SNOL

Documentation & Manual

Apostol, Lada, Tampugao, Salcedo

June 2024

Table of Contents

I. Documentation

Introduction

This language is a Simple Number-Only Language (SNOL) which is a simplified custom programming language which only involves integer, real values, operations, and expressions. This is in partial fulfillment of the CSMC 124 course.

The Group

Tyrael Lada led the development of the SNOL programming language. Chris Samuel Salcedo was responsible for the testing and edge casing of the program. Documentation, presentation, and demonstration were worked on by Chris Samuel Salcedo, Danica Apostol and Mohammad Muraya Tampugao.

Main Program

The main.py file is the entry point of the SNOL interpreter. It imports the necessary modules and defines two main functions: interpret and eval_loop.

1. **interpret function:** This function takes a command string and an environment as input. The environment is a dictionary that maps variable names to their current values. The function tokenizes the command using the lexer, parses the tokens into an Abstract Syntax Tree (AST) using the parser, and then evaluates the AST using the evaluator. If any of these steps raise an error, the function catches it and prints an error message.

2. **eval_loop function:** This function is the main loop of the interpreter. It creates an empty environment and then repeatedly asks the user for a command, interprets the command using the interpret function, and prints the result. The loop continues until the user enters an exit command.

Interfaces

The provided Python code defines some interfaces and classes that are used in a parser or interpreter for a custom language. Here's a breakdown of the code:

Node class: This class represents a node in an abstract syntax tree (AST). Each node has a value, a type, and potentially some children nodes. The __str__ method is overridden to provide a custom string representation of the node, and the __eq__ method is overridden to provide a custom equality check.

Error class: This is a simple custom exception class that takes a message as an argument.

Environment TypeAlias: This is a type alias for a dictionary that maps strings to either integers or floats. This is likely used to represent the environment in which the code is executed, i.e., the current state of all variables.

Token TypeAlias: This is a type alias for a tuple of two strings. This is likely used to represent a token, where the first string is the type of the token and the second string is the value of the token.

Lexer

The lexer is implemented in Python and it's used to break down a command into a list of tokens. Here's a step-by-step breakdown:

1. The **lexer function** takes a command as input. This command is a string that represents the code to be tokenized.

2. It defines a regular expression pattern for each type of token it can recognize. These types include *numbers, variables, operators*, and a few specific keywords like *"BEG", "PRINT", and "EXIT!"*.

3. It uses the **re.findall function** to find all tokens in the command that match any of the defined patterns. The patterns are joined together with the "|" operator, which in regular expressions

means "or". This way, re.findall will match any part of the command that looks like any of the defined token types.

4. The list of matched tokens is then passed to the _tokenize function.

5. The **_tokenize** function goes through each token and determines its type. It does this by trying to match the token against each pattern again, this time separately. If a match is found, it returns a tuple with the type of the token and the token itself.

6. If no match is found for a token, it raises an Error.

7. Finally, it appends an "EOF" (End Of File) token to the list. This is a common practice in lexers and parsers, and it's used to signal that there are no more tokens left.

8. The function returns the list of tokenized tokens. Each token is represented as a tuple, where the first element is the type of the token and the second element is the token itself.

Parser

The parser in the provided Python code is implemented using a recursive descent parsing strategy. Here's a step-by-step breakdown:

1. The **parser function** is the entry point. It takes a list of tokens as input and returns an Abstract Syntax Tree (AST) that represents the parsed command. It does this by calling the _parse_command function.

2. The _parse_command function tries to parse the command as an assignment, output, exit, or expression, in that order. If none of these succeed, it raises an Error.

3. The *_parse_assignment, _parse_output, _parse_exit, and _parse_expression* functions each try to parse a specific type of command. If the command does not match the expected format, they return None.

4. The _parse_expression function parses an expression, which is defined as a term followed by zero or more pairs of a precedence 1 operator and a term. It does this by first parsing a term and then entering a loop where it parses the operator and the next term. The parsed terms and operators are combined into a tree structure.

5. The _parse_term function parses a term, which is defined as a factor followed by zero or more pairs of a precedence 2 operator and a factor. It works similarly to the _parse_expression function.

6. The _parse_factor function parses a factor, which can be a number, a variable, or an expression in parentheses.

7. The _parse_assignment function parses an assignment, which is a variable followed by an equals sign and an expression.

8. The _parse_output function parses an output command, which is the keyword "PRINT" followed by a number or a variable.

9. The _parse_exit function parses an exit command, which is the keyword "EXIT!".

10. Each of these functions modify the list of tokens in-place by removing the tokens they have parsed. This is why they don't need to return the remaining tokens.

11. Parsed commands, expressions, terms, and factors are shown as nodes in an Abstract Syntax Tree (AST). Each node has a type, a value, and a list of children nodes. The type tells what kind of node it is (like "EXPRESSION", "TERM", "FACTOR", "ASSIGNMENT", "OUTPUT", or "EXIT"). The value is the specific content (like an operator, a variable, or a number). The children are the smaller parts that make up the node, like the terms and operators in an expression, or the parts of an assignment or output.

Evaluator

The evaluator in the provided Python code is implemented as a function that takes an Abstract Syntax Tree (AST) and an environment as input and returns the result of evaluating the AST in the given environment. Here's a step-by-step breakdown:

1. The evaluator function is the entry point. It uses a match statement to determine the type of the root node of the AST and calls the appropriate helper function to evaluate it.

2. The _evaluate_expression, _evaluate_term, and _evaluate_factor functions evaluate expressions, terms, and factors respectively. They do this by recursively evaluating their children nodes and applying the operator stored in their value field to the results.

3. The _evaluate_assignment function evaluates an assignment. It does this by evaluating the expression on the right-hand side of the assignment and storing the result in the variable on the left-hand side.

4. The _evaluate_output function evaluates an output command. It does this by printing the value of the variable or the number specified in the command.

5. The _evaluate_beg function evaluates a beg command. It does this by asking the user for a value and storing it in the specified variable.

6. The _evaluate_exit function evaluates an exit command. It does this by printing a message and exiting the program.

7. The environment is a dictionary that maps variable names to their current values. It's passed as an argument to the evaluator and to all helper functions, so they can access and modify the current state of the variables.

8. The AST (Abstract Syntax Tree) is a tree structure where each node stands for a command, an expression, a term, a factor, or a number. Each node has a type, a value, and a list of child nodes. The type is a label that says what kind of node it is (like

"EXPRESSION", "TERM", "FACTOR", "ASSIGNMENT", "OUTPUT", or "EXIT"). The value is the specific content of the node (such as an operator, a variable, or a number). The children are the smaller parts that make up the node, like the terms and operators in an expression, or the parts of an assignment or output command.

Testing

The tester.py file is a collection of unit tests for the lexer, parser, and evaluator functions of a programming language interpreter. It uses Python's built-in unittest module to define and run the tests. Here's a step-by-step breakdown:

1. The TestLexer class tests the lexer function. It has four test methods:

    a. test_lexer_with_numbers tests that the lexer can correctly tokenize numbers.
    b. test_lexer_with_keywords tests that the lexer can correctly tokenize keywords.
    c. test_lexer_with_operators tests that the lexer can correctly tokenize operators.
    d. test_lexer_with_invalid_token tests that the lexer raises an Error when it encounters an invalid token.

2. The TestParser class tests the parser function. It has five test methods:

    a. test_parse_assignment tests that the parser can correctly parse assignments.
    b. test_parse_output tests that the parser can correctly parse output commands.
    c. test_parse_exit tests that the parser can correctly parse exit commands.
    d. test_parse_expression tests that the parser can correctly parse expressions.
    e. test_parse_complex_expression tests that the parser can correctly parse complex expressions.

3. The TestEvaluator class tests the evaluator function. It has seven test methods:

    a. test_evaluate_expression tests that the evaluator can correctly evaluate expressions.
    b. test_evaluate_type_error tests that the evaluator raises an Error when it encounters a type error.

c. test_evaluate_term tests that the evaluator can correctly evaluate terms.
d. test_evaluate_assignment tests that the evaluator can correctly evaluate assignments.
e. test_evaluate_output tests that the evaluator can correctly evaluate output commands.
f. test_evaluate_beg tests that the evaluator can correctly evaluate beg commands.
g. test_evaluate_exit tests that the evaluator raises a SystemExit when it evaluates an exit command.

4. Each test method creates an input, calls the function under test with that input, and then checks that the function's output is as expected. If the output is not as expected, the test fails and an error message is printed.

5. The unittest.mock.patch function is used to mock the built-in input function in the test_evaluate_beg method. This allows the test to control the user's input.

6. The unittest.main() function is called when the file is run as a script. This function runs all the tests and prints the results.

II. Manual

Starting the SNOL

To launch the SNOL, use the **main.exe** executable file, a provided Python script. This script takes care of setting up all the necessary configurations and dependencies before starting the application. If for some reason you can't use the primary method, you have an alternative: the main.py . This script acts as a backup to get the SNOL executable running. Having both options ensures that you can start the SNOL application smoothly and reliably, no matter the situation.

Check python's documentation for more information: https://docs.python.org/3/using/index.html

Assigning Variables

1. Direct assignment: variable = expression.
   For example, x = 5 assigns the value 5 to the variable x.
2. Using the BEG keyword: BEG variable. This initializes a variable.
   For example, BEG x initializes the variable x.

Operations

The <expression>, <term>, and <factor> rules in the grammar allow you to perform arithmetic operations. Here's how you can use them:

1. Addition and subtraction: expression = term + term or expression = term - term.
   For example, x = 5 + 3 assigns the value 8 to the variable x.
2. Multiplication and division: term = factor * factor or term = factor / factor.
   For example, x = 5 * 3 assigns the value 15 to the variable x.
3. Parentheses: factor = (expression). This allows you to control the order of operations.
   For example, x = (5 + 3) * 2 assigns the value 16 to the variable x.
4. Negative numbers: The <int> rule in the grammar allows for an optional negative sign at the beginning. This means you can use negative numbers in your expressions.
   For example, x = -5 + 3 assigns the value -2 to the variable x.

5. Variable usage: Once a variable has been assigned a value, you can use it in other expressions. For example, if you have previously assigned x = 5, you can then assign y = x * 2 to assign the value 10 to the variable y.

Printing

The <output> rule in the grammar allows you to print the value of a variable or a number. Here's how you can use it:

Print a variable: PRINT variable. For example, PRINT x prints the value of the variable x.
Print a number: PRINT number. For example, PRINT 5 prints the number 5.

Exiting

The <EXIT!> rule in the grammar allows you to exit the program. You can do this by simply entering EXIT!.