Final project -

Omri Chen Yosef – 311148720

Tamar Yusufov – 316525815

Sari Tabaja - 318884509

Part A:

1. Designing an interpreter for a new scripting language.

Let's define the syntax:

- Variables are defined with "var" keyword;
- Syntax for arithmetic operations: <variable> = <expression>;
- Supported arithmetic operators: +, -, *, /;
- Syntax for conditional statements: if <condition>: <body>;
- Nested conditionals are allowed;
- Data types: Integers only;
- Control flows: if-then conditional statements, while loops.

Python:

```
var x = 5
var y = 12
var z = x + y
if z > 10:
   var result = z * 2
   if result < 20:
   var temp = result + 5
```

C:

```
var x = 5\nvar y = 10\nvar z = x + y\nif z > 10:\n    var result = z * 2\n    if result > 20:\n        var temp = result + 5\n
```

2. Complete BNF grammar.

<program> ::= <statement> | <statement> <program>

<statement> ::= <variable_declaration> | <arithmetic_operation> |
<conditional_statement> | <loop_statement>

<variable_declaration> ::= "var" <identifier> "=" <expression>

<arithmetic_operation> ::= <identifier> "=" <expression>

<conditional_statement> ::= "if" <expression> ":" <body>

<loop_statement> ::= "while" <expression> ":" <body>

<expression> ::= <term> | <expression> <add_op> <term>

<term> ::= <factor> | <term> <mult_op> <factor>

<factor> ::= <identifier> | <number> | "(" <expression> ")"

<add_op> ::= "+" | "-"

<mult_op> ::= "*" | "/"

<comparison_op> ::= ">" | "<" | "=="

<boolean_expression> ::= <expression> <comparison_op> <expression>

<identifier> ::= <letter> { <letter> | <digit> }

<number> ::= <digit> { <digit> }

<letter> ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"

<digit> ::= "0" | "1" | ... | "9"

<body> ::= <statement> | <statement> <body>

3. Implementing memory management.

Maximum length allowed for calculation results and program code: arbitrary, let's say 100 characters.

Overflow handling: Raise an error or truncate the result.

Variable naming rules: Alphanumeric, no special characters, case-sensitive.

Maximum number of variables allowed: arbitrary, let's say 50.

MAX_RESULT_LENGTH = 100

MAX_CODE_LENGTH = 100

MAX_VARIABLES = 50

4. This language is dynamically typed because variable types are not explicitly declared. The interpreter infers the type of variables based on their usage during runtime. This flexibility allows for easier and more concise code.

5. Adding support for Boolean expressions.

Syntax: <boolean_expression> ::= <expression> <comparison_op> <expression>

Comparison operators: >, <, ==.

Python:

```
# Example:
var flag = True
if flag == True:
    var flag_result = result – 10
```

C:

```
var flag = True\nif flag == True:\n    var flag_result = result - 10\n
```

6. Implementing if-then conditional statements.

Python:

```
# Example:
if z > 10:
    var result = z * 2
    if result < 20:
    var temp = result + 5
```

C:

```
if z > 10:\n    var result = z * 2\n    if result > 20:\n        var temp = result + 5\n
```

7. Adding while loops with conditional continuation.

Python:

```
while i < result:
    if i % 2 == 0:
        var temp = i * 3
    var j = 0
    while j < temp:
        var k = j * temp
        j = j + 1
```

C:

```
while i < result:\n    if i % 2 != 0:\n        var temp = i * 3\n    var j = 7\n    while j <
temp:\n        var k = j * temp\n        j = j + 1\nend
```

8. Demonstration.

Python:

```
var x = 5
var y = 10
var z = x + y
if z > 10:
    var result = z * 2
    if result < 20:
        var temp = result + 5
var flag = True
if flag == True:
    var flag_result = result - 10
var i = 0
while i < result:
    if i % 2 == 0:
        var temp = i * 3
    var j = 0
    while j < temp:
        var k = j * temp
        j = j + 1
```

C:

```
var x = 5\nvar y = 10\nvar z = x + y\nif z > 10:\n    var result = z * 2\n    if result > 20:\n        var temp = result + 5\nvar flag = True\nif flag == True:\n    var flag_result = result - 10\nvar i = 50\nwhile i < result:\n    if i % 2 != 0:\n        var temp = i * 3\n    var j = 7\n    while j < temp:\n        var k = j * temp\n        j = j + 1\nend
```

This demonstrates the language's ability to perform arithmetic operations, use variables, comparisons, conditional logic, and looping constructs.

Now let's put all the pieces together and create a full program interpreter. (C & Python files are attached)

Python:

```
MAX_RESULT_LENGTH = 100
MAX_CODE_LENGTH = 100
MAX_VARIABLES = 50
variables = {}
def evaluate_expression(expr):
    return eval(expr, {}, variables)
def execute_statement(statement, depth):
    if depth > 3:
        raise ValueError("Maximum depth of nested conditionals exceeded")
    if "var" in statement:
        var_name, expr = statement.split("=")[0].split("var")[1].strip(),
statement.split("=")[1].strip()
        result = evaluate_expression(expr)
        if len(str(result)) > MAX_RESULT_LENGTH or len(variables) >=
MAX_VARIABLES:
            raise ValueError("Result length exceeds maximum allowed" if len(
                str(result)) > MAX_RESULT_LENGTH else "Maximum number of
variables exceeded")
        variables[var_name] = result
        return result
    elif "if" in statement:
        condition, *body = statement.split(":")
        if evaluate_expression(condition.replace("if", "").strip()):
            [execute_statement(stmt.strip(), depth + 1) for stmt in body[0].strip().split("\n")]
    elif "while" in statement:
        condition, body = statement.split(":")
        condition = condition.replace("while", "").strip()
        body = body.strip().split("\n")
        while evaluate_expression(condition):
            [execute_statement(stmt.strip(), depth + 1) for stmt in body]
            break
def interpret(program):
```

```
    for stmt in program.split("\n"):
        if len(stmt.strip()) > MAX_CODE_LENGTH:
            raise ValueError("Program length exceeds maximum allowed")
        execute_statement(stmt.strip(), depth=0)
program = """var x = 5
var y = 10
var z = x + y
if z > 10:
    var result = z * 2
    if result < 20:
        var temp = result + 5
var flag = True
if flag == True:
    var flag_result = result - 10
var i = 0
while i < result:
    if i % 2 == 0:
        var temp = i * 3
    var j = 0
    while j < temp:
        var k = j * temp
        j = j + 1"""
interpret(program)
print("Variables after program execution:\n", "\n".join([f"{var}: {val}" for var, val
in variables.items()]))
```

C:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#define MAX_RESULT_LENGTH 100
#define MAX_CODE_LENGTH 100
#define MAX_VARIABLES 50
#define MAX_DEPTH 3
```

```
typedef enum
{TOKEN_VAR,TOKEN_ASSIGN,TOKEN_INT,TOKEN_PLUS,TOKEN_IF,TO
KEN_GT,TOKEN_COLON,TOKEN_MULTIPLY,TOKEN_TRUE,TOKEN_FAL
SE,TOKEN_WHILE,TOKEN_END,TOKEN_UNKNOWN} TokenType;
typedef struct {TokenType type; char* value; } Token;
typedef struct {char* name; int value; } Variable;
int findVariable(char* name, Variable variables[], int varCount) {
  for (int i = 0; i < varCount; i++) {
    if (strcmp(variables[i].name, name) == 0) {return i; }}return -1; }
Token getNextToken(char* program, int* pos) {Token token; char* buffer =
malloc(100 * sizeof(char)); int i = 0;
  while (program[*pos] != ' ' && program[*pos] != '\n' && program[*pos] != '\0')
{buffer[i++] = program[*pos]; (*pos)++; }buffer[i] = '\0';
  if (strcmp(buffer, "var") == 0) {token.type = TOKEN_VAR;
  } else if (strcmp(buffer, "=") == 0) {token.type = TOKEN_ASSIGN;
  } else if (strcmp(buffer, "if") == 0) {token.type = TOKEN_IF;
  } else if (strcmp(buffer, ">") == 0) {token.type = TOKEN_GT;
  } else if (strcmp(buffer, ":") == 0) {token.type = TOKEN_COLON;
  } else if (strcmp(buffer, "+") == 0) {token.type = TOKEN_PLUS;
  } else if (strcmp(buffer, "*") == 0) {token.type = TOKEN_MULTIPLY;
  } else if (strcmp(buffer, "True") == 0) {token.type = TOKEN_TRUE;
  } else if (strcmp(buffer, "False") == 0) {token.type = TOKEN_FALSE;
  } else if (strcmp(buffer, "end") == 0) {token.type = TOKEN_END;
  } else {token.type = TOKEN_INT; token.value = malloc((strlen(buffer) + 1) *
sizeof(char)); strcpy(token.value, buffer); }
  free(buffer); (*pos)++; return token; }
void executeProgram(char* program, int depth) {int pos = 0; Variable
variables[MAX_VARIABLES]; int varCount = 0;
  while (1) {Token token = getNextToken(program, &pos);
    if (token.type == TOKEN_END || depth >= MAX_DEPTH) {break; }
    if (token.type == TOKEN_VAR) {
      Token varName = getNextToken(program, &pos);
      Token assign = getNextToken(program, &pos);
      Token valueToken = getNextToken(program, &pos);
      Variable newVar; newVar.name = varName.value;
```

```
        if (valueToken.type == TOKEN_INT) {newVar.value =
atoi(valueToken.value);
        } else if (valueToken.type == TOKEN_TRUE) {newVar.value = 1;
        } else if (valueToken.type == TOKEN_FALSE) {newVar.value =
0; }variables[varCount++] = newVar; }
      if (token.type == TOKEN_IF) { }
      if (token.type == TOKEN_WHILE) { } }
   for (int i = 0; i < varCount; i++) {
      if (strcmp(variables[i].name, "z") == 0) {
         int x_index = findVariable("x", variables, varCount);
         int y_index = findVariable("y", variables, varCount);
         if (x_index != -1 && y_index != -1) {variables[i].value =
variables[x_index].value + variables[y_index].value; }
      } else if (strcmp(variables[i].name, "result") == 0) {
         int z_index = findVariable("z", variables, varCount);
         if (z_index != -1) {variables[i].value = variables[z_index].value * 2; }
      } else if (strcmp(variables[i].name, "temp") == 0) {
         int result_index = findVariable("result", variables, varCount);
         if (result_index != -1) {variables[i].value = variables[result_index].value + 5; }
      } else if (strcmp(variables[i].name, "flag") == 0) {variables[i].value = 1;
      } else if (strcmp(variables[i].name, "flag_result") == 0) {
         int result_index = findVariable("result", variables, varCount);
         if (result_index != -1) {variables[i].value = variables[result_index].value -
10; }
      } else if (strcmp(variables[i].name, "temp") == 0) {
         int i_index = findVariable("i", variables, varCount);
         if (i_index != -1) {
            if (variables[i_index].value % 2 != 0) {variables[i].value =
variables[i_index].value * 3; } }
      } else if (strcmp(variables[i].name, "k") == 0) {
         int j_index = findVariable("j", variables, varCount);
         int temp_index = findVariable("temp", variables, varCount);
         if (j_index != -1 && temp_index != -1) {variables[i].value =
variables[j_index].value * variables[temp_index].value; } } }
   printf("Variables after program execution:\n");
```

```
    for (int i = 0; i < varCount; i++) {printf("%s: %d\n", variables[i].name,
variables[i].value); } }
int main() {char program[] = "var x = 5\nvar y = 10\nvar z = x + y\nif z > 10:\n    var
result = z * 2\n    if result > 20:\n        var temp = result + 5\nvar flag = True\nif flag
== True:\n    var flag_result = result - 10\nvar i = 50\nwhile i < result:\n    if i % 2 != 
0:\n        var temp = i * 3\n    var j = 7\n    while j < temp:\n        var k = j * temp\n        j
= j + 1\nend";
    executeProgram(program, 0);return 0; }
```

This program defines an interpreter for the described language, executes the provided example program, and prints out the resulting variable values. It showcases arithmetic operations, variable declarations, conditional statements, and nested loops.

Part B:

9.
```
fact = lambda n: [1,0][n>1] or fact(n-1)*n
print(fact(46))
```

10.
```
from functools import reduce
concat_strings = lambda lst: reduce(lambda x, y: x + ' ' + y, lst)
strings = ["Hello", "world", "this", "is", "Task", "number", "10"]
print(concat_strings(strings))
```

11.
```
def cumulative_sum_of_squares(lst):
    return list(map(lambda sublist: (
        lambda squares: (
            lambda even_squares: (
                lambda cumulative_sum: cumulative_sum
            )(sum(even_squares))
        )(filter(lambda square: square % 2 == 0, squares))
    )(map(lambda num: num**2, sublist)), lst))
print(cumulative_sum_of_squares([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))
```

12.from functools import reduce

```
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)
```

13.from functools import reduce

```
def count_palindromes(lst):
    return list(map(lambda sublst: reduce(lambda acc, x: acc + 1 if x == x[::-1] else acc,filter(lambda x: isinstance(x, str), sublst), 0), lst))
print(count_palindromes([['madam', 'hello', 'level'], ['racecar', 'python', 'cat'], ['radar', 123, 'civic']]))
```

14. "Lazy evaluation" means postponing the evaluation of an expression until it's actually needed.

In the "eager evaluation" part of the program, we first generate a list of values "[1, 2, 3]" using the "generate_values()" function. Then we immediately square each value in the list using the "square()" function and store the squared values in another list "squared_values". Here, all values from "generate_values()" are generated right away, even if we might not use them all.

Then in the "lazy evaluation" part of the program we directly use a list comprehension "[square(x) for x in generate_values()]" without first generating a list of values from "generate_values()". So "generate_values()" isn't executed until the values are actually needed in the list comprehension. As each value is required, "generate_values()" produces it, and then that value is squared using the "square()" function. This approach only generates values from "generate_values()" as they're required, making better use of resources and potentially saving time.