**Programming Languages**

**Final Project Report**

Gal Ogdan

Oron Mozes

**Github link: https://github.com/galogdan**

**Github direct link : https://github.com/galogdan/Basic-Python-Interpreter**

# **Running instructions:**

For the interactive mode just run the main without any args.

For file mode please run the following command in terminal:

“python main.py input\_file.txt” (the file name)

Make sure the input file is in the same folder as the python files.

# BNF Grammar

<program> ::= <block> | %empty

<block> ::= <statement> <block>

| <statement>

<statement> ::= <assignment\_statement>

| <if\_statement>

| <while\_statement>

| <expression>

<assignment\_statement> ::= <variable> "=" <expression>

<if\_statement> ::= "if" "(" <bool\_expression> ")" "{"<block>"}"

["else" "{" <block> "}"]

<while\_statement> ::= "while" "(" <bool\_expression> ")" "{" <block> "}"

<expression> ::= <term> ("+" <term> | "-" <term>)\*

| <bool\_expression>

<bool\_expression> ::= <term> (">" <term> | "<" <term> | "==" <term>)

<term> ::= <factor> ("\*" <factor> | "/" <factor>)\*

<factor> ::= <number>

| <variable>

| "(" <expression> ")"

| <factor>

<number> ::= [0-9]+ | [0-9]+.[0-9]+

<variable> ::= [a-zA-Z\_]+

### Syntax and Grammar Design:

* 1. Minimalistic Language: Minimalistic syntax with basic constructs such as if-else, while loops, and assignment statements.
  2. Arithmetic and Relational Expressions: The language supports basic arithmetic operations (addition, subtraction, multiplication, division) and relational operations (greater than, less than, equal to).
  3. Block Structure: The language allows block structures for if and while statements, where multiple statements can be grouped within curly braces ({}).
  4. Variable Handling: The language allows the use of variables for assignment and manipulation within expressions.

### Tokenization and Lexical Analysis:

* 1. Token Types: The language uses predefined token types such as arithmetic operators (PLUS, MINUS, MULTIPLY, DIVIDE), relational operators (GREATER, LESS, EQUAL), control flow keywords (IF, ELSE, WHILE), and others.
  2. Token Structure: Some Tokens must have types and semantic values as necessary, e.g., TokenType.NUMBER with a numeric value or TokenType.VARIABLE with a variable name.

### Parsing:

* 1. Recursive Descent Parsing: This method involves recursively breaking down the input based on the grammar rules.
  2. Error Handling: The parser includes error handling for invalid syntax through the raise\_error() method. This catches invalid tokens and structures, providing descriptive error messages.
  3. Lookahead Token: The parser uses a lookahead approach, keeping track of the current token and the next token, allowing for decision-making based on the next token's type.

### Control Flow Constructs:

* 1. If Statements: The language supports if-else constructs with a required condition and optional else block.
  2. While Loops: While loops are supported with a condition and a block of statements that can be repeated based on the condition.

### Expressions and Operations:

* 1. Precedence and Associativity: The language follows common mathematical precedents and associativity for arithmetic and relational expressions.
  2. Operator Types: Binary operations (addition, subtraction, multiplication, division) and relational operations (greater, less, equal) are supported.
  3. The language supports parentheses for grouping expressions.

### Variables and Assignment:

* 1. Variable Declarations: Variables are declared implicitly upon first use and are case-sensitive.
  2. Assignment Statements: Assignments use the = operator to assign a value to a variable.

### Code Structure:

* 1. Statements and Blocks: Code can be structured into statements and blocks of code. Blocks are enclosed within {} to denote scope and grouping.

### Others:

* 1. Type System: Dynamically typed with simple types (numbers).

# Trade-offs & Limitations

1. Lack of Advanced Features: The language does not support advanced language features such as classes or modules, which limits its expressiveness.
2. Performance Considerations: Recursive descent parsing can be slower compared to other parsing techniques.
3. No Type Inference: The language does not perform type inference (the numbers are float dynamically).

# Milestones and Language Evolution

The initial implementation focuses on basic constructs for basic arithmetic and relational expressions which are supported to enable logical control flow.

After implementing the basic arithmetic we implemented the more complex statements such as if-else and while.

As we added keywords and options the BNF grew accordingly from a basic BNF of expressions to a more complex one with variables and different conditions and expressions handling.

After implementing the statements (and block of statements) we could test the interpreter and as we expected: the Lexer raising the errors of Illegal characters , the Parser raising errors referring the syntax and the Interpreter for semantic issues such as undefined variables.

“4. Is your language is statically typed or dynamically typed?  
Explain in detail your choice.”

**Answer:** The language is dynamically typed. Here’s why:

Type Checking at Runtime: Variable are determined and checked at runtime, meaning that type information is associated with values during execution.

Also, there are no explicit type declarations for variables or function return types. Variables can hold values of “Any” type, and functions can return values of any type without declaring their return type beforehand.

**Answers for part B**

**Also in Questions.py with examples**

# This file is for part B of the project questions 9-14  
  
  
from functools import reduce  
  
################Question 9######################  
factorial = lambda n: reduce(lambda x, y: x \* y, range(1, n + 1), 1)  
print(factorial(5))  
  
  
################Question 10######################  
concatenate\_with\_space = lambda strings: reduce(lambda x, y: x + ' ' + y, strings)  
strings = ['check', 'if', 'its', 'working']  
single\_string = concatenate\_with\_space(strings)  
print(single\_string)  
  
################Question 11######################  
def cumulative\_sum\_of\_squares\_even\_numbers(lists):  
 return list(map(  
 lambda sublist: reduce(  
 lambda total, num: total + (  
 lambda even: (  
 lambda square: (  
 lambda add\_square: add\_square(num)  
 )(lambda n: square(n) if even(n) else 0)  
 )(lambda n: n \*\* 2)  
 )(lambda n: n % 2 == 0),  
 sublist,  
 0  
 ),  
 lists  
 ))  
  
  
lists\_of\_numbers\_to\_square = [  
 [2, 4, 6, 5, 8],  
 [1, 5, 10, 9],  
 [8, 11, 12, 4]  
]  
  
calculated\_square\_lists = cumulative\_sum\_of\_squares\_even\_numbers(lists\_of\_numbers\_to\_square)  
print(calculated\_square\_lists)  
  
  
###############Question 12#######################

nums = [1, 2, 3, 4, 5, 6]  
sum\_squared = reduce(lambda x, y: x + y, map(lambda x: x\*\*2, filter(lambda x: x % 2 == 0, nums)))  
print(sum\_squared)  
  
  
##############Question 13#######################  
  
count\_palindromes = lambda lists: list(map(lambda sublist: reduce(lambda count, string: count + 1 if string == string[::-1] else count, sublist, 0), lists))  
  
# Example usage  
lists\_of\_strings = [  
 ["racecar", "level", "hello", "madam"],  
 ["world", "noon", "deed", "test"],  
 ["python", "civic", "radar", "tree", "aca"]  
]  
  
num\_of\_palindroms = count\_palindromes(lists\_of\_strings)  
print(num\_of\_palindroms)  
  
  
##############Question 14#######################  
"""  
  
In this program we can see the differences between lazy and eager evaluation.  
While in the eager evaluation all the values generated and only after performing squared\_values = [square(x) for x in values],  
in the lazy evaluation the calculations occuring exactly when needed as we can see we perform square for each generated instead of generating all first.  
In that way the "lazy evaluation" delaying calculate the evaluations until needed.   
  
"""