final project- our language

submit:

Iris Grabov 325948875

Daniel Berenshteyn 211993616

Documentation:

We divided the work into 3 main parts: Lexer, Presser and Interpreter

Each part is in a separate file corresponding to its name

Lexer

Purpose

The **lexer** (or lexical analyzer) converts the raw input text into a sequence of tokens.

Responsibilities

- **Tokenization**: Breaks the input text into meaningful tokens.
- **Whitespace and Comment Handling**: Skips over spaces and comments that are not relevant for parsing.
- **Error Detection**: Identifies any invalid tokens.

Example

Input text: 5 + 5

Tokens produced:

1. Integer: 5
2. Operator: +
3. Integer: 5

Parser

Purpose

The **parser** takes the sequence of tokens from the lexer and organizes them into a structured format, such as an Abstract Syntax Tree (AST).

Responsibilities

- **Syntax Analysis**: Ensures the tokens follow the grammatical rules of the language.
- **AST Construction**: Creates a hierarchical tree structure representing the syntactic relationships.
- **Error Reporting**: Detects syntax errors.

Example

Tokens: [Integer: 5, Operator: +, Integer: 5]

AST:

- **Binary Operation Node** (representing the addition operation)
    - **Left Operand**: 5 (Integer)
    - **Right Operand**: 5 (Integer)

Interpreter

Purpose

The **interpreter** executes the program by traversing the AST or intermediate representation, performing the operations defined by the program.

Responsibilities

- **Evaluation**: Computes the result based on the AST.
- **Runtime Error Handling**: Reports errors that occur during execution.

Example

AST: A Binary Operation Node representing 5 + 5.

Execution:

- Evaluates 5 + 5 to get 10.
- Outputs the result 10.

Summary

- **Lexer**: Converts raw text (5 + 5) into tokens ([Integer: 5, Operator: +, Integer: 5]).
- **Parser**: Constructs an AST representing the addition operation.
- **Interpreter**: Executes the AST and calculates the result (10).

BNF Grammar Documentation

This document describes the BNF (Backus-Naur Form) grammar for a programming language. The language supports basic arithmetic operations, logical operations, function definitions, and lambda expressions.

BNF Grammar

1. Expressions (<expr>)

   <expr> ::= <comp-expr>
   | <comp-expr> KEYWORD:AND <expr>
   | <comp-expr> KEYWORD:OR <expr>

   Description: An expression can be a comparison expression (<comp-expr>), or a logical combination of comparison expressions using AND or OR.

2. Comparison Expressions (<comp-expr>)
   <comp-expr> ::= "NOT" <comp-expr>
   | <arith-expr> ("==" | "<" | ">" | "<=" | ">=") <comp-expr>
   | <arith-expr>

Description: A comparison expression can be a negated comparison (NOT <comp-expr>), a comparison between arithmetic expressions using relational operators (==, <, >, <=, >=), or a simple arithmetic expression.

3. Arithmetic Expressions (<arith-expr>)
   <arith-expr> ::= <term>
   | <term> "+" <arith-expr>
   | <term> "-" <arith-expr>
   Description: An arithmetic expression consists of terms combined using addition (+) or subtraction (-).

4. Terms (<term>)
   <term> ::= <factor>
   | <factor> ("*" | "/" | "%") <term>
   Description: A term consists of factors combined using multiplication (*), division (/), or modulus (%).

5. Factors (<factor>)
   <factor> ::= <call>
   | "+" <factor>
   | "-" <factor>
   Description: A factor can be a call (<call>), or a unary plus (+) or minus (-) applied to another factor.

6. Calls (<call>)
   <call> ::= <atom>
   | <atom> "(" <arg_list> ")"
   Description: A call consists of an atom followed optionally by arguments enclosed in parentheses.

7. Argument List (<arg_list>)
   <arg_list> ::= <expr>
   | <expr> "," <arg_list>
   Description: An argument list consists of one or more expressions separated by commas.

8. Atoms (<atom>)
   <atom> ::= IDENTIFIER
   | <INT>
   | <BOOLEAN>
   | "(" <expr> ")"
   | <func-def>
   | <lambda-expr>

Description: An atom can be an identifier, integer, boolean, parenthesized expression, function definition, or lambda expression.

9. Function Definitions (<func-def>)
   <func-def> ::= KEYWORD:Defun IDENTIFIER? "(" (IDENTIFIER ("," IDENTIFIER)*)? ")" "->" <expr>
   Description: A function definition starts with Defun, followed by an optional function name, a list of parameters enclosed in parentheses, and a body expression after ->.

10. Lambda Expressions (<lambda-expr>)
    <lambda-expr> ::= KEYWORD:Lambda <params> "." "(" <expr> ")"
    Description: A lambda expression begins with Lambda, followed by parameters, a dot (.), and an expression enclosed in parentheses.

11. Parameters (<params>)
    <params> ::= <IDENTIFIER>
    | <IDENTIFIER> "," <params>
    Description: A list of parameters, where each parameter is an identifier.

12. Identifiers (<IDENTIFIER>) Booleans (<BOOLEAN>)
    <IDENTIFIER> ::= <LETTER> <identifier-tail>
    <identifier-tail> ::= <LETTER> <identifier-tail>
    | <INT> <identifier-tail>
    | ε
    <LETTER> ::= [A-Z] | [a-z]
    <INT> ::= [0-9] | [0-9]<INT>

- **Description**: Identifiers start with a letter and can be followed by letters or digits. Booleans Represents boolean values.

Features of the Language

1. **Basic Arithmetic Operations:**
   - Supports fundamental arithmetic operations such as addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). This allows for arithmetic expressions involving numbers.
2. **Logical Operations:**
   - Includes logical operations such as AND, OR, and NOT for building complex logical expressions. This enables the construction of boolean expressions and conditions.
3. **Comparison Operators:**
   - Provides relational operators like equality (==), less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=). These are used for comparing values and creating comparison expressions.
4. **Function Definitions:**

- Allows the definition of functions using the Defun keyword. Functions can have optional names, parameters, and return an expression. This enables modular and reusable code.

5. **Lambda Expressions:**
    - Supports anonymous functions through lambda expressions using the Lambda keyword. Lambda expressions allow for creating functions without naming them, useful for functional programming patterns.
6. **Function Calls and Recursion:**
    - Facilitates calling functions with arguments. Function calls can include parentheses and argument lists, allowing for the execution of functions with specific parameters.
7. **Identifiers:**
    - Uses identifiers to name functions, and parameters. Identifiers can include letters and digits, providing flexibility in naming conventions.
8. **Boolean Values:**
    - Includes boolean literals TRUE and FALSE, supporting boolean logic and conditional operations within expressions.

Summary of Features

- Arithmetic Operations: +, -, *, /, %
- Logical Operations: AND, OR, NOT
- Comparison Operators: ==, <, >, <=, >=
- Function Definitions: Defun keyword for named functions
- Lambda Expressions: Lambda keyword for anonymous functions
- Boolean Values: TRUE, FALSE

These features collectively make the language capable of expressing a range of programming constructs, from basic arithmetic to complex logical and functional programming tasks.

Trade-offs and Limitations

Trade-offs

1. **Simplicity vs. Expressiveness**:
    - The language is designed to be simple. However, this simplicity may limit expressiveness compared to more feature-rich languages.
2. **Recursive Parsing**:

- The grammar allows for recursive parsing, which is useful for representing nested expressions and function calls. However, it may be more complex to implement parsers and interpreters.

Limitations

1. **Limited Built-in Functions**:
    - The language provides basic arithmetic and logical operations but lacks built-in functions or libraries for more advanced operations.
2. **No Built-in Support for Mutable State**:
    - The language does not support mutable state or side effects, limiting its ability to model more complex behavior or interact with external systems.
3. **Basic Error Reporting**:
    - Error reporting is basic and primarily focused on syntax errors. More detailed error reporting and debugging features would need to be added for practical use.

examples: there is more examples in file: test.lambda

```
Defun power (base,exp)-> exp==0 OR base * power(base, exp - 1)

power(3,2)

-5/5+6*6+(5+6)*2

Lambda x,y,z . ((x+y)/z) (5,5,1)

Lambda x,y,z . (x+y+z) (1,2,3)

Defun factorial(n)-> n==0 OR n*factorial(n-1)

factorial(5)
```

User Guide for Interpreter Execution

1. File Mode: Execute code from a file with a .lambda suffix.
2. Interactive Mode: Execute code entered directly by the user.

Mode Selection

File Mode

1. Choosing File Mode:
    - When prompted with "Do you want to execute a test file? [Y/N]", enter Y to select file mode.
2. Provide Filename:
    - You will be prompted to enter the name of a file with a .lambda suffix or type 'exit' to quit.
    - Enter a valid filename with a .lambda extension, or type 'exit' to exit file mode.
3. File Validation:
    - Ensure the file name ends with .lambda. If not, you will be informed of the error, and you should enter a correct filename.
4. File Processing:
    - The application will attempt to read the file line by line.
    - If the file is successfully processed, you will return to the initial prompt. If the file is not found, you will be prompted to enter the filename again.
5. Error Handling:
    - If a file is not found or an error occurs during processing, you will be notified with an error message. You can then provide a new file name or choose to exit.

Interactive Mode

1. Choosing Interactive Mode:
    - When prompted with "Do you want to execute a test file? [Y/N]", enter N to select interactive mode.
2. Enter Code:
    - After selecting interactive mode, you will be prompted with execute >.
    - Enter your code or expressions directly into this prompt.
3. Code Execution:
    - Each line of code you enter is processed by the interpreter.
    - The result or error for each line is displayed immediately.
4. Error Handling:

- If there are syntax or execution errors in your code, they will be displayed in the console. You can then correct your input and continue.

Exiting the Application

- In File Mode: You can exit by typing 'exit' when prompted for a filename.

Comments

In the test file you can comments by using : #.....

how to execute "while loop"

To demonstrate how to simulate a while loop using recursion in the language defined by your BNF, you can use the Defun function to define a factorial function, which inherently uses recursion to iterate over values. This approach showcases how recursion can be used to achieve loop-like behavior.

Example: Factorial Function

BNF for Factorial Function

<func-def> ::= KEYWORD:Defun IDENTIFIER? "(" (IDENTIFIER ("," IDENTIFIER)*)? ")" "->" <expr>

Factorial Function Definition

define the factorial function as follows:

Defun factorial(n) -> n==0 OR n * factorial(n-1)

Explanation

- **Function Definition**:
  - Defun factorial(n) defines a function named factorial that takes one parameter n.
  - The function body is specified after ->.
- **Recursive Case**:
  - n * factorial(n-1) represents the recursive case. Here, factorial(n-1) is a recursive call to the factorial function with n-1 as the argument.
  - This is where the recursion occurs, effectively simulating a loop by repeatedly calling itself with a decremented value until the base case is reached.
- **Base Case**:
  - n==0 represents the base case of the recursion. If n is 0, the function returns 1 (since 0! is 1).
  - The OR operator is used to choose between the base case and the recursive case.

How It Simulates a While Loop

In traditional imperative programming, a while loop repeatedly executes a block of code as long as a condition is true. In the factorial function:

- **Simulating the Loop**:
    - The while loop is simulated by the recursive calls. Each call to factorial decreases the value of n until it reaches 0, analogous to decrementing a counter in a while loop.
    - The recursion continues (similar to continuing the loop) until the base case is met (like checking the loop condition).

Example Execution

Here's how the factorial function would be executed for factorial(3):

1. factorial(3) checks if n==0. It is not, so it evaluates 3 * factorial(2).
2. factorial(2) checks if n==0. It is not, so it evaluates 2 * factorial(1).
3. factorial(1) checks if n==0. It is not, so it evaluates 1 * factorial(0).
4. factorial(0) checks if n==0. It is, so it returns 1.

The recursive calls then return as follows:

- factorial(1) returns 1 * 1 = 1.
- factorial(2) returns 2 * 1 = 2.
- factorial(3) returns 3 * 2 = 6.

Thus, factorial(3) evaluates to 6.

Report: Design Decisions, Challenges, and Solutions for the Interpreter

Introduction

Design Decisions

Language Features

- **Grammar**: Defined using BNF to handle expressions, functions, and lambda functions. Supports recursion for flexible coding.
- **Expression Types**: Includes arithmetic, comparison, and logical operations, with recursive structures to handle complex calculations.

Implementation Choices

- **Lexer and Parser**: Uses recursive descent parsing to manage BNF rules and nested expressions. Lexer converts text into tokens.
- **Error Handling**: Provides detailed error messages for syntax and runtime issues.
- **Execution Model**: The interpreter processes the AST, handling and function calls.

Challenges and Solution:

The main challenge:

- **Challenge:** This is the first time we have been asked to write a language and this is the first experience for both of us in the Python language in general.
- **Solution:** We went through all the explanatory files in an orderly manner, we went through the presentation we studied in class, we went through the recording of the lesson in which there was an explanation of the final project and we used the chat and the YouTube videos.

Tokenization and Parsing

- **Challenge**: Accurate tokenization and parsing of complex expressions.
- **Solution**: Developed a comprehensive lexer and parser for handling various token types and nested constructs.

Error Detection and Reporting

- **Challenge**: Providing meaningful error messages for different issues.
- **Solution**: Custom error classes and detailed messages for syntax and runtime errors.

Lambda Expressions

- **Challenge**: Supporting anonymous functions and proper scope handling.
- **Solution**: Extended the parser and interpreter to handle lambda expressions and their execution context.

Conclusion

The interpreter's design involved careful consideration of recursion, error handling, and expression parsing. The solutions implemented ensure robust handling of complex features like functions and lambda expressions, providing clear feedback and effective execution.

part B: the solution- the python file is additionally added in the GitHub

```
#question 1

Fibonacci = lambda n: (lambda f: f(f, n))(lambda self, n, a=0, b=1: [a] + self(self, n-1, b, a+b) if n > 1 else [a])


print(Fibonacci(8))

#for x in :

#print(x)


lambda x,y : x*y


#question 2

concat_strings = lambda lst: lst[0] + ' ' + concat_strings(lst[1:]) if len(lst) > 1 else lst[0]

result = concat_strings([ "this", "is", "our", "final", "project"])

print(result)




#question 3

from functools import reduce

#cumulative_sum_of_squares = lambda lst: list(map(lambda sublist: reduce(lambda acc, x: acc + (x**2 if x % 2 == 0 else 0), sublist, 0), lst))


#cumulative_sum_of_squares = lambda lst: [reduce(lambda acc, x: acc + (x**2 if x % 2 == 0 else 0), sublist, 0) for sublist in lst]



cumulative_sum_of_squares = lambda lst: (

   lambda f: f(f, lst)

)(

   lambda self, lst: (
```

```
    lambda map_reduce: list(map(map_reduce, lst))
  )(
    lambda sublist: (
      reduce(
        lambda acc, x: acc + (x**2 if x % 2 == 0 else 0),
        sublist,
        0
      )
    )
  )
)

# Example usage
input_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9, 10]]
result = cumulative_sum_of_squares(input_list)
print(result)



#question 4
def cumulative_op(binary_op):
 return lambda sequence: reduce(binary_op, sequence)



#cumulative_op= lambda binary_op:  lambda sequence: reduce(binary_op, sequence)

# Example usage
# Factorial implementation
factorial = cumulative_op(lambda x, y: x * y)

# Exponentiation implementation
exponentiation = cumulative_op(lambda x, y: x ** y)
```

```
# Test cases
print(factorial(range(1, 6)))
print(exponentiation([2, 3, 2]))
```

```
#question 5

nums = [1, 2, 3, 4, 5, 6]
sum_squared = reduce(lambda acc, x: acc + x, map(lambda x: x**2, filter(lambda num: num % 2 == 0, nums)))
print(sum_squared)
```

```
#question 6
count_palindromes = lambda lst: list(map(lambda sublist: reduce(lambda acc, s: acc + (s == s[::-1]), sublist, 0), lst))

# Example usage
input_list = [["racecar", "hello", "level"], ["madam", "world"], ["noon", "radar", "python"]]
print(count_palindromes(input_list))
```

```
#question 7
def generate_values():
 print('Generating values...')
 yield 1
```

```
 yield 2
 yield 3


def square(x):
 print(f'Squaring {x}')
 return x * x



print('Eager evaluation:')
values = list(generate_values())
squared_values = [square(x) for x in values]
print(squared_values)


#הפונקציה generate_values נקראת ותוצאותיה נאספות לרשימה באמצעות list(generate_values()).
#  זה מאלץ יצירה של כל הערכים מראש, כלומר יש הקצאת זכרון לכל הערכים ברשימה לפני שמשתמשים בהם בפונקציה הבאה.


print('\nLazy evaluation:')
squared_values = [square(x) for x in generate_values()]
print(squared_values)
# בשיטת Lazy evaluation
# הערכים לא מחושבים עד שהם באמת נדרשים. כאן generate_values,
# לא מחשבת את כל הערכים מראש אלא מייצרת אותם אחד אחרי השני
# מה שמוביל לחיסכון פוטנציאלי בזיכרון ובחישוב.



#question 8


primes_descending = lambda lst: sorted([x for x in lst if all(x % i != 0 for i in range(2, int(x**0.5) + 1)) and x > 1], reverse=True)
numbers = [10, 15, 2, 7, 19, 23]
```

```
print(primes_descending(numbers))
```