Creación de un lenguaje de Progrmación:

Mamani Maraza Rusbel Yampier

May 22, 2024

1 Lenguaje generado en Python:

El siguiente código en Python es el código base de mi lenguaje, siendo este un archivo.py pero luego lo haremos un archivo.exe Al abrir el símbolo del sistema, primeramente tenemos que instalar pip install pyinstaller luego creamos el ejecutable: pyinstaller –onefile "Nombre del archivo".py, Esto creara un archivo ejecutable del lenguaje.

Abrimos el símbolo del sistema o CMD, luego colocamos el nombre del ejecutable, seguidamente por el nombre de un archivo TXT previamente realizado, ya que este será como un interpretador de código, en este archivo estará el código que queremos realizar. Crea un archivo de texto y escribe tu programa en el lenguaje que has creado. Por ejemplo: x = whole10 y = whole5 see x + y

Guardamos este archivo con una extensión que identifique el tipo de archivo, por ejemplo, prueba.txt.

Ejecutar el programa

Abre el símbolo del sistema o CMD y navega al directorio donde se encuentra el programa .exe de tu lenguaje y donde has guardado tu programa.

Luego, ejecuta el programa proporcionando el nombre del archivo de tu programa como argumento:

dist/main.exe mi$_p$*rograma.txt*

Interpretar los resultados

El programa ejecutable leerá el archivo del programa, lo analizará, ejecutará y mostrará los resultados en la terminal o línea de comandos:

```
import re
import sys
import os

def lexer(code):
    tokens = []
    token_specification = [
        ('COMMENT',   r'\*\|.*?\|\*'),  # Comentarios
        ('NUMBER',    r'whole\d+'),       # Enteros
        ('FLOAT',     r'imp\d+(\.\d+)?'),# No exactos
        ('STRING',    r'\[.*?\]'),        # Cadenas
```

```
        ('IDENTIFIER', r'[a-zA-Z][a-zA-Z0-9]*'), # Identificadores
        ('ASSIGN',     r'='),                    # Asignaci n
        ('PRINT',      r'see'),                  # Imprimir
        ('PLUS',       r'\+'),                   # Suma
        ('MINUS',      r'-'),                    # Resta
        ('TIMES',      r'\*'),                   # Multiplicaci n
        ('DIVIDE',     r'/'),                    # Divisi n
        ('SKIP',       r'[ \t]+'),               # Espacios y tabulaciones
        ('NEWLINE',    r'\n'),                   # Nueva l nea
        ('MISMATCH',   r'.'),                    # Caracteres no reconocidos
    ]
    token_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    for mo in re.finditer(token_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        if kind == 'NUMBER':
            value = int(value[5:])  # Remove 'whole'
        elif kind == 'FLOAT':
            value = float(value[3:])  # Remove 'imp'
        elif kind == 'STRING':
            value = value[1:-1]  # Remove brackets
        elif kind == 'IDENTIFIER':
            value = str(value)
        elif kind == 'SKIP':
            continue
        elif kind == 'NEWLINE':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'Error: {value}')
        tokens.append((kind, value))
    print("Tokens:", tokens)  # A adir impresi n para depuraci n
    return tokens

class Node:
    def __init__(self, type, value=None, children=None):
        self.type = type
        self.value = value
        self.children = children if children else []

def parse(tokens):
    def parse_program(index):
        nodes = []
        while index < len(tokens):
            print(f"parse_program: index {index}, token {tokens[index]}")
# A adir impresi n para depuraci n
            if tokens[index][0] == 'COMMENT':
```

```
                index += 1
                continue
            node, index = parse_instruction(index)
            nodes.append(node)
        return nodes, index

    def parse_instruction(index):
        print(f"parse_instruction: index {index}, token {tokens[index]}")
# A adir impresi n para depuraci n
        if tokens[index][0] == 'IDENTIFIER' and tokens[index + 1][0] == 'ASSIGN'
            return parse_assignment(index)
        elif tokens[index][0] == 'PRINT':
            return parse_print(index)
        else:
            raise SyntaxError(f'Unexpected token: {tokens[index]}')

    def parse_assignment(index):
        identifier = tokens[index][1]
        index += 2  # Skip identifier and '='
        expr, index = parse_expression(index)
        return Node('Assignment', value=identifier, children=[expr]), index

    def parse_print(index):
        index += 1  # Skip 'see'
        expr, index = parse_expression(index)
        return Node('Print', children=[expr]), index

    def parse_expression(index):
        left, index = parse_operand(index)
        while index < len(tokens) and tokens[index][0] in ('PLUS', 'MINUS', 'TIM
            op = tokens[index][0]
            index += 1
            right, index = parse_operand(index)
            left = Node('BinaryOp', value=op, children=[left, right])
        return left, index

    def parse_operand(index):
        token_type, value = tokens[index]
        if token_type == 'NUMBER':
            node = Node('Number', value=value)
        elif token_type == 'FLOAT':
            node = Node('Float', value=value)
        elif token_type == 'STRING':
            node = Node('String', value=value)
        elif token_type == 'IDENTIFIER':
            node = Node('Identifier', value=value)
```

```
            else:
                raise SyntaxError(f'Unexpected token in operand: {tokens[index]}')
            return node, index + 1

        ast, _ = parse_program(0)
        return ast

    def evaluate(ast, env=None):
        if env is None:
            env = {}
        results = []
        for node in ast:
            if node.type == 'Assignment':
                identifier = node.value
                value = evaluate_expression(node.children[0], env)
                env[identifier] = value
            elif node.type == 'Print':
                value = evaluate_expression(node.children[0], env)
                results.append(value)
        return results

    def evaluate_expression(node, env):
        if node.type == 'Number':
            return node.value
        elif node.type == 'Float':
            return node.value
        elif node.type == 'String':
            return node.value
        elif node.type == 'Identifier':
            return env[node.value]
        elif node.type == 'BinaryOp':
            left = evaluate_expression(node.children[0], env)
            right = evaluate_expression(node.children[1], env)
            if node.value == 'PLUS':
                return left + right
            elif node.value == 'MINUS':
                return left - right
            elif node.value == 'TIMES':
                return left * right
            elif node.value == 'DIVIDE':
                return left / right
        else:
            raise ValueError('Unknown node type')

    def repl():
        env = {}
```

```
    while True:
        try:
            code = input('> ')
            if code.strip() == 'exit':
                break
            tokens = lexer(code + '\n')
            ast = parse(tokens)
            results = evaluate(ast, env)
            for result in results:
                print(result)
        except Exception as e:
            print(e)

def main():
    if len(sys.argv) == 2:
        source_file = sys.argv[1]
        if not os.path.isfile(source_file):
            print(f"Error: No se puede encontrar el archivo {source_file}")
            return
        try:
            with open(source_file, 'r') as file:
                code = file.read()
            tokens = lexer(code)
            ast = parse(tokens)
            results = evaluate(ast)
            for result in results:
                print(result)
        except PermissionError:
            print(f"Error: No se tienen permisos para acceder al archivo {source
        except Exception as e:
            print(f"Error al procesar el archivo {source_file}: {e}")
    else:
        repl()

if __name__ == "__main__":
    main()
```