable string input in the rpal.cpp file

Step 2: Lexical analyzer

This consists of the scanner and the screener (lex.h file). First the scanner will be called with the input string as its argument.

This scanner will then iterate through each character in the string with a pointer (named int index). It will also maintain a buffer to store each word that is being read until the word reaches an end. The suitable category is then identified using several functions (ie. isDigit(), isOperator(), …) that are implemented in the vocabulary.h file. These help identify if each character is a letter, digit, space operator or a punctuation. Depending on the output of those functions the strings (which are now stored in the buffer variable) are categorized as identifiers, integers, string, space, comment, operator and are stored in an array as Token data structures. At the end of the file an “EOF” token is added to identify the end of the token sequence.

Token data structure:

```
typedef struct Token // Scanned tokens
{
  string type;
  string value;

  Token(string type, string value) : type(type), value(value) {}
} Token;
```

The array of tokens created are stored in the global variable vector<Token> tokens; This variable is accessible by any function in the programme. So in the screener, we iterate through this array and remove any unnecessary tokens such as comments and space tokens. Once the EOF is reached we terminate this process and move back to the main function.

Step 3: Parser

The main file then calls the parser function to parse the program. The purpose of this function is to create the Abstract Syntax Tree based on the grammar provided. The created Abstract Syntax Tree is stored in the stack ast_bu. Since this is built in the bottom up manner we will only need to access the top of the stack and at the end of Parsing the top item of the stack will be the root of the AST. A node of the AST is represented by the Node data structure. Following is the basic structure of it.

```
typedef struct Node // Building block of abstract syntax tree
{
  string token;
  vector<Node *> children;

  Node(string token) : token(token), children() {}

  ~Node()···
} Node;
```

There are three useful functions used in the parser.

1. Read(string type, string value)
   This function reads the next available token and checks if they match the provided arguments of type and value. If they do not match up it will throw an Error. If not it will read that token and move to the next command. If the token is an identifier, integer or a string it will create a node in the AST. If the value is not provided it will Read any token of the given type.

2. Bool NextToken(string type, string value)
   This function will return true or false if the next token available matches the given description. If no value is specified, this will return true only for a type match. An exception will be thrown in an error.

3. Build_tree(string token, int arguments) in the struct.h file
   This function will pop the given number of Nodes (int arguments) from the ast_bu stack and will add them to a newly created Node as children. The name (attribute token) of the new node will be the string token parameter in this function. This newly created Node will then be added to the top of the ast_bu stack.

When the parser function is called it will propagate through the tokens and create a suitable AST based on the rules. If there are any syntax errors it will immediately throw an error.

Apart from the AST we also build a bottom up derivation tree for debugging purposes which is only accessible through the program source code.

Step 4: Standardizer

The AST created is standardized using the standardizer() function. The logical operations required for this are stored in the standize.h file.

This function standardizers the following nodes:

- Let
- Where
- Within
- Rec
- Fcn_form
- And
- @
- Lambda (with more than two children)

After standardizing, the AST is now in a suitable format that can be used in the CSE Machine.

Step 5: CSE Machine

Main logic of the CSE machine in the cse.h file. Additional data structures and functions required to implement that are stored in the cse_structs.h file.
The entire CSE Machine works upon the custom data structure name Base. This structure is used to store and manipulate all kinds of data within the CSE. Below is the structure of the Base data type:

```
class Base // Each node in the control stack and the stack stack
{
public:
  string type = "none";

  string arg_str = "";
  int arg_int;

  Base *prev = nullptr;
  vector<Base *> children = {};

  // Constructors
  Base(string type) : type(type){};
  Base(string type, int arg_int) : type(type), arg_int(arg_int){};    // Eg: integer
  Base(string type, string arg_str) : type(type), arg_str(arg_str){}; // Eg: identifier
  Base(string type, string arg_str, int arg_int) : type(type), arg_str(arg_str), arg_int(arg_int){};
  Base(string type, string arg_str, Base *prev) : type(type), arg_str(arg_str), prev(prev){};
  Base(string type, Base *prev) : type(type), prev(prev){};

  // Destructor
  ~Base() ···
};
```

There are five arguments and each will serve different purposes depending on the type of the Base (type of the structure in the CSE environment) Following table shows what each attribute stores depending on their type

type	**arg_str**	**arg_int**	***prev**	**children**
string	value	-	-	-
tuple	-	-	-	elements
integer	-	value	-	-
boolean	true/false	-	-	-
function (lambda/eta)	Function name	Stored index	Called environment	Function parameters
dummy	-	-	-	-
delta	-	index	-	-
gamma	-	-	-	-
environment	-	-	Parent environment	-
beta	-	-	-	-
tau	-	size	-	-
ystar	-	-	-	-

In addition to this data structures 3 separate stacks and a vector array is used to store information:

1. stack<Base*> control_stk : Working control stack
2. stack<Base*> stack_stk : Working stack stack
3. stack<Base*> parsing_env : Stored environments
4. vector<vector<Base *>> control_structures: Stores each function read from the syntax tree.

Additionally there are some auxiliary functions that are used for the cse machine. These are basic functions and their purpose is identifiable through their names.

CSE Process

Initially we iterate through the AST and create suitable Base data structures for each Node in the tree. Here we normally iterate in a preorder traversal except for a few exceptions. While iterating through the AST we add each new function to the control_structures. Since the control_structures is an array, we can identify each function based on their index. Therefore the main function of our rpal program will always be on the 0th index. Apart from that the inbuilt functions use negative indices. They are not stored in control_structures but are called upon using the negative indices.

After creating all the “Base” structures for the programme the cse function will create a global environment in the parsing_env stack. All global variables and identifiers for inbuilt functions are defined here. Then starting from the main function each new function will create a new environment variable in this stack and will remove it from the stack upon leaving the function. Once a function is called the function “Base” structures are added to the control_stk using the add_func_to_control() function. The CSE will iterate through the control_stk indefinitely until it reaches the main function environment “Base”. For each item in the contorl_stk it will perform suitable actions based as stated in the rule() function.

All these functions check for exceptions and throw an error message if there are any issues regarding them.

At the end of the CSE Machine the programme will return to the main function and it will call the clear_stacks() function to clear up all the memory stacks to avoid memory leaks.

Team Contributions

Manawathilake K.C.K. : CSE machine, Standardizer, Document
Bandara.W.G.W.M. : Lexical analyzer, Parser, Document