Final Report

RPAL Interpreter

Group 13

CHINTHANA S.K.G. 200093M

JANITH H.M.K. 200236G

JAYASINGHE A.I. 200255M

MOHOTTALA B.M.K.L. 200399G

Introduction

This report presents a detailed overview of the key components of our RPAL compiler: the lexical analyser, parser, and control stack environment (CSE).

1. **Lexical Analyser**
   The lexical analyser is the first phase of the RPAL compilation process. Its primary responsibility is to read the input source code and convert it into a stream of tokens. These tokens, which stand in for keywords, identifiers, operators, and other syntactic elements, are the basic building blocks of the RPAL language.

2. **Parser**
   The parser takes over the compilation process once the token stream has been produced by the lexical analyser. The parser uses a bottom-up parsing technique, specifically the iterative parsing approach, to construct the Abstract Syntax Tree (AST) from the token stream. Parser employs a set of grammar rules that define the valid syntax and structure of RPAL programs. By applying the iterative parsing approach, we can efficiently recognize and handle the language's complex grammar.

3. **Control Stack Environment (CSE)**
   Compiler includes a Control Stack Environment (CSE) to manage the execution of RPAL programs. The CSE is an essential component that facilitates the handling of function calls, variable bindings, and control flow during runtime. When the RPAL program is executed, the CSE maintains a stack data structure to keep track of active function calls and their associated environments. This enables us to handle recursion, scoping, and nested function definitions effectively.

Program Structure

main.cpp works as the starting point of the program. Other than that following file modules are used and stored in a separate folder called “support”.

The programme will begin to execute from main.cpp

main.cpp

```
#include <stdio.h>
#include <fstream>
#include "support/LexicalAnalyser.h"
#include "support/Parser.h"
```

```cpp
using namespace std;

ifstream fileRead;

int main (int argc, char *argv[]){

    // only two args possible
    //      main
    //      filename
    if (argc >= 2) {

        char* filename = argv[argc-1];


        //opening the RPAL file
        fileRead.open(filename);

        // check for file existence
        if (!fileRead.good()){
            printf ("File \"%s\" not found\n", filename);
            return 0;
        }


        // calling tokenization
        auto rpalLexer = new LexicalAnalyser(&fileRead);

        // calling bottom up abstract syntax tree construction
        auto rpalParser = new Parser(rpalLexer);

        //function for CSE handnling
        rpalParser->evaluateProgram();

        //closing the RPAL file
        fileRead.close();

    }


    // for unexpected argument count
    else {
        cout << "usage : ./rpal20 <filename>" << endl;
    }

}
```

1. Lexical Analyzer

Scanner + Screener

Tokenizing and identifying each tokens

- *LexicalAnalyser.cpp*
- *LexicalAnalyser.h*

lexicalAnalyzer.h file

```
class LexicalAnalyser {

public:

    LexicalAnalyser(ifstream*);
    virtual ~LexicalAnalyser();
    Token* getNxtToken();
    void lexerReset();
private:
    ifstream* file;
    int lineCount, charCount;
    string tokenIdentifier();
    string tokenInteger();
    string tokenStrings();
    string tokenSpaces();
    string tokenComment();
    string tokenOperator();
    string tokenPunctuation();

    bool isPunctuation(char);
    bool isOperatorSymbol(char);
    bool isSpaces(char);
    bool isEOL(char);
    bool isEOF(char);
    bool isCommentChar(char);
    bool isKeyword(string);

};

#endif /* LEXICAL_ANALYSER_H_ */
```

Provinding common model of tokens

- *Token.cpp*
- *Token.h*

Token.h file

```
enum tokenType{
    TOK_DEFAULT = 0,
    TOK_KEYWORD = 1,
    TOK_IDENTIFIER = 2,
    TOK_OPERATOR = 3,
    TOK_WHITESPACE = 4,
    TOK_COMMENT = 5,
    TOK_STRING = 6,
    TOK_INTEGER = 7,
    TOK_PUNCTUATION = 8,
    TOK_EOF = 9,
    TOK_DELETE = 10,
    TOK_ERROR,
    TOK_ANY
};

class Token {

public:
    Token();
    virtual ~Token();
    int tokType;
    std::string tokValue;
    int lineCount, chrCount;

};

#endif /* TOKEN_H_ */
```

2. Parser

Common model of tree nodes

- *Node.h*

Node.h file

```
class Node{

public:
    enum Type {

        LAMBDA = 1 ,
        WHERE = 2 ,
        TAU = 3 ,
        AUG = 4 ,
        TERNARY = 5 ,
        OR = 6 ,
        AND_LOGICAL = 7 ,
        NOT = 8 ,
        GR = 9 ,
        GE = 10 ,
        LS = 11 ,
        LE = 12 ,
        EQ = 13 ,
        NE = 14 ,
        ADD = 15 ,
        SUBTRACT = 16 ,
        NEG = 17 ,
        MULTIPLY = 18 ,
        DIVIDE = 19 ,
        EXPONENTIAL = 20 ,
        AT = 21 ,
        GAMMA = 22 ,
        TRUE = 23 ,
        FALSE = 24 ,
        NIL = 25 ,
        DUMMY = 26 ,
        WITHIN = 27 ,
        AND = 28 ,
        REC= 29 ,
        BINDING = 30 ,
        FCN_FORM = 31 ,
        PARANTHESES = 32 ,
        COMMA = 33 ,
        LET = 34 ,
        IDENTIFIER = 35 ,
        STRING = 36 ,
```

```
        INTEGER = 37,
        YSTAR = 38

        };

    string nodeString;
    Node* childNode = NULL;
    Node* siblingNode = NULL;
    int type;

};


#endif /* NODE_H_ */
```

Parser part of the program

- *Parser.cpp*
- *Parser.h*

Parser.h file

```
class Parser {

public:
    Parser (LexicalAnalyser*);
    void evaluateProgram();
    virtual ~Parser();

private:
    LexicalAnalyser* lex;
    stack <Node*> treeStack;
    Token *nextToken;

    void E();
    void Ew();
    void T();
    void Ta();
    void Tc();
    void B();
    void Bt();
    void Bs();
    void Bp();
    void A();
    void At();
```

```
    void Af();
    void Ap();
    void R();
    void Rn();
    void D();
    void Da();
    void Dr();
    void Db();
    void Vb();
    void Vl();

    void parse();
    void standardize(Node*);
    void buildTree(string, int);
    void buildTree(string, int, int);
    void treePrettyPrint(Node*, int);
    void read(string);
    bool isKeyword(string);
    string to_s(Node*);

};

#endif /* PARSER_H_ */
```

Standardizing tree generated by parser

- *TreeStand.cpp*
- *TreeStand.h*

TreeStand.h file

```
class TreeStand {
    void standardize(Node*);
    void standardizeLET(Node*);
    void standardizeWHERE(Node*);
    void standardizeWITHIN(Node*);
    void standardizeREC(Node*);
    void standardizeFCNFORM(Node*);
    void standardizeLAMBDA(Node*);
    void standardizeAND(Node*);
    void standardizeAT(Node*);
public:
    TreeStand(Node* topNode);
    virtual ~TreeStand();
};
#endif /* TREESTAND_H_ */
```

3. CSE machine

Environment stack management

- *Env.cpp*
- *Env.h*

Env.h file

```
class Env {
public:
    int id;
    Env *parent;
    void assignParent(Env* );
    Ctrl* lookup(string);
    Env(int);
    virtual ~Env();
    map<string, Ctrl *> symbolTable;
};

#endif /* ENV_H_ */
```

Control structure management

- *Ctrl.cpp*
- *Ctrl.h*

Ctrl.h File

```
class Ctrl{

public:
  void addCtrl(Node* node, int , string, vector<string> *, Ctrl*, int);
  enum Type{
    ENV = 1 ,
    DELTA = 2 ,
    NAME = 3 ,
    LAMBDA = 4 ,
    GAMMA = 5 ,
    AUG = 6 ,
    BETA = 7 ,
    OR = 8 ,
    AND_LOGICAL = 9 ,
    NOT = 10 ,
    GR = 11 ,
    GE = 12 ,
    LS = 13 ,
```

```
    LE = 14 ,
    EQ = 15 ,
    NE = 16 ,
    ADD = 17 ,
    SUBTRACT = 18 ,
    NEG = 19 ,
    MULTIPLY = 20 ,
    DIVIDE = 21 ,
    EXP = 22 ,
    AT = 23 ,
    TRUE = 24 ,
    FALSE = 25 ,
    NIL = 26 ,
    DUMMY = 27 ,
    YSTAR = 28 ,
    ETA = 29 ,
    TAU = 30 ,
    STRING = 31 ,
    INTEGER = 32 ,
    TUPLE = 33
    };
  string toStr();
  Ctrl(Ctrl *ctrl);
  Ctrl(Ctrl::Type type, int index);
  Ctrl(Ctrl::Type type, vector<string> *variables, Ctrl *del_ptr, int
delta_index);
  Ctrl(Ctrl::Type type, int index, bool b);
  Ctrl(string var_value, Ctrl::Type type );
  Ctrl(Ctrl::Type type, string value);
  Ctrl();
  Ctrl(Ctrl::Type type);
  vector<Ctrl *> *ctrlStruct;
  vector<string> variables;
  vector<Ctrl *> ctrlTuples;
  int associatedENV;
  Type type;
  int index;
  string ctrlVal;
  Ctrl *delta;

};
```

CSE machine code

- *CSE.cpp*
- *CSE.h*

CSE.h File

```
static vector <string> inbuiltFuncVector =
        {"Order","Print","Isinteger","Istruthvalue","Isstring",
         "Istuple","Isfunction","Isdummy","Stem","Stern",
         "Conc","Conc2","ItoS","Null"};

class CSE {

    void flattenDeltaThen(Node*, Ctrl *, vector<Ctrl *> *);
    void flattenDeltaElse(Node*, Ctrl *, vector<Ctrl *> *);
    void flattenLAMBDA(Node*, Ctrl *, vector<Ctrl *> *);
    void flattenTernary(Node*, Ctrl *, vector<Ctrl *> *);
    void flattenTree(Node*, Ctrl *, vector<Ctrl *> *);
    void init(Node*);
    void applyBinaryOPR(int);
    void applyThisRator(Ctrl*);
    void printCS();
    bool checkInbuilt(string);
    void escapePrintStr(string);
    void rule411(Ctrl*, Ctrl*, Ctrl*, Env*, int);
    void rule12(Ctrl*, Ctrl*, Ctrl*, Env*, int);
    void rule13(Ctrl*, Ctrl*, Ctrl*, Env*, int);
    void rule10(Ctrl*, Ctrl*, Ctrl*, Env*, int);
    void handleNeg(Ctrl*, Ctrl*, Ctrl*, Env*, int);
    void handleEnv(Ctrl*, Ctrl*, Ctrl*, Env*, int);
    void handleTau(Ctrl*, Ctrl*, Ctrl*, Env*, int);
    void handleBeta(Ctrl*, Ctrl*, Ctrl*, Env*, int);
    void handleName(Ctrl*, Ctrl*, Ctrl*, Env*, int);
    void handleGAMMA(Ctrl*, Ctrl*, Ctrl*, Env*, int);

    Node* top;
    vector<Ctrl *> *deltas;
    Ctrl *rootDelta;
    int numEnvironment;
    vector<Ctrl *> control;
    stack<Ctrl *> execStack;
    Env *currEnvironment;
    Env *tempEnvironment;
    Env *PE;
    stack<Env*> environmentStack;
    map<int, Env *> environmentMap;

public:
```

```
    CSE();
    CSE(Node*);
    virtual ~CSE();
    Env* primitiveEnv;
    void run(Node*);
    Env* createNewEnvironment();
    void execCSE();

};
#endif /* CSE_H_ */
```

Conclusion

This RPAL compiler has been developed, providing an essential tool for programmers to work with the RPAL programming language. This project represents a significant achievement in compiler construction, empowering developers to utilize RPAL effectively and facilitating future language processing advancements.