**Programming Languages - Final Project**

**BNF Grammar**

We decided to use a prefix syntax. Based on what we learned in class, it is the simplest syntax we know to implement and handle complex expressions.

* Expressions can be arithmetic expressions, variable declarations, assignments, conditionals, or loops.
* Arithmetic expressions consist of operators (+, -, \*, /) and operands, which can be integers, variable names, or nested expressions.
* Variable declarations and assignments involve keywords ('declare', 'set'), variable names, and integer values.
* Conditional expressions consist of 'if', followed by a condition, a true branch, and a false branch, each of which can be expressions.
* Loop expressions involve 'while' keyword, followed by a condition, a true branch, and a false branch each of which can be expressions.

<expression> ::= <arithmetic\_expr> | <variable\_declaration> | <variable\_assignment> | <conditional\_expr> | <loop\_expr>  
  
<arithmetic\_expr> ::= '(' <operator> <operand> <operand> ')'  
<operand> ::= <integer> | <variable\_name> | <expression>  
<operator> ::= '+' | '-' | '\*' | '/'  
  
<variable\_declaration> ::= '(' 'declare' <variable\_name> <integer> ')'  
<variable\_assignment> ::= '(' 'set' <variable\_name> <integer> ')'  
  
<conditional\_expr> ::= '(' 'if' <condition> <true\_branch> <false\_branch> ')'  
<condition> ::= <expression>  
<true\_branch> ::= <expression>  
<false\_branch> ::= <expression>  
  
<loop\_expr> ::= '(' 'loop' <condition> <body> ')'  
<body> ::= <expression>  
  
<integer> ::= [0-9]+  
<variable\_name> ::= [a-zA-Z][a-zA-Z0-9\_]\*

**Design Decisions**

**Lexer:**

* The lexer takes an input string and tokenizes it based on regular expressions corresponding to different token types defined in the grammar.
* Each token is a tuple containing its type and value.

**Parser:**

* The parser takes the tokens produced by the lexer and builds a parse tree according to the grammar rules.
* It recursively processes tokens to construct the parse tree.

**Main:**

* the main function asks the user for a file path to read from.
* it tokenizes the input using the lexer, parse the tokens using the parser, and then interpret the resulting parse tree.
* The environment (**env**) keeps track of variable values during interpretation.

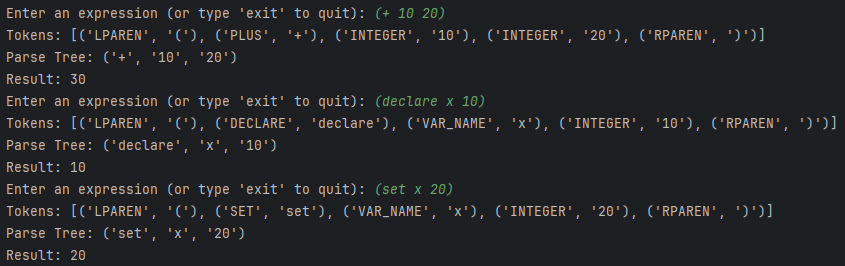
**Alter\_main:**

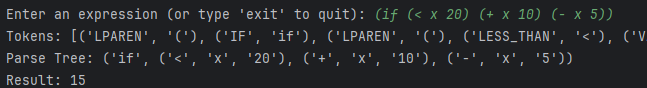
* An alternative main file
* The main function prompts the user asking for expressions.
* Easy way to check the language capabilities and understand the syntax.

**Evolution of the Language**

* Initially, the language was defined using a simple BNF grammar with basic arithmetic operations and variable manipulation.
* Then, additional features such as conditionals and loops were added, enhancing the language's expressiveness and capability.
* The implementations evolved to accommodate these new features, introducing design decisions and trade-offs to balance simplicity, efficiency, and correctness.

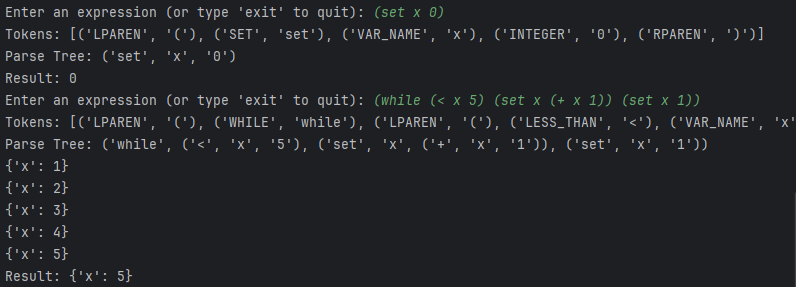
**Examples and Test Cases**

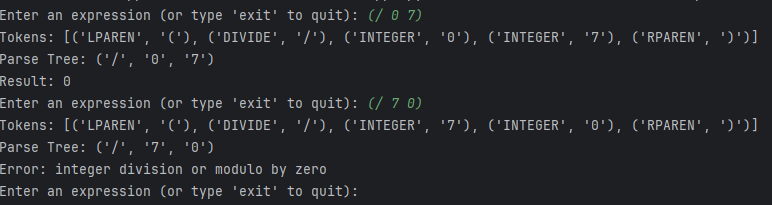


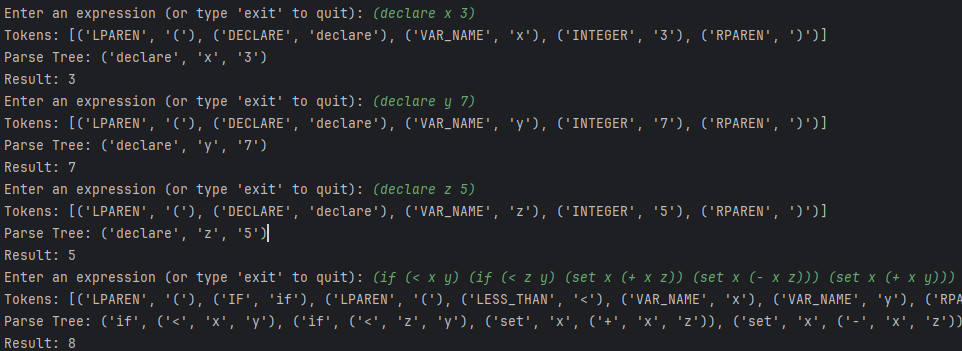


תמונה שמכילה טקסט, צילום מסך, גופן

התיאור נוצר באופן אוטומטי





****

**Instructions to Run and Compile the Code**

1. Save the interpreter code files (lexer.py, parser.py, interpret.py, main.py, and language\_specifications.py) in a directory.
2. Open a terminal or command prompt and navigate to the directory containing your code.
3. Run the main.py file using the Python interpreter:

Python main.py

1. Follow the on-screen instructions to enter a file path, make sure the file has 'exit' at the end.

**Explanation of Outputs:**

The interpreter will tokenize, parse, and interpret the input code expressions.

For each input code expression, the interpreter will display the intermediate tokenization and parsing results (if debug prints are enabled).

Finally, the interpreter will display the result of evaluating the code expression.

For example, for the test case (declare x 10), the output might be x = 10, indicating that the variable x has been declared with an initial value of 10.

The language we implemented in the interpreter is **dynamically typed**.

In a dynamically typed language, variable types are determined at runtime rather than compile time.

**Variable Declarations and Assignments:**

* It allows variable declarations using the **declare** keyword, but it doesn't specify the type explicitly. For example, **declare x 10** creates a variable **x** with the value **10**, but it doesn't specify whether **x** is an integer, float, or any other type.
* Similarly, variable assignments using the **set** keyword don't require specifying the type. For example, **set x 20** changes the value of variable **x** to **20**, regardless of its original type.

**Implicit Type Conversion:**

* the language doesn't enforce strict type checking. Operations such as addition (+), subtraction (-), multiplication (\*), and division (/) can be performed on variables without checking their types beforehand.
* We only support integers, but if we add more variable types- we could support **(+ 10.5 20)** and more operations between different types.

**Part B – Question 14**

Lazy evaluation and eager evaluation are two strategies used in programming to handle the evaluation of expressions. Lazy evaluation delays the evaluation of expressions until their results are actually needed, while eager evaluation evaluates expressions as soon as they are bound to a variable.

In practical terms, the program provided has a function called generate\_values() that produces a sequence of numbers, and another function called square(x) that squares a number.

In **eager evaluation**, we first generate all the numbers using generate\_values(), storing them in memory, and then square each of them. This approach consumes memory and computational resources upfront, even if not all generated values are ultimately needed.

In contrast, **lazy evaluation** defers the generation of values until they are required. Rather than generating and storing all numbers immediately, we generate each number only when it's needed, such as when it's about to be squared. This approach conserves resources, especially when dealing with large sequences or potentially infinite lists, as it avoids unnecessary computation and memory usage until absolutely necessary.