Programming Languages Project

Group - 17

Group Members

MARASINGHE M.M.R.S. 200382A

PATHIRANA A.S.W. 200448H

SAMARAKOON R.S.A. 200555H

SATHIYENDRA T.J. 200590J

Problem

You are required to implement a lexical analyzer and a parser for the RPAL language. Refer to RPARPAL_Lex.pdf for the lexical rules and RPAL_Grammar.pdf for the grammar details. Should not use 'lex'. 'yacc' or any such tool. The Output of the parser should be the Abstract Syntax Tree (AST) for the given input program. Then you need to implement an algorithm to convert the Abstract Syntax Tree (AST) into Standardize Tree (ST) and implement CSE machine. Your program should be able to read an input file that contains a RPAL program. The output of your program should match the output of "rpal.exe" for the relevant program. You must use C/C++ or Java for this project.

Run Program

run: make

```
PS D:\University\Academics_2020_4th_Sem\7. Programming Languages\group_p
roject> .\rpal20.exe .\TestCases\add.rpal
15
PS D:\University\Academics_2020_4th_Sem\7. Programming Languages\group_p
roject>
```

Solution

1. Read the input from file
2. Tokenizing the input
3. Defining the grammar and building the AST
4. Standardizing the AST
5. From the ST evaluating the program using CSE machine

Read the input from file

In the command line, we must specify the interpreter's path followed by a space and the file name where the input should be specified. The tokenizer will get the file path from the command line. Using the file path, it will open the specified file and read the input.

Tokenizing the input

Next in tokenizing we read the input and return a vector containing the tokens of the input string separated by whitespace, treating the comments and string as special cases.

First, we created the class tokenize to read the input and return the set of tokens. We implemented functions within the class as follows to do the task.

1. Read the input file from Test_cases folder

- First, we should read the content from a file and then pass the obtained input through the tokenizer to break it down into meaningful tokens.
- By implementing the "read_file" function we read the content of the file we selected by removing the whitespaces or the tabs at the beginning of each line.
- Then stored the content as a single string while handling file errors.
- Finally, the read input string is passed to the be broken to meaningful tokens.

Tokenize the input string by white spaces while handling comments and string as special cases using the lexer function.

```
void lexer()
 {
    // Temporary storage for building a token
    string currentToken = "";

    // State variables to keep track of special cases
    bool isComment = false;
    bool isString = false;
    char stringDelimiter;
    // Loop through each character in the input string
    for (int i=0; i<input.length(); i++)
    {
        char ch = input[i];
        // Check if we are inside a comment
        if (isComment)
        {
            if (ch == '\n')
            {
                // If a newline is encountered, the comment ends
                isComment = false;
            }
            continue;
        }

        // Check if we are inside a string
        if (isString)
        {
            if (ch == stringDelimiter)
            {
                isString = false;
            }
            currentToken += ch;
            continue;
        }

        // Check if the current character is a whitespace
        if (isspace(ch))
        {
            if (!currentToken.empty())
            {
                this->push_token(currentToken);
                currentToken = ""; // Reset the temporary storage
```

```
            }
        }
        // Search through the operator_symbols and if it matches
        else if (find(this->operator_symbols.begin(), this->
           operator_symbols.end(), ch) != this->operator_symbols.end())
        {
            // If we have any other token in the buffer
            // Push it to the tokens vector
            if (isString)
            {
                currentToken += ch;
                continue;
            }
            if (currentToken != "")
            {
                this->push_token(currentToken);
                currentToken = "";
            }
            currentToken += ch;

            if (currentToken == "-" && ((i + 1) < this->input.length())
                                      && (this->input[i + 1] == '>'))
            {
                currentToken += this->input[i + 1];
                i++;
            }
            else if (currentToken == "/" && ((i + 1) < this->
                     input.length()) && (this->input[i + 1] == '/'))
            {
                isComment = true;
                currentToken = "";
                continue;
            }
            // Now the currentToken is an operator symbol
            // Push it to the tokens vector
            this->push_token(currentToken);
            currentToken = "";
        }
        // Check if the current character starts a string
        else if (ch == '\'')
        {
            isString = true;       // Set the state to inside string
            stringDelimiter = ch; // Store the string delimiter
            currentToken += ch;
        }

        else
        {
            currentToken += ch;
        }
    }

    // Add the last token (if any) after the loop finishes
    if (!currentToken.empty())
    {
        this->push_token(currentToken);
```

```
        }
    }
```

Using the following function of validate_tokens we validate and classifies the tokens in the vector of tokens. After determing the type of the tokens the identified tokens are stored in the vector called 'lex'

```
    void validate_tokens() {
        regex identifier_pattern("[a-zA-Z][a-zA-Z0-9_]*");
        regex integer_pattern("[0-9]+");
        regex op_pattern(R"([+\-*<>&.@/:=~|$!#%^_\[\]{}"`?]+)");
        regex string_pattern("\'[^\"]*\'");
        regex space_pattern(R"(\s+)");

        int i = 0;
        while (i < this->tokens.size()) {
            string token = this->tokens[i];
            i++;

            if (regex_match(token, identifier_pattern)) {
                this->lex.emplace_back("identifier");
            } else if (regex_match(token, integer_pattern)) {
                this->lex.emplace_back("integer");
            } else if (regex_match(token, op_pattern)) {
                this->lex.emplace_back("operator");
            } else if (regex_match(token, string_pattern)) {
                this->lex.emplace_back("string");
            } else if (token == "(") {
                this->lex.emplace_back("(");
            } else if (token == ")") {
                this->lex.emplace_back(")");
            } else if (token == ";") {
                this->lex.emplace_back(";");
            } else if (token == ",") {
                this->lex.emplace_back(",");
            } else if (regex_match(token, space_pattern)) {
                continue;
            }
            else
            {
                std::cout << "Invalid Token" << endl;
                std::exit(0);
            }
        }
    }
```

Defining the grammar and building the AST

To create AST, we should first implement the tree node. Children of the node are stored in a vector.

```
struct Node
{
    string data;
    string type = "internal node";
    vector<Node*> children;
};
```

Then, class Graph is implemented which provides methods to create AST.

```
class Graph
{
public:
    Graph(){
        // Initialize the poiter to the root node
        this->root = new Node();
    }

    Node* get_root(){
        return this->root;
    }

    void build_Tree(string data, long long pop_tree_cnt){
        if (this->node_stack.size() == 0)
        {
            this->root->data = data;
            this->node_stack.push_back(this->root);
            // this->push_to_stack(data);
            return;
        }


           if (pop_tree_cnt == 0)
        {
            this->push_to_stack(data);
            return;
        }
        Node* node = new Node();
        node->data = data;


           for (int i = 0; i < pop_tree_cnt; i++)
        {
            Node* cur_node = this->node_stack.back();
```

```
            this->node_stack.pop_back();
            node->children.push_back(cur_node);
        }


          // reverse the children
        reverse(node->children.begin(), node->children.end());

        this->root = node;
        this->node_stack.push_back(node);
    }

    void push_to_stack(string node_name, string node_type = 
"internal node"){
        Node* node = new Node();
        node->data = node_name;
        node->type = node_type;
        this->node_stack.push_back(node);
    }

private:
    vector<string> graph;
    Node* root;
    vector<Node*> node_stack;

    void addNode(Node* node, Node* parent)
    {
        parent->children.push_back(node);
    }

    void inorder(Node* node){
        if (node->children.size() == 0)
        {
            std::cout << node->data << endl;
            return;

          }
        for (int i = 0; i < node->children.size(); i++)
        {
            inorder(node->children[i]);
        }
    }
};
```

In the above code, nodes are stored in the vector node_stack. When build_Tree method is called, a new node is created with the data passed through the first parameter. Then the number of nodes given as the second parameter are popped from the node_stack and they are attached to the newly created node as children nodes. Then it's pushed back to the node_stack.

To implement grammar, we implemented the class, 'Grammar'. Here, we check for the valid grammar rules and build the AST accordingly. An error is thrown if the grammar rules are invalid.

```
class Grammar
{
public:
    Grammar(vector<string> grammar)
    {
        this->grammar = grammar;
        this->token = grammar[0];
        this->AST = Graph();
    }

    Node* get_ast_root()
    {
        return this->AST.get_root();
    }

    void parse()
    {
        E();
    }

private:
    vector<string> grammar;
    string token;
    Graph AST;

    // Have to check for the keywords that are used for error
handling
    // For example: Isinteger, Isstring
    vector<string> keywords = {"let", "in", "fn", "where", "aug",

                               "or", "not", "gr", "ge", "ls", "le",
"eq", "ne","true", "false", "nil", "dummy", "within", "and","rec"};

    bool CheckType(string token, string str_type)
    {
        if (str_type.compare("identifier") == 0)
        {
            // We don't want to match a keyword as an identifier
            if (std::find(keywords.begin(), keywords.end(), token)
!= keywords.end())
                {
                return false;
            }
            regex identifier_pat("[a-zA-Z][a-zA-Z0-9_]*");
            return regex_match(token, identifier_pat);
        }
        else if (str_type.compare("integer") == 0)
```

```
        {
            regex integer_pat("[0-9]+");
            return regex_match(token, integer_pat);
        }
        else if (str_type.compare("string") == 0)
        {
            regex string_pat("\'[^\"]*\'");
            return regex_match(token, string_pat);
        }
        return false;
    }

    /**
     * @brief This function will take the token which should be
parsed in the grammar & consume it.
     *
     * @param token
     */
    void CONSUME(string token)
    {
        if (CheckType(token, "identifier")){
            this->AST.push_to_stack(token, "identifier");
        }

          else if (CheckType(token, "string")){
            this->AST.push_to_stack(token, "string");
        }
        else if (CheckType(token, "integer")){
            this->AST.push_to_stack(token, "integer");
        }


         this->grammar.erase(this->grammar.begin());
        // Update the token

          if (this->grammar.size() > 0)
        {
            this->token = this->grammar[0];
        }
        else
        {
            this->token = "";
        }

    }
```

The grammar rules are defined using the RPAL grammar rules. As an example,

```
Tc    -> B '->' Tc '|' Tc                    => '->'
      -> B ;
```

The above grammar rule is implemented as,

```
void Tc()
{
    B();
    if ((this->token.compare("->")) == 0)
    {
        CONSUME("->");
;
        Tc();

        if ((this->token.compare("|")) == 0)
            CONSUME("|");
        else
            throw std::runtime_error("Can't parse the given input!
\'|\' is missing");

        Tc();

        this->AST.build_Tree("->", 3);
    }
}
```

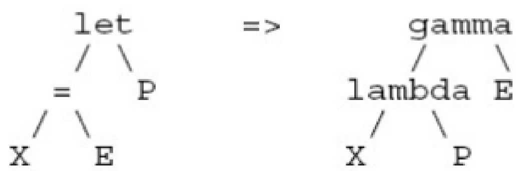
Standardizing the AST

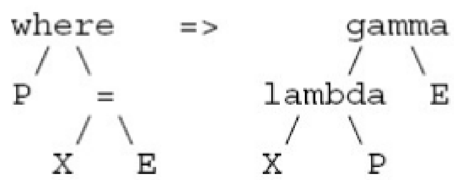
The AST is implemented using the Node pointers. Hence, we only need to keep track of the root node. We standardized the tree in a bottom-up manner. We traversed the tree until we hit a child and we standardized starting from the child. The code for traversing is as follows.

```
void standardize(){
    construct_ST(this->st_root);
}

void construct_ST(Node* node){
     for (int i = 0; i < node->children.size(); i++){
         construct_ST(node->children[i]);
     }
rec_standardize_AST(node);
}
```

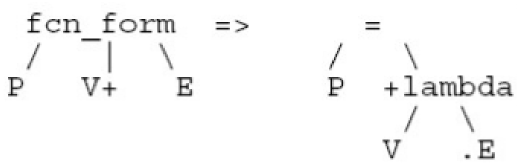
Then using the standardizing rules, let node, where node, fcn_form node, within node, and node, @ node, and rec node were standardized. The rules for standardization are as follows.

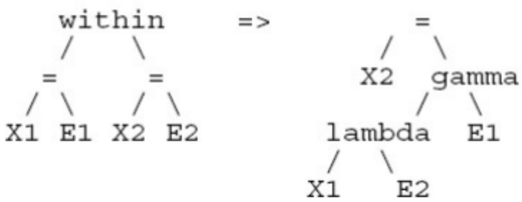




Let standardization

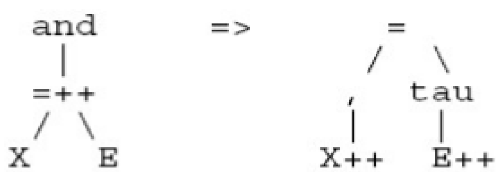
Where standardization

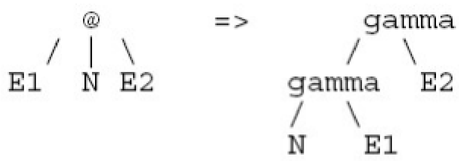




fcn_form standardization

within standardization





and standardization

@ standardization

```
rec    =>     =
 |           / \
 =          X   gamma
/ \              /  \
```

rec standardization

```
  / \
 X   E
```

Using these standardization rules the nodes were standardized. Finally, the root node of the ST is returned to solve the CSE machine.

From the ST evaluating the program using CSE machine

Initializing the control structure variable

In the control structure we need to store strings, integers, tuples, environment variables, functions, operators, conditional variables, dummy variables, lambda variables, eta variables, and tau variables. So, we created a structure to hold all these variables.

```
struct env_var{
    int env_num;
};
struct lambda_var{
    vector<string> bdd_vars;
    int bdd_var_cnt = 0;

    int ctrl_no;
    int env_no;
};
struct eta_var{
    vector<string> bdd_vars;
    int bdd_var_cnt;

    int ctrl_no;
    int env_no;

};

struct tau_var{
    int tau_no;
};
struct cond_var{
    int if_cond_ctrl_no;

     int else_cond_ctrl_no;
};

struct binop_var{
    string op;
```

```
};

struct unop_var{
    string op;
};

struct ctrl_and_stack_var{
    bool is_env = false;
    bool is_lambda_var = false;
    bool is_eta_var = false;
    bool is_tau_var = false;
    bool is_cond_var = false;
    bool is_binop_var = false;
    bool is_unop_var = false;
    bool is_name_var = false;
    bool is_bool_var = false;
    bool is_tuple_var = false;
    bool is_func_var = false;
    bool is_int_var = false;
    bool is_string_var = false;
    bool is_dummy_var = false;

    string name_var;
    bool bool_var;
    env_var env;
    lambda_var lambda;
    eta_var eta;
    tau_var tau;

     cond_var cond;
    binop_var binop;
    unop_var unop;
    vector<ctrl_and_stack_var> tuple_elements;
    string func_name;
    int int_var;
    string string_var;
    string dummy;

    int conc_func_cnt = 2;
};
```

Find the control structures from the ST

We used pre-order traversal on the Standardized tree to find the control structures.

```
void initialize_control_structures(){
            q.push(this->st_root);
            while (!q.empty()){
                Node* cur_node = q.front();
                q.pop();
                this->current_control_structure.clear();
                pre_order(cur_node);
                this->control_structures.push_back
               (current_control_structure);
            }
}
```

Initializing the environment tree

The environment tree is initialized using vectors. When the control is going into separate environments, we need to keep track of the variables corresponding to the environment. The environment tree is initialized as follows.

```
// Maintain the environment tree
// first -> cur_env
// second -> parent_env
map<int, int> env_tree;
int env_number = 0;
// Maintain the environment variables in a map
map<int, vector<string>> env_vars;
map<int, vector<ctrl_and_stack_var>> env_vals;
int cur_env_number = 0;
map<int, int> prev_env;
```

Evaluate using the implemented rules

After initializing the control structures, the control and the stack are initialized. While the control is not empty, we checked for the applicable rule and applied it. For the rule number 3, we need to apply the function and pass the parameters. We defined a separate function to evaluate the function and return the results. The rules are implemented as follows.

```
void use_rule_1(){
            ctrl_and_stack_var cur_ctrl_var = this->control.back();
            this->control.pop_back();

            if (cur_ctrl_var.is_int_var || cur_ctrl_var.is_bool_var ||
                cur_ctrl_var.is_string_var || cur_ctrl_var.is_func_var ||
                 cur_ctrl_var.is_dummy_var ||
                (cur_ctrl_var.is_name_var && (cur_ctrl_var.name_var == "Y*"
                 || cur_ctrl_var.name_var == "nil"))){
                this->stack.push_back(cur_ctrl_var);
                return;
            }
            else{
                int cur_env_no = this->cur_env_number;
                while (cur_env_no != -1){
```

```
                    for (int i=0; i< this->env_vars[cur_env_no].size();
i++){
                        if (this->env_vars[cur_env_no][i] ==

cur_ctrl_var.name_var){
                            this->stack.push_back(this->
                                env_vals[cur_env_no][i]);
                            return;
                        }
                    }

                    cur_env_no = this->env_tree[cur_env_no];
                }
                int val = cur_ctrl_var.name_var[0];
                std::cout << "Variable not found: " <<cur_ctrl_var.name_var
<< endl;
                std::exit(0);
            }


        }

        void use_rule_2(){
            ctrl_and_stack_var cur_ctrl_var = this->control.back();
            this->control.pop_back();
            cur_ctrl_var.lambda.env_no = this->env_number;
            this->stack.push_back(cur_ctrl_var);
        }




        void use_rule_3(){
            ctrl_and_stack_var gamma = this->control.back();
            this->control.pop_back();

            ctrl_and_stack_var rator = this->stack.back();
            this->stack.pop_back();

            ctrl_and_stack_var rand = this->stack.back();
            this->stack.pop_back();

            ctrl_and_stack_var new_ctrl_var;

            new_ctrl_var = apply_functions(rator, rand);

            if (rator.func_name != "Print" )
                this->stack.push_back(new_ctrl_var);
            else{
                ctrl_and_stack_var dummy_ctrl;
                dummy_ctrl.is_dummy_var = true;
                dummy_ctrl.dummy = "dummy";
                this->stack.push_back(dummy_ctrl);
            }
        }
```

```
void use_rule_4(bool if_use_11 = false){
    // Remove Gamma node from control
    this->control.pop_back();

    // Get the lambda node from stack
    ctrl_and_stack_var lambda = this->stack.back();
    this->stack.pop_back();

    // Link the child with the parent env
    this->env_tree[this->env_number+1] = lambda.lambda.env_no;
    // Link the previous environmnet
    this->prev_env[this->env_number+1] = this->cur_env_number;

    if (! if_use_11){
        ctrl_and_stack_var rand = this->stack.back();
        this->stack.pop_back();

        this->env_vars[this-> env_number + 1 ].push_back(
                                     lambda.lambda.bdd_vars[0]);
        this->env_vals[this->env_number+1].push_back(rand);
    }
    else{
        ctrl_and_stack_var rand = this->stack.back();
        this->stack.pop_back();

        if (! rand.is_tuple_var){
            std::cout << "Expected tuple variable" << endl;
            std::exit(0);
        }

        for (int i=0; i<lambda.lambda.bdd_var_cnt; i++){
            this->env_vars[this->env_number+1].push_back
                                     (lambda.lambda.bdd_vars[i]);
            this->env_vals[this->env_number+1].push_back
                                     (rand.tuple_elements[i]);
        }
    }

    // Create new environment and push it to control and stack
    ctrl_and_stack_var new_env;
    new_env.is_env = true;
    new_env.env.env_num = this->env_number+1;
    this->env_number++;

    this->cur_env_number = this->env_number;

    this->control.push_back(new_env);
    this->stack.push_back(new_env);

    // Get the ctrl number of lambda node
    int ctrl_num = lambda.lambda.ctrl_no;
    // Push the elements of the bounded env to control
    for (int i=0; i < this->control_structures[ctrl_num].size();
                                           i++)
```

```
            {
           this->control.push_back(this->control_structures[ctrl_num][i] );
            }

        }

        void use_rule_5(){
            ctrl_and_stack_var cur_ctrl_var = this->control.back();
            this->control.pop_back();

            int env_to_disable = cur_ctrl_var.env.env_num;
            this->cur_env_number = this->prev_env[env_to_disable];

            for (int i = this->stack.size()-1; i >= 0 ; i--){
                if (this->stack[i].is_env && (this->stack[i].env.env_num ==

env_to_disable)){
                    this->stack.erase(this->stack.begin() + i);
                    break;
                }
            }
        }

        void use_rule_6(){
            string bin_op_to_apply = this->control.back().binop.op;
            this->control.pop_back();

            ctrl_and_stack_var rand1 = this->stack.back();
            this->stack.pop_back();
            ctrl_and_stack_var rand2 = this->stack.back();
            this->stack.pop_back();

            ctrl_and_stack_var new_ctrl_var = solve_binary_ops
                                    (bin_op_to_apply, rand1, rand2);
            this->stack.push_back(new_ctrl_var);
        }

        void use_rule_7(){
            string un_op_to_apply = this->control.back().unop.op;
            this->control.pop_back();

            ctrl_and_stack_var rand = this->stack.back();
            this->stack.pop_back();

            ctrl_and_stack_var new_ctrl_var =
                        solve_unary_ops(un_op_to_apply, rand);
            this->stack.push_back(new_ctrl_var);
        }

        void use_rule_8(){
            ctrl_and_stack_var cur_ctrl_var = this->control.back();
            this->control.pop_back();

            ctrl_and_stack_var bool_stack_content = this->stack.back();
            this->stack.pop_back();
```

```
        int if_ctrl_num = cur_ctrl_var.cond.if_cond_ctrl_no;
        int else_ctrl_num = cur_ctrl_var.cond.else_cond_ctrl_no;

        int ctrl_to_push;

        if (bool_stack_content.bool_var){
            ctrl_to_push = if_ctrl_num;
        }
        else{
            ctrl_to_push = else_ctrl_num;
        }

        for (int i=0; i<this->control_structures[ctrl_to_push].size();
                                                                i++){
            this->control.push_back
                          (this->control_structures[ctrl_to_push][i]);
        }
    }

    void use_rule_9(){
        int tau_cnt = this->control.back().tau.tau_no;
        this->control.pop_back();

        ctrl_and_stack_var new_ctrl_var;
        new_ctrl_var.is_tuple_var = true;

        for (int i=0; i<tau_cnt; i++){
            ctrl_and_stack_var rand = this->stack.back();
            this->stack.pop_back();
            if (! (rand.is_tuple_var || rand.is_int_var ||
                   rand.is_string_var || rand.is_bool_var)){
                if (rand.is_name_var && rand.name_var == "nil"){
                    // Do nothing
                }
                else{
                    std::cout << "Error: Tuple element is not a tuple,
                                  int, string or bool" << std::endl;
                    exit(1);
                }
            }
            new_ctrl_var.tuple_elements.push_back(rand);
        }

        this->stack.push_back(new_ctrl_var);
    }

    void use_rule_10(){
        // remove the gamma node from the control
        this->control.pop_back();

        // remove the tuple from the stack
        ctrl_and_stack_var tuple_elm = this->stack.back();
        this->stack.pop_back();

        // remove the index from the stack
        ctrl_and_stack_var index = this->stack.back();
```

```
            this->stack.pop_back();

            // get the desired tuple element
            ctrl_and_stack_var desired_tuple_elm =
                        tuple_elm.tuple_elements[index.int_var-1];

            // push the desired tuple element to the stack
            this->stack.push_back(desired_tuple_elm);
        }

        void use_rule_11(){
            use_rule_4(true);
        }

        void use_rule_12(){
            // Remove the Gamma node from the control
            this->control.pop_back();

            // Remove Y from the stack
            this->stack.pop_back();

            // Get the lambda variable from the stack
            ctrl_and_stack_var cur_stack_var = this->stack.back();
            this->stack.pop_back();

            cur_stack_var.is_lambda_var = false;
            cur_stack_var.is_eta_var = true;
            cur_stack_var.eta.bdd_var_cnt =
cur_stack_var.lambda.bdd_var_cnt;
            cur_stack_var.eta.bdd_vars = cur_stack_var.lambda.bdd_vars;
            cur_stack_var.eta.ctrl_no = cur_stack_var.lambda.ctrl_no;
            cur_stack_var.eta.env_no = cur_stack_var.lambda.env_no;

            this->stack.push_back(cur_stack_var);
        }

        void use_rule_13(){
            // Push the gamma node
            ctrl_and_stack_var new_ctrl_var;
            new_ctrl_var.is_name_var = true;
            new_ctrl_var.name_var = this->control.back().name_var;

            this->control.push_back(new_ctrl_var);

            // Push the lambda node
            ctrl_and_stack_var cur_stack_var = this->stack.back();

            cur_stack_var.is_eta_var = false;
            cur_stack_var.is_lambda_var = true;
            cur_stack_var.lambda.bdd_var_cnt =
cur_stack_var.eta.bdd_var_cnt;
            cur_stack_var.lambda.bdd_vars = cur_stack_var.eta.bdd_vars;
            cur_stack_var.lambda.ctrl_no = cur_stack_var.eta.ctrl_no;
            cur_stack_var.lambda.env_no = cur_stack_var.eta.env_no;

            this->stack.push_back(cur_stack_var);
        }
```

Exception handling

The exceptions that could occur when tokenizing, creating the AST from the tokens, standardizing the AST, and evaluating the CSE machine were handled in appropriate places. The program will stop its execution and report the error in the command line.