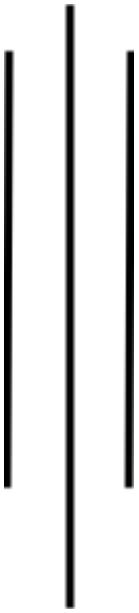
Final Project Report

Programming language Project



Date: 05/05/2024

Submitted To
Prof. Phan Minh Dung

Submitted By:
Ritik Sareen [st123844]
Jannatun Nayeem [st124438]
Pheakdey Tum [st124386]

The Grammar of the Programming language:-

grammar.txt

```
expr                    : KEYWORD:VAR IDENTIFIER EQ expr
                        : comp-expr ((KEYWORD:AND|KEYWORD:OR) comp-expr)*

comp-expr               : NOT comp-expr
                        : arith-expr ((EE|LT|GT|LTE|GTE) arith-expr)*

arith-expr              :term ((PLUS|MINUS) term)*

term                    : factor ((MUL|DIV) factor)*

factor                  : (PLUS|MINUS) factor
                        : power

power                   : atom (POW factor)*

atom                    : INT|FLOAT|IDENTIFIER
                        : LPAREN expr RPAREN
                        : if-expr
                        : for-expr
                        : while-expr

if-expr                 : KEYWORD:IF expr KEYWORD:THEN expr
                          (KEYWORD:ELIF expr KEYWORD:THEN expr)*
                          (KEYWORD:ELSE expr)?

for-expr                : KEYWORD:FOR IDENTIFIER EQ expr KEYWORD:TO expr
                          (KEYWORD:STEP expr)? KEYWORD:THEN expr

while-expr  : KEYWORD:WHILE expr KEYWORD:THEN expr
```

About the Grammar:-

The grammar is for a programming language and defines the syntax rules for expressions, including arithmetic, comparison, and logical operations, as well as control structures like loops and conditional statements. Here's a summary:

1. **Expressions** (expr):
    - Basic expressions are either comparison expressions or can be chained with logical AND or OR operators.
2. **Comparison Expressions** (comp-expr):
    - Can be negated with NOT or involve arithmetic expressions compared using equality or inequality operators.
3. **Arithmetic Expressions** (arith-expr):
    - Consist of terms added or subtracted.
4. **Terms** (term):
    - Composed of factors that are multiplied or divided.
5. **Factors** (factor):
    - Can be negated or positive and are either powers or base arithmetic units.
6. **Powers** (power):
    - Base atoms potentially raised to the power of another factor.
7. **Atoms** (atom):
    - Can be integers, floats, identifiers (variables), expressions within parentheses, or control structures.
8. **Control Structures:**
    - if-expr: Conditional expressions with optional elif and else clauses.
    - for-expr: For loops with optional step expressions.
    - while-expr: While loops.

Each part of the grammar specifies how smaller components build up complex statements, supporting typical programming constructs like loops, conditions, and various operations.

Sections In the Program In basic.py:-

1. Constants
2. Errors
3. Position
4. Tokens
5. Lexer
6. Nodes
7. Parse Result
8. Parser
9. Run Time result
10. Values
11. Context
12. Symbol table
13. Interpreter
14. Run

What do these sections do?

1. **Constants:**
    - Defines commonly used character sets such as digits and letters, which are useful in tokenization and parsing.
2. **Errors:**
    - Defines various error classes like `IllegalCharError`, `ExpectedCharError`, `InvalidSyntaxError`, and `RTError` to handle different error states throughout the parsing and execution processes.
3. **Position:**
    - Manages the position within the source code being parsed, tracking indices, lines, columns, file names, and file text. This is useful for error reporting and debugging.
4. **Tokens:**
    - Defines token types and the `Token` class to encapsulate properties of tokens identified during the lexing process. Tokens are the basic building blocks parsed from the input code.

5. **Lexer:**
    - Transforms a string of characters into a list of tokens. The lexer handles the initial stage of parsing by recognizing patterns in the input string and categorizing them as different token types.
6. **Nodes:**
    - Represents various types of syntax tree nodes like NumberNode, BinOpNode, and IfNode, which are used to build an Abstract Syntax Tree (AST) during parsing.
7. **Parse Result:**
    - Encapsulates the result of parsing operations, including any nodes generated or errors encountered. This helps in managing the state of the parse and propagating errors.
8. **Parser:**
    - Parses a list of tokens into an AST based on the grammar rules defined. The parser handles expressions, control structures, and operations by creating appropriate nodes in the AST.
9. **Run Time Result:**
    - Similar to Parse Result, but used during the execution phase to manage values and errors as code is interpreted.

10. **Values:**
    - Defines a Number class that encapsulates numerical values and provides methods for arithmetic and logical operations, including handling errors like division by zero.
11. **Context:**
    - Manages the context in which code runs, including parent-child relationships and symbol tables, which store variable values.
12. **Symbol Table:**
    - Manages variables and their values. Provides methods to set, get, and delete variable entries, supporting scope and nested expressions.
13. **Interpreter:**
    - Visits nodes in the AST to execute the program. It performs operations, evaluates expressions, and manages control flow based on the node types and the context.
14. **Run:**
    - The entry point for running the interpreter. It integrates the lexer, parser, and interpreter to process input text, handle errors, and execute the parsed code. It also initializes the global symbol table with predefined values.

Use of Shell.py:-

The shell.py script serves as a simple command-line interface for the basic interpreter, which is developed in Python. Here's what each part of the script does:

1. **Importing the basic module:**
    - Starts by importing the basic module. This module contains the implementation of your interpreter, including the lexer, parser, and executor. It's necessary for shell.py to access the run function within this module.
2. **Infinite Loop:**
    - The script enters an infinite loop with while True:. This loop continues to execute until the program is explicitly stopped (e.g., through a break statement or terminating the program with a keyboard interrupt like Ctrl+C).
3. **Taking User Input:**
    - Inside the loop, the script prompts the user for input using input('basic > '). This displays basic > as a prompt in the terminal, and waits for the user to enter a line of text. This text is expected to be code written in the language that the basic interpreter can understand.
4. **Running the Interpreter:**
    - The text entered by the user is then passed to the basic.run('<stdin>', text) function. Here, '<stdin>' is used as a placeholder to indicate that the source of the text is standard input (i.e., the terminal), not a file.
    - The run function processes this input text, presumably tokenizing, parsing, and executing it according to the rules and logic defined in the basic module.
5. **Handling Errors and Output:**
    - The run function returns two values: result and error. If an error occurs during the processing of the input (e.g., syntax error, runtime error), the error object is returned. If the code is executed successfully, the result of the code execution is returned.
    - The script checks if an error is returned. If so, it prints the error message using error.as_string(). Otherwise, it prints the result.

Use of string_with_arrow.py File:-

The function string_with_arrows provided in your code snippet is designed to visually indicate a specific section of text by adding arrows (^) beneath the text. This is typically used to highlight parts of the text where errors have occurred, making it easier for users to locate and understand the exact location and context of an error in their code. Here's how it works step by step:

1. **Calculate Start and End Indices:**
    - It first calculates the start (idx_start) and end (idx_end) indices of the line that contains pos_start.idx, the start position index. This is done by searching for newline characters (\n) in the text.
    - idx_start is determined by finding the last newline character before pos_start.idx. If none is found, it defaults to the start of the text (0).
    - idx_end is the position of the first newline character after idx_start. If no newline is found, it defaults to the end of the text.
2. **Generate Each Line:**
    - The function then determines the number of lines (line_count) affected by the range from pos_start to pos_end. This involves computing the difference in line numbers between the start and end positions.
    - For each line within this range, it slices the text from idx_start to idx_end to isolate the line of text where the error is to be indicated.
3. **Calculate Column Start and End:**
    - For the first line, the column start (col_start) is pos_start.col. For subsequent lines, it starts from the beginning of the line (0).
    - For the last line in the range, the column end (col_end) is pos_end.col. For all other lines, it extends to the end of the line.
4. **Append Arrows to Result:**
    - Each line of text is appended to result followed by a newline character.
    - A string of spaces is added to align the start of the arrows with col_start, followed by a series of arrows (^). The number of arrows spans from col_start to col_end to underline the specific section of the text.
    - The indices (idx_start and idx_end) are updated to move to the next line, if any.
5. **Handle Tabs:**
    - Finally, before returning result, all tab characters (\t) are replaced with empty spaces to avoid misalignment in environments where tab sizes may differ.