Code Report

Programming Languages 2410211

Dr. SHaron Yalov

Students:

Shiran Issan	207497561
Alon Bitran	206842775
Omer roth	314865569

Introduction

This project involves building an interpreter for a custom functional programming language. The language emphasizes functional concepts such as immutability, function definitions, and lambda expressions, without mutable state or variable assignments.

The interpreter processes the language through several key stages:

- **Lexer**: Converts input source code into tokens.
- **Parser**: Transforms tokens into an Abstract Syntax Tree (AST) using the language's BNF grammar.
- **Semantic Analyzer**: Validates the AST to ensure correctness. The analyzer verifies that particularly for function and lambda definitions and scope management.
- **Interpreter**: Executes the validated AST using lazy evaluation, supporting both file-parsing and an interactive REPL mode.

This interpreter handles recursive functions, higher-order functions, and immutable data, providing a strong foundation for functional programming.

Design Decisions

BNF

Assumption	Implementation
Named functions will be declared at scope level 1 which is the root of the program.	The the program will be built as a list of statements of types **empty**, **function declaration**, **parenthesized statement** and a **logical expression**.
Since there are no assignment operations, anonymous functions (lambdas) can be declared as an actual parameter of a function call, or can be declared and immediately called as a part of a logical expression	The **lambda_decleration** symbol is available as a part of the **actual_parameters** used by function calls and nested lambdas, and the **nested_lambda** symbol is available in the **factor** symbol which handles function calls.
The language should support PEMDAS operations.	To implement PEMDAS we should implement the following order of operations: 1. ( ) ! not 2. * / % 3. + - 4. ==, !=, >, <, <=, >= 5. &&, and, or , || The symbols **logical_expr**, **compare_expr**, **addition_expr**, **multiplication_expr**, **factor** wraps each other and so implements PEMDAS order of operation.

Lexer & Tokens

Assumption	Implementation
The source of the code should be abstracted from the Lexer to allow it to work with different sources	The Lexer class instance is initialized with a string containing all the code we want to tokenize. It cannot be extended, replaced after initialization to prevent unexpected behavior.
The Lexer should decouple the input text from the parser.	The lexer identifies a basic set of characters and converts them into tokens with the "whole" value and their token type so the parser will be able to function as expected
The language should support one-line comments only	if the # character is identified – skip the next text till the next new line (\n).

Parser & AST

Assumption	Implementation
Since the AST is decoupled from the lexer's input string - each node should include a reference the token used during the creation of that node.	For nodes that represent a specific operation (binary operation, declaration, call) we keep track of a token node that can give us added value during the semantic analysis and interpretation process.
The Parser should verify that the input token sequence follows the BNF structure.	The Parser's eat() function takes a TokenType and verifies that the current token we "parsed" was of the expected type by the BNF, else we throw a parsing error.
Since the BNF is hard to follow by the human eye, the parser should implement the BNF parsing in a way that is easy to maintain.	Each BNF symbol is represented by a function that parses exactly what the symbol is comprised of and calls other functions in case the symbol has other symbols within.
A nested lambda is a lambda declaration that is called immediately with a list of actual parameters. The NestedLambda AST node should not be a duplicate of the Lambda AST node's code but an extension of it.	The NestedLambda stores a lambda declaration node and the list of actual parameters in it.

Semantic Analyzer & Symbols

Assumption	Implementation
To manage and validate scopes, we should use an Abstract Data Type (ADT) table to keep track of functions and parameters	The ScopedSymbolsTable class allows to manage scopes, symbols and support recursive symbol lookup to help validating parameters and functions.
The analyzer should verify that all parameters (inside functions) and function calls can be resolved during evaluation i.e., they were declared beforehand.	Using the ScopedSymbolsTable class, the analyzer traverses the different nodes of the AST; creating / closing new scopes every time a function call is made and added formal parameters to the current scope when a function declaration is made.

Interpreter

Assumption	Implementation
While named-function declaration have a one-to-one sort of relationship between call and declaration - lambdas don't have such relationship	Lambda resolution is being done during the evaluation phase, meaning we might get an interpreter error if a formal parameter is treated as a function call, while it's not a one.
The make the interpreter's output test-able we need it to return one output at a time instead of printing it to the screen / returning everything as a chunk.	The Interpreter utilizes a generator approach with yield from and yield to evaluate each line, one by one.

Errors

Assumption	Implementation
Errors should specify which phase of the interpretation process failed with a category and a fitting string.	Each phase has it's own custom error with an error code, specific token and a message option.

CLI file

Assumption	Implementation
The interpreter should be executed in two modes: prompt (REPL) or parse which parses a file	Using argparse module, the code is divided to two functions for the different modes each managed a different aspect.
In the REPL mode, the session should be treated as one big program where lines are added one after another	The REPL function stores a root AST node with no statements and a persistent semantic analyzer. Each time the user submits a new code the statements are saved to the root AST node, replacing the previous nodes. In addition the semantic analyzer makes sure the global scope is persistent between instructions to allow for function calls.
Function declaration is defined be a two-line process: first for configuration and second is the boy.	When a function declaration is provided to the REPL - it should wait for the 2nd line before passing it to the parser.

Challenges and Implementation

File	Challenge	Solution
token.py	Aside from mapping all valid characters as token types, we need to be able to aggregate common token types	In the token.py file there's a function named "_build_keywords_dictionary" which takes an enum class, and a range of values within that enum class and creates a dictionaries where the key is enum value and the value is the token type. This function is useful to build lists with all characters / symbols of a specific type like, arithemetic operations, comparison operations etc.
lexer.py	When an error arises, how can we tell where in the file it happened?	the Lexer keeps track of it's location within the file using self.lineo (lines) and self.column (the index of the character within a line). It is then appended to the thrown error.
lexer.py	Some tokens rely on the next character to define the token type (e.g. param vs function call)	The Lexer has the peek() function that returns the next character in the input string without consuming it. That way we can differentiate between identifiers such as param and a function call (x vs x())
lexer.py	Since the parser is decoupled from the lexer's input string - we should have the peek option for tokens too	The Lexer has the peek_next_token(n) function that returns the n-th next token without consuming it.
lexer.py	some "words" of the language require more than 1 character, e.g. multi-digit numbers or keywords	The Lexer has __get_multichar_by_confition() function that takes a lambda / function as a stop condition and "builds" the multi-character value from the input string. An example of a stop condition would be "isdigit()" or "isalnum()"
parser.py	Function declaration configuration's order shouldn't matter.	To allow for all order options, during the parsing of a function declaration - we store the required fields we found from tokens in a dictionary and verify they are all populated at the end of the declaration parsing process.

parser.py	To parse expressions correctly we need to identify the different operators by groups.	Using the token.py's function _build_keywords_dictionary we built global variables that store operators for logical, comparison, addition and multiplication together for ease of use.
parser.py	Comparison operation should not be chained.	To prevent chaining of comparison operations such as: 1 == 2 == 3 We didn't use a loop in the compare_expr() function like we did with the other expr functions
semantic_analyzer.py	Function calls with a lambda declaration should link the parameter to the lambda symbol correctly.	While functions have a one-to-one relationship between their name and their function calls, lambdas are anonymous and different lambdas can be passed to the same formal parameter. In order to handle that, during the function call verification phase, we adjust the lambda_name member in the the lambda AST node to match the formal parameter name. That way the lambda's name inside the function's scope is the same as the parameter and it can be executed correctly.
Interpreter.py	When a new activation record is added to the call stack - we should be able to access variables from previous stacks	Each activation record (AR) has an update() method that adds members from an old AR to the new AR. This method is used when creating a new AR.

References

- Let's build A Simple Interpreter, https://ruslanspivak.com/lsbasi-part1
- Abstract Data Types, https://www.geeksforgeeks.org/abstract-data-types/
- Abstract Syntax Tree, https://www.geeksforgeeks.org/abstract-syntax-tree-ast-in-java/
- Tree traversal techniques, https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/