CS3513 - Programming Languages

Programming Project - Report

May 2024

Group 0820

Team Members
D.G.H. Prabashwara - 210483T
I.T.M. Perera - 210460V

Task Overview

The task assigned was to implement an interpreter to read a RPAL program and give the relevant output.

Under the above task, firstly it was required to implement a lexical analyzer and a parser for the RPAL language. The parser must output the Abstract Syntax Tree (AST) of the given input program. Then an algorithm must be implemented to convert the AST to its Standardised Tree (ST) and the CSE (Control Stack Environment) machine also be implemented.

Solution

The implemented solution to the assigned task includes the following processes:

1. **Tokenization** - The input program is broken down into a stream of tokens. Basic token types such as keywords, integers, strings, identifiers, operators, punctuation and removable tokens (comments, whitespaces, etc.) are identified here.

2. **Parsing** - The token list given from the previous step is parsed using the bottom-up parsing approach. A recursive descent parser prepared based on RPAL grammar rules is used in this task.

3. **Building the Abstract Syntax Tree (AST)** - An AST is built during the parsing process.

4. **Building the Standardised Tree (ST)** - The built AST is converted to a Standardised Tree (ST) by eliminating redundant expressions and constructs.

5. **Evaluating using the CSE machine** - The CSE machine is a virtual stack-based machine designed to execute the RPAL program represented by the Standardised Tree (ST). The machine operates on a stack, manipulating data and function calls.

6. **Error handling** - Any syntax error or parsing error in the input RPAL program is detected and displayed.

Implementation

The above process was implemented by us using the following components:

1. Scanner

The scanner, or the Lexical Analyzer breaks down the input RPAL program into a list of tokens. A token is a key-value pair where value is the term in the input RPAL program and the key is its type. The token types identified by the scanner are as follows.

Token Type	**Values**
IDENTIFIER	User-defined names consisting of an arbitrarily long sequence of letters, digits, and underscore. Begins with a letter or an underscore.
INTEGER	Numerical values
OPERATOR	Symbols representing operations like arithmetic, comparison, or logical operations Eg.: +, -, /, @, <, >, =
PUNCTUATION	Special characters Eg.: “ (”, “) ”, “ ; ”, “ , ”
KEYWORD	Reserved words with specific meanings in RPAL

	Eg.: “let”, “in”, “where”, “within”
STRING	String literals (Any string of characters enclosed in ‘ ’, a pair of single quotes, is considered a string)
DELETE	Characters ignored during interpretation, including comments (started with //) and whitespaces (spaces, tabs, newlines)

The tokens are built by reading the RPAL program character by character and classifying them into the above types based on the character stream. The process is done using the following data structures and classes.

Reader class - This class is implemented to read the RPAL program from the input file. A file name must be given when initialising an object of this class.

Class attributes

- file - the input file name
- str - a string; used to store the text in the file as a string; initialised with a blank string
- lines - a list to store the text in the file line by line, each line a string; initialised with an empty list

Class methods

- read_whole() - Reads the whole file as a string. Returns the content in the file as a string.
- read_lines() - Returns the lines of the file as a list. Returns a list containing lines of the input text as strings.
- find_line(index) - Given an index, finds the corresponding line and the corresponding line index of a character. Returns the line number and the index of the position within the line.
- get_line(line_number) - Given the line number, returns the line.

Recognizer class - This class is implemented to recognise the class of a given character. Lists of characters belonging to different classes are declared within the class and methods have been implemented to recognize the character type.

Token class - Defines the structure of a token. When initialising a token object, the token type and the value is given, and the class returns a token of the format "<type, value>". If the token value is among the keywords of RPAL, it creates a token of the format "<'KEYWORD', value>".

Scanner class - This class is responsible for the scanning process. The name of the file to be scanned must be given when initialising an object of this class.

Class attributes

- reader - A 'Reader' object, initialised with the given file name.
- source_list - The content of the input file, read by the "read_whole()" method of the 'reader' object.
- token_list - A list that stores the created token list. Initialised with an empty list.
- curr_index - Keeps track of the index of the value being tokenized in the 'source_list'. Initialised with 0.
- last_index - Stores the index of the last value in the 'source_list'.
- errors - An 'ErrorHandler' object (discussed later)
- recognizer - A 'Recognizer' object
- screened - A boolean variable to keep track whether the token list was screened or not. Initialised with 'False'.
- <type>_name - The token type names

Class methods

- handle_identifier() - Handles identifier tokens. Adds an identifier token to the 'token_list'.
- handle_integer() - Handles integer tokens. Adds an integer token to the 'token_list'.
- handle_operator() - Handles operator tokens. Adds an operator token to the 'token_list'.
- handle_comment() - Handles comment tokens. Adds a comment token to the 'token_list'.
- handle_space() - Handles space tokens. Adds a space token to the 'token_list'.

- handle_string() - Handles string tokens. Adds a string token to the 'token_list'.
- handle_punctuation() - Handles punctuation tokens. Adds a punctuation token to the 'token_list'.

  The above methods handle the respective token type based on the character class obtained using the 'recognizer'.

- screen() - Does the screening process for the created tokens. Removes any deletable token (token type is DELETE) and updates the 'token_list'. Then sets the value of 'screened' to 'True'.
- tokenize() - Read the first character or the second character from the 'source_list' and call a separate handler function for each data type. Returns the token list of the source code.
- get_tokens() - Returns the scanned and screened token list.

2. Parser

This is implemented in the "parser.py" file. Here, the token list from the scanner above is parsed to build the Abstract Syntax Tree (AST) and its Standardised Tree (ST) is built using the AST. A recursive descent parser, prepared based on the RPAL grammar rules, is used for parsing. The parser uses the following components for its operation.

Node class - Defines the structure of a node in the Abstract Syntax Tree and the Standardised Tree. The data to be put in the node must be given when initialising a Node object.

Class attributes

- data - The data stored in the node (a token)
- children - A list that stores the children nodes of the node. Initialised with an empty list.

Class methods

- add_child(child) - Given a node, adds the node to the start of 'children' list.
- add_child_end(child) - Given a node, appends the node to the 'children' list.

- remove_child(child) - Given a node, if the node is in the 'children' list, removes the node from the list; otherwise displays an error message.

Parser class - This class is responsible for the parsing, building the Abstract Syntax Tree and the Standardised Tree. A screener must be passed as a parameter when initialising an object from this class.

Note: Abstract Syntax Tree and Standardised Tree are node trees.

Class attributes

- screener - a screener to obtain the scanned and screened token list
- token_list - the token list obtained from from the screener
- next_token - holds the token to be parsed. Initialised with an empty token.
- index - holds the index of the token that is being parsed
- stack - holds the Abstract Syntax Tree (AST). Initialised with an empty list.
- ST - holds the Standardised Tree (ST). Initialised with an empty object.
- pre_ordered - holds the list of items obtained by pre-order traversing the ST. Initialised with an empty list.

Class methods

- build_tree(transduction, n) - Given a token (transduction) and the number of children under the transduction node (n), this function builds the AST by creating a node from the token, popping n nodes from the 'stack' and adding them to the 'children' list of the created node. Then this node is added into the 'stack' (AST).
- read(token) - Reads the provided token and increments the index to move to the next token to be read. The read operation is done by calling the build_tree() method, based on the token type.
- parse() - Initiates the parsing process. Firstly, a new token ($, $) is added to identify the end of the token list. Then the function corresponding to the starting symbol of the RPAL grammar, E(), is called to start the parsing process. If ($, $) token is reached during the parsing process, it indicates the parsing was successful.

- Functions have been prepared for each rule in the RPAL grammar. RPAL grammar is a context-free grammar.
- printAST() - Prints the AST for the given source code in the format that was given under this assignment.
- printST() - Prints the corresponding Standardised Tree.
- printNode(node) - Given a node, prints the node and its 'children'.
- buildST(node) - Given a node, this function standardises the node using the standardisation rules. Rules are implemented inside the function and based on the node 'data', the relevant rule is applied. When a node is given, the rules are applied to the 'children' of the node (by making a recursion call to the function for each child node), then to the node itself. Thus, this function builds the ST from the bottom to the top.
- standardize() - Creates the Standardised Tree (ST) for the AST built during the parsing process. A copy of the AST is obtained and assigned to ST. Then the buildST() method is called while passing the first node in the AST (the root node in the AST) as the argument.

3. CSE (Control Stack Environment) machine

The CSE machine evaluates the Standardised Tree prepared in the previous part (Parser) and gives the result of RPAL source code. It is a stack-based interpreter designed to execute the RPAL program represented by the ST. It operates on a stack, manipulating data and function calls. The steps of the operation of the CSE machine can be given as follows.

1. Building the control structures - control structures are built by pre-order traversing the Standardised Tree (ST). When a 'lambda' node is encountered, a new control structure is built while adding a new <λ k x> node to the current control structure, where k is an index that references the new control structure and x is the left child of the lambda node in the ST. If a name node (an identifier or a value) or a 'gamma' node is encountered, it is added to the current control structure.

2. Operation of the CSE Machine - Three main components are present in the CSE Machine.

- Control - Holds the control structures
- Stack - Holds the values
- Environment - Holds the names and their values in a particular environment. Every environment created in the process is linked to a previously created environment, making the environment structure a tree.

The operation of the CSE machine can be given as follows.

1) The initial control structure is loaded onto Control. The Stack is initially empty. Both contain an environment marker referring to the Primitive Environment.
2) The topmost (or the rightmost) element of Control is popped out.
3) Based on the popped element and according to the CSE rules, an operation is performed.
4) The steps 2-3 are repeated until the Control is empty.
5) The value remaining in the Stack is the final result of the RPAL input program.

To perform the above set of tasks, following structures and classes have been implemented.

Lambda class - Create instances that represent lambda (**λ**) elements in the CSE machine.

Class attributes

- id - Control structure id for the lambda element
- val - Variable or variables associated with the lambda
- env - Environment id associated with the lambda

Delta class - Create instances that replicate delta (**δ**) elements in the CSE machine.

Class attributes

- id - Control Structure Id for the delta element

Tau class - Create instances that replicate the tau element in the CSE machine.

Class attributes

- size - Size of the tuple

Eta class - Create instances that replicate the eta (η) element in CSE machine

Class attributes

- id - Control structure id for the eta element
- val - Variable or variables associated with the eta
- env - Environment id associates with the eta

ControlSuctureBuilder class - Given the Standardised Tree, generates the control structures.

Class attributes

- ST - The ST obtained by the parser
- control_structures - Holds the generated control structures
- count - Holds the number of control structures built

Class methods

- pre_order() - Does a pre-order traversal on ST and generates the control structures.
- linearize() - Returns the generated control structures
- printCS() - Prints the generated control structures

Stack class - Create stack objects to be used in the CSEMachine class.

Class attributes

- items - Hold the elements pushed into the stack

Class methods

- push(item) - Pushes a given item to the stack
- pop() - Pops the top item from the stack
- is_empty() - Returns true if the stack is empty
- peek() - Return the top element of the stack without popping it
- print_stack() - Prints all the stack elements
- get_elements() - Returns all the stack elements

Environment class - Creates environment objects for CSEMachine class.

Class attributes

- name - Name of the environment
- variables - Variables inside the environment
- children - Childrens of the environment
- parent - Parent environment

Class methods

- add_child(child_env) - Add a child environment to the current environment instance.
- add_variable(key, value) - Add the variable name (key) and its value (value) to the current environment instance.

CSEMachine class - Evaluate the provided control structures using the Environment instances and a Stack instance. This is the implementation of the CSE Machine.

Class attributes

- control_structures - Holds the provided control structures
- errors - Holds the error stack. Passed as an argument.
- stack - Stack for the CSE machine
- control - Currently evaluating control structure/s
- environments - All the Environment instances
- current_environment - The Id of the current environment
- built_in_functions - All the built-in functions in RPAL
- binop - A list containing all the binary operation keywords.
- unop - A list containing all the unary operation keywords.
- print_state - Returns true if the “Print” command is found. Initialised to false.

Class methods

- lookup(var) - Given a variable name (var), returns its value by processing or searching in the current environment.
- apply_rules() - Apply the CSE rules on the CSE machine
- print() - Print the final value in the stack if the ‘print_state’ is true.

4. Error Handling

Because of the issues in the RPAL source code, errors could occur during the execution process. This RPAL implementation will consider errors occurring during the scanning, parsing and CSE machine execution. And, having errors does not stop the scanning process and parsing process giving the users the ability to find all errors in the first run. The errors encountered will be stacked in a separate **ErrorHandler** object. And there are three types of errors.

- Syntax Errors
- Unrecoginced Character
- Parsing Errors
- Unsupported Operands
- Zero Division Error

And other than these three types of errors, the error of Unrecognized Filename is handled in the reader class. If the file is not recognized the program execution will stop immediately.

ErrorHandler class - This class is responsible for handling errors occurring during the scanning, screening and parsing processes.

Class attributes

- reader - Reader instance associated with the scanning. Must be passed as an argument during initialisation of an 'ErrorHandler' object.
- error_status - A boolean value indicating the presence of errors. Default value is set to false. If any error is encountered, the value changes to true.
- source_list - Generated string list from the source code. Passed as an argument when initialising the 'ErrorHandler' object
- error_list - Encountered error list. Initialised with an empty list.

Class methods

- syntax_error(index) - Given the index where the syntax error occurred in the source list, adds the error to the stack (error_list) which includes the line number and the position of the error in the source code.
- unrecognized_error(index) - Given the index where the unrecognised character appeared in the source list, adds the error to the stack (error_list)

which includes the line number and the position of the unrecognised character in the source code.

- parse_error(error) - Given the error string involved with parsing add the parsing error to the error stack (error_list).
- unsupported_operands(operation, types) - Adds an unsupported operand error to the error stack (error_list). The parameters ‘operation’ and ‘types’ represent the operation being performed and the unsupported data type being used respectively.
- zero_division_error(operand1) - Adds a zero division error to the error stack (error_list). The parameter ‘operand1’ is the number that is being divided by zero.
- print() - If the error_status is true, prints all the errors in the error stack (error_list).