CS 3513 - Programming Languages

Programming Project 01

Prepared by Group 5

210249H – Jayasinghe M.P.S

210648F – Thilakarathna K.D.B

Date: 10/5/2024

Problem Description

- We are required to implement a lexical analyzer and a parser for the RPAL language.
- Output of the parser should be the Abstract Syntax Tree (AST) for the given input program.
- Then we need to implement an algorithm to convert the Abstract Syntax Tree (AST) into Standardize Tree (ST) and implement a CSE machine.
- Our program should be able to read an input file which contains a RPAL program.
- The output of our program should match the output of “rpal.exe“ for the relevant program.

How to Run

Basic Usage

```
C:\PROGRAMMING\compilerDesign\PL-project-compiler-RPAL-\reCreate\main>python main.py  defns.1
[lambda closure: x: 2]
```

1. **Open your terminal or command prompt.**
2. **Navigate to the directory where your main.py file is located.** (In the image, it's C:\PROGRAMMING\compilerDesign\PL-project-compiler-RPAL-\reCreate\main)
3. **Run the compiler with the following command: python main.py <filename>**

To See the AST:

1. **Use the --ast flag: python main.py --ast <filename>**

```
(c) Microsoft Corporation. All rights reserved.

C:\PROGRAMMING\compilerDesign\PL-project-compiler-RPAL-\reCreate\main>python main.py --ast defns.1
gamma
.<ID:Print>
.let
..function_form
...<ID:f>
...<ID:x>
...<ID:y>
...let
....function_form
.....<ID:g>
.....<ID:x>
.....<ID:y>
.....let
......function_form
.......<ID:h>
.......<ID:x>
.......<ID:y>
.......gamma
........gamma
.........<ID:x>
.........<ID:y>
........<ID:x>
......<ID:h>
....<ID:g>
..<ID:f>


[lambda closure: x: 2]
```

Main Flow(main.py)

- **Handles Command-Line Arguments:** Accepts the source file path and an optional flag to display the AST.
- **Performs Lexical Analysis:** Reads the source code and converts it into tokens.
- **Parsing + Builds the AST:** Uses the tokens to construct the Abstract Syntax Tree.
- **Transforms AST to ST:** Converts the AST into a Syntax Tree (ST).
- **Generates Control Structures:** Extracts control flow information from the ST.
- **Evaluates Code:** Runs the generated control structures through the CSE machine to produce the final output.
- **Handles Errors:** Catches and reports any exceptions or runtime errors encountered during compilation.

```
lines=file.readlines()
tokens=get_next_token(lines,tokens)
pasrser = Parser(tokens)
ast=pasrser.buildAst()

if astVisible:
    print(ast.getAST())
    print("")


text =ast.getAST().split("\n")
root=CreateTree().nodeFromFile(text)
AstToSt().astToSt(root)
controls=ElementParser().generateCs(root)
cseMachine=CSEMachine(controls)
cseMachine.evaluateTree()
```

1. Scanner (Lexical analyzer)

A compiler translates a program (source) written in one language into a program written in a lower-level language than its source language. Compilation consists of multiple phases. Scanning belongs to phase one.

In scanning (also called lexical analysis, linear analysis, or lex), we are going to implement a lexical analyzer (also known as a lexer or scanner) that reads the input RPAL program and converts it into a stream of tokens. Tokens are the smallest meaningful units in a programming language (e.g., keywords, identifiers, operators, etc.). Define rules to identify different types of tokens based on the RPAL language specifications.

As the second part of phase one (screening), we discard unwanted tokens like white spaces, comments, tabs, and new lines. Merge and simplify tokens.

- **Token Class:** Defines the structure to store token information (type, value, line number).
- **is_... Functions:** Use regular expressions to check if a portion of text matches a specific token type (letter, digit, operator, etc.).
- **get_next_token Function:** The main function of the scanner. It iterates through the source code, identifying and classifying individual tokens using the helper functions.

Function Prototypes

```
class Token:
    def __init__(self, type, value,line)
    def getType(self)
    def getValue(self)
    def getLineNumber(self)

def is_letter(line)
def is_digit(line)
def is_operator(line)
def is_punctuation(line)
def is_string(line)
def is_removeable(line)
def get_next_token(lines,tokens)
```

2. Parsing

The second phase of the compilation process is parsing. Here we group the tokens into the correct syntactic structures as expressions, statements, procedures, functions and modules.

- **Parser Class:**
    - __init__(self, tokens): Initializes the parser with the list of tokens from the lexer.
    - getNextToken(): Fetches the next token from the list and potentially creates a terminal AST node for it.
    - buildAst(): Initiates the parsing process and returns the built Abstract Syntax Tree (AST).
    - startParsing(): Starts the parsing process by calling the procE function.
    - createTerminalASTNode(type, value): Creates a leaf node in the AST with the given type and value.
    - createArrayAstNode(type, ariness): Creates a non-leaf node in the AST with multiple children, representing an operation or expression.
    - isCurrentToken(type, value): Checks if the current token matches the given type and value.
    - isCurrentTokenType(type): Checks if the current token matches the given type.
- **Recursive Descent Parsing Functions (proc...)**:
    - procE(), procEW(), procT(), procTA(), etc.: These functions implement the recursive descent parsing strategy, each corresponding to a non-terminal in the grammar of the language being parsed. They consume tokens, check for correct syntax, and build the AST based on the grammar rules.

Example Description for procE()

procE(): This function parses an expression (E) based on the grammar rules. It handles different cases like "let" expressions (creating a "LET" node), lambda expressions (creating a "LAMBDA" node), and other expressions (Ew) by calling the appropriate sub-parsing functions.

Function Prototypes

```
class Token:
    def __init__(self, type, value,line)
    def getType(self)
    def getValue(self)
    def getLineNumber(self)
```

```
class AST:
    def __init__(self,root) -> None
    def getAST(self)
    def preOrder(self,node,printPrefix)
    def addNodeDetails(self,node,printPrefix)
class ASTNode:
    def __init__(self) -> None
    def getName(self)
    def getPrintName(self)
    def getType(self)
    def setType(self,type)
    def getChild(self)
    def setChild(self,child)
    def getSibling(self)
    def setSibling(self,sibling)
    def getValue(self)
    def setValue(self,value)
    def getSourceLineNumber(self)
    def setSourceLineNumber(self,sourceLineNumber)

class Parser:
    def __init__(self, tokens)
    def getNextToken(self)
    def buildAst(self)
    def startParsing(self)
    def createTerminalASTNode(self, type, value)
    def createArrayAstNode(self, type, ariness)
    def isCurrentToken(self, type, value)
    def isCurrentTokenType(self, type)
    def procE(self)
    def procEW(self)
    def procT(self)
    def procTA(self)
    def procTC(self)
    def procB(self)
    def procBT(self)
    def procBS(self)
    def procBP(self)
    def procA(self)
    def procAT(self)
    def procAF(self)
    def procAP(self)
    def procR(self)
    def procRN(self)
    def procD(self)
    def procDA(self)
    def procDR(self)
    def procDB(self)
    def procVB(self)
    def procVL(self)
```

3. AST to Standardize Tree (ST)

Its primary purpose is to traverse an Abstract Syntax Tree (AST) generated by a parser and transform it into another tree structure known as a Syntax Tree (ST). The ST is likely a more standardized or simplified representation of the original AST, making it easier to process in later compiler phases.

- **AstToSt Class:**
  - expectChildren(node, expect): Ensures a node has a specific number of children, raising an exception if it doesn't.
  - expectMoreChildren(node, minimum): Checks that a node has at least a minimum number of children.
  - checkLabel(node, expect): Verifies a node has the expected label, raising an exception if it doesn't match.
  - astToSt(node): This is the core function. It recursively traverses the AST, performing transformations on specific node types and structures to convert it into an ST.
- **Transformation Logic (within astToSt):**
  - **Label-Specific Transformations:** The code contains a series of conditional blocks (e.g., if node.isLabel("let"):, elif node.isLabel("where"):, etc.). Each block handles the transformation of nodes with specific labels. These transformations involve restructuring the tree by adding, removing, or changing the labels of nodes. The goal is to align the AST structure with the desired ST format.

Function Prototypes

```
class AstToSt:
    def expectChildren(self,node, expect)
    def expectMoreChildren(self,node, minimum)
    def checkLabel(self,node, expect)
    def astToSt(self,node)

class Node:
    def __init__(self, label, value=None)
    def copy(self)
    def getParent(self)
    def getLabel(self)
    def getValue(self)
    def getNumChild(self)
    def hasChildren(self, n)
    def isLabel(self, label)
```

```
    def getChild(self, i)
    def forEachChild(self, action)
    def setLabel(self, label)
    def clearChildren(self)
    def addChild(self, child)

class ExceptionHandlerOfAST(Exception):
    def __init__(self, message)

class CreateTree:
    def nodeFromFile(self,astTexStrings)
    def getJavaValue(self,st)
class DepthOfNode(Node):
    def getDepth(self)
    def getParent(self)
```

4. CSE machine for created ST

This code defines a **CSEMachine** class, which acts as an interpreter for a specific type of control structure. It processes these structures using a set of rules and operates on elements called **EleValue, EleTuple**, and **EleValueOrTuple**. The primary goal is to evaluate the control structures and compute the final output of the code.

- **CSEMachine Class:**
  - __init__(self, controlStructures): Initializes the machine with the control structures and sets up initial values and environments.
  - toString(): Provides a string representation of the machine's state.
  - currentEnvironmentIndex(): Determines the index of the current environment being used.
  - currentEnvironment(): Returns the current environment.
  - evaluateTree(): The core function that evaluates the control structures by processing elements based on their labels and applying specific rules.
  - extractDelta(controlIndex): Extracts elements from a delta control structure.
  - rule1(name), rule2(lambdaVal), ..., rule13(gamma, eta): These functions implement the rules for evaluating different elements and structures within the control structure. Each rule corresponds to specific operations or transformations based on the labels of the elements.

Function Prototypes

```
class ElementParser:
    def generateCs(self,node,controls=None, currentControl=None)
    def generateCsLambda(self,node, controls, currentControl)
    def generateCsIf(self,node, controls, currentControl)
    def generateCsTau(self,node, controls, currentControl)

class Environment:
    def __init__(self, parent=None, key=None, value=None)
    def initializePrimaryEnv(self)
    def remember(self, key, value)
    def lookup(self, id)
    def toString(self)

class ExceptionHandlerOfCSE(Exception):
    def __init__(self, message)

class CSEMachine:
    def __init__(self, controlStructures)
    def toString(self)
    def currentEnvironmentIndex(self)
    def currentEnvironment(self)
    def evaluateTree(self)
    def extractDelta(self, controlIndex)
    def rule1(self, name)
    def rule2(self, lambdaVal)
    def rule3(self, rator, rand)
    def rule4(self, lambdaVal, rand)
    def rule5(self, env)
    def rule6_7(self, element)
    def rule8(self)
    def rule9(self, tau)
    def rule10(self, tupleVal, index)
    def rule11(self, lambdaVal, rand)
    def rule12(self, lambdaVal)
    def rule13(self, gamma, eta)
```

****