CS 3513 - Programming Languages

Programming Project

Implementation of RPAL Compiler

Group – 45

ABISHERK S. 210018B

SUBAVARSHANA A. 210621R

Introduction

In this project, we aim to develop a lexical analyzer and a parser for RPAL (Right-reference Pedagogic Algorithmic Language), a functional programming language derived from PAL, which originated at MIT in the early '70s by J. Wozencraft and A. Evans. RPAL programs are expressed as expressions and executing them involves evaluating these expressions.

Initially, we create a lexical analyzer based on the lexical rules outlined in the [1] document of RPAL's Lexicon. The role of this analyzer is to tokenize the input RPAL program and produce a stream of tokens representing the various elements within the code.

Following that, we construct a parser utilizing the grammar specifications provided in the [2] document of RPAL's Phrase Structure Grammer. The parser receives the token stream generated by the lexical analyzer and builds an Abstract Syntax Tree (AST) reflecting the program's structure. The AST serves as a condensed representation of the Derivation Tree, facilitating subsequent processing.

Once we have the AST, we transform it into a Standardized Tree (ST) using the RPAL Subtree Transformational Grammar table, as detailed in the document available at [3] sourceforge website. The ST refines the AST, preparing it for evaluation.

For evaluating RPAL programs, we implement the CSE (Control-Stack-Environment) machine. The CSE machine operates according to specific rules governing the evaluation process. These rules are outlined in the slides provided at [4] pdfslide website. Our task involves understanding and implementing these rules to execute RPAL programs effectively.

Note: The implementation is done in Python.

Fundamental concepts in RPAL language

Our RPAL programming language system is meticulously crafted to embrace and manage an array of fundamental concepts pivotal to functional programming. These core concepts encompass:

Data Types:

RPAL embraces a diverse range of data types to accommodate various programming needs:

1. Integer: Ideal for representing whole numbers in arithmetic operations and calculations.

2. Truth-value (Boolean): Facilitates binary logic operations with values of true or false.

3. String: Enables the representation and manipulation of character sequences.

4. Tuple: A versatile collection of elements, each potentially of different data types.

5. Function: Functions themselves are treated as first-class data types, affording them the same flexibility and utility as any other data entity.

Conditional Expressions:

RPAL furnishes support for conditional expressions via the "if-then-else" construct. This facilitates decision-making based on truth values, thereby enabling the execution of distinct code blocks as warranted.

Constant Definitions:

The language also accommodates the declaration of constants, empowering users to establish fixed values that endure unaltered throughout program execution.

Function Definitions:

In RPAL, users have the freedom to define their own functions. Following the lambda calculus paradigm, functions are bestowed with first-class citizenship, permitting assignment to variables and passage as arguments to other functions.

Function Application:

Integral to functional programming, RPAL allows the application of functions to arguments. This paradigm underpins the process of utilizing functions to transform inputs into desired outcomes.

Operators:

RPAL boasts a rich set of operators that facilitate the manipulation and evaluation of data. Among these operators are arithmetic operations designed for integers (+, -, *, /, **) alongside truth-value operations (or, &, not, eq, ne), and string manipulations (eq, ne, Stem S, Stern S, Conc S T).

Recursion:

Recursion, a cornerstone of RPAL, furnishes the ability for functions to call themselves within their definitions. This elegant technique empowers the formulation of iterative solutions to complex problems.

These foundational data types collectively serve as the cornerstone for crafting sophisticated and adaptable programs within the RPAL ecosystem.

Classes

ASTNode Class:

The ASTNode class embodies the representation of nodes within the Abstract Syntax Tree (AST) or Standardized Tree (ST) of an RPAL program. Each node holds attributes for first, sibling, and previous, facilitating navigation through the tree structure. These attributes allow for linkage to child nodes, next siblings, and preceding siblings. A Token object representing the lexical token correlated with the node is encapsulated within, encompassing token type, lexeme, line number, and column number. Nodes serve as the building blocks for hierarchical structures within RPAL programs, aiding in parsing, evaluation, and subsequent code processing.

ControlStructure Class:

The ControlStructure class acts as the primary orchestrator for generating and managing control structures within RPAL programs. It maintains a queue of triples and a map to represent control structures. The class facilitates pre-order traversal of the abstract syntax tree (AST) and tracks index deltas during traversal.

CSEMachine Class:

The CSEMachine class embodies the abstract CSE machine, featuring stacks for control structures, data, and environments, along with mappings. It initializes required stacks and mappings and contains methods for stack manipulation and applying operations. The class orchestrates the evaluation loop for the CSE machine, processing control structures, evaluating expressions, and managing stacks and environments in accordance with RPAL language rules.

Lexical Class:

The Lexical class assumes responsibility for tokenizing the input program. It reads the program from a file and processes it line by line, employing methods to discern various token types such as identifiers, strings, operators, delimiters, integers, and comments. The scan() method sequentially retrieves tokens from the input program.

Parser Class:

The Parser class undertakes the parsing of tokenized input programs and the construction of the Abstract Syntax Tree (AST). It defines methods for parsing diverse expressions, statements, and definitions. The E() method initiates expression parsing, while the D() method commences definition parsing. Employing a recursive descent strategy, the parser navigates through the input program.

Standardizer Class:

The Standardizer class is tasked with standardizing the AST of a program. It conducts a post-order traversal, identifying and transforming specific control structures and expressions into standardized forms. The class provides methods for standardizing different control structures, enhancing the AST's uniformity and aiding subsequent processing and analysis.

TreeBuilder Class:

The TreeBuilder class is instrumental in constructing the Abstract Syntax Tree (AST) during parsing. It defines methods for creating AST nodes based on parsed tokens.

Steps in running the program

Step 01

Extracting the compressed zip file

Can use the following command to extract the file to a directory of files containing the project.

(replace with the actual filename)

>> **tar xvf <submission_file>.tar**

Step 02

We can execute the RPAL program using the following python command

>>**python .\myrpal.py file_name**

Where file_name is the name of the file that has the RPAL program as the input.

(You can also include directory of the input file also)

To get the Abstract Syntax Tree (AST) can use the following command

>>**python .\myrpal.py -ast file_name**

References

[1] RPAL's Lexicon - https://rpal.sourceforge.net/doc/lexer.pdf

[2] RPAL's Phrase Structure Grammer - https://rpal.sourceforge.net/doc/grammar.pdf

[3] RPAL Subtree Transformational Grammar table - https://rpal.sourceforge.net/doc/semantics.pdf

[4] CSE rules - https://pdfslide.net/documents/the-cse-machine.html