Universidad de Guadalajara
Centro Universitario de Ciencias Exactas E Ingenierías
División de Electrónica y Computación
Ingeniería en computación

D02 SEMINARIO DE SOLUCION DE PROBLEMAS DE TRADUCTORES DE LENGUAJES II

Analizador Léxico Completo

216788333 Alejandro de Jesús Romo Rosales

15/09/2020

Objetivo

Genera un analizador léxico utilizando todos los símbolos léxicos en el archivo simbolos_lexicos.pdf

Introducción

Un analizador léxico o analizador lexicográfico (en inglés scanner) es la primera fase de un compilador, consistente en un programa que recibe como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de tokens (componentes léxicos) o símbolos. Estos tokens sirven para una posterior etapa del proceso de traducción, siendo la entrada para el analizador sintáctico (en inglés parser).

Esta etapa está basada usualmente en una máquina de estados finitos. Esta máquina contiene la información de las posibles secuencias de caracteres que puede conformar cualquier token que sea parte del lenguaje (las instancias individuales de estas secuencias de caracteres son denominados lexemas). Por ejemplo, un token de naturaleza entero puede contener cualquier secuencia de caracteres numéricos.

Metodología

Posterior a una investigación sobre el tema en cuestión, implementar seleccionar un lenguaje que soporte el paradigma orientado a objetos para así poder modelar de una manera más efectiva y eficiente la abstracción de las clases, por ejemplo, la clase Token. Una vez realizado esto se procederá a programar el analizador léxico mediante el uso de funciones y finalizando esto predefinir una serie de pruebas las cuales nos ayudaran a corroborar el su debido comportamiento.

Materiales

1. Python 3.7
2. VS Code
3. Windows 10

Desarrollo

Para programar nuestro analizador léxico el primer paso fue definir nuestra tabla de símbolos.

```
"""
                        **********************    TOKENS    ******************
****
"""

DIGITS = '0123456789'
LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWYXZabcdefghijklmnopqrstuvwxyz'
ASING = '='
```

```
IDENTIFIER = 'ID'
DIRECTIVE = '#'
LIBRARY = 'LIBRARY'
KEYWORD = ['return', 'int', 'float', 'string', 'char', 'stack', 'list', 'que
ue', 'tuple',
           'sort', 'cout', 'printf', 'scanf', 'include', 'void', 'public', '
private', 'class',
           'null', 'scanf_s', 'array', 'vector', 'define', 'extern', 'for',
'while', 'if', 'else', 'switch',
           'case', 'default', 'do', 'continue', 'getchar', 'cout', 'cin', 'g
ets']

TT_KEYWORD = 'KEYWORD'
TT_INT = 'INT'
TT_FLOAT = 'FLOAT'
TT_PLUS = 'PLUS'
TT_MINUS = 'MINUS'
TT_MUL = 'MULTIPLICATION'
TT_DIV = 'DIVISION'
TT_EXP = 'POWER'
TT_LPAREN = 'LEFT PARENTESIS'
TT_RPAREN = 'RIGHT PARENTESIS'
TT_COMA = 'COMA'
TT_SEMI = 'SEMICOLON'
TT_LCBRACK = 'LEFT CURVY BRACKET'
TT_RCBRACK = 'RIGHT CURVY BRACKET'
TT_AMPERSON = 'AMPERSON'
TT_BAR = 'VERTICAL BAR'
TT_DQUOTATION = 'DOUBLE QUOTATION MARKS'
TT_SQUOTATION = 'SIMPLE QUOTATION MARKS'
TT_LESSTHAN = 'LESS THAN'
TT_GREATERTHAN = 'GREATER THAN'
TT_MODULE = 'MODULE OPERATOR'
TT_DOLLAR = 'DOLLAR SIGN'
TT_RELATIONAL = 'RELATIONAL OPERATOR'
TT_NEGATION = 'NEGATION OPERATOR'
TT_EQUALITY = 'EQUALITY OPERATOR'
TT_OR = 'OR OPERATOR'
TT_AND = 'AND OPERATOR'
TT_DOT = 'DOT'
TT_EOF = 'EOF'
```

Definida nuestra tabla de símbolos, procederemos a definir los posibles errores que pueden surgir debido a una mala entrada

```
"""
                        **********************   ERRORS   ******************
****
"""
# La clase base representativa de un Error


class Error:
    def __init__(self, pos_start, pos_end, error_name, details):
        self.pos_start = pos_start
        self.pos_end = pos_end
        self.error_name = error_name
        self.details = details

    def as_string(self):
        result = f'{self.error_name}: {self.details}\n'
        # result += f'File {self.pos_start.fn}, line {self.pos_start.ln + 1}
'
        result += '\n\n' + \
            string_with_arrows(self.pos_start.ftxt,
                               self.pos_start, self.pos_end)
        return result

# Error de caracteres desconocidos


class IllegalCharError(Error):

    def __init__(self, pos_start, pos_end, details=''):
        super().__init__(pos_start, pos_end, 'Invalid Character ', details)


class InvalidSyntaxError(Error):
    def __init__(self, pos_start, pos_end, details=''):
        super().__init__(pos_start, pos_end, 'Invalid Syntax', details)

# Error en la definicion de la variable


class IllegalVariableError(Error):

    def __init__(self, pos_start, pos_end, details=''):
        super().__init__(pos_start, pos_end, 'Invalid Variable ', details)

# Señala donde está el error
```

```
def string_with_arrows(text, pos_start, pos_end):
    result = ''

    # Calculate indices
    idx_start = max(text.rfind('\n', 0, pos_start.idx), 0)
    idx_end = text.find('\n', idx_start + 1)
    if idx_end < 0:
        idx_end = len(text)

    # Generate each line
    line_count = pos_end.ln - pos_start.ln + 1
    for i in range(line_count):
        # Calculate line columns
        line = text[idx_start:idx_end]
        col_start = pos_start.col if i == 0 else 0
        col_end = pos_end.col if i == line_count - 1 else len(line) - 1

        # Append to result
        result += line + '\n'
        result += ' ' * col_start + '^' * (col_end - col_start)

        # Re-calculate indices
        idx_start = idx_end
        idx_end = text.find('\n', idx_start + 1)
        if idx_end < 0:
            idx_end = len(text)

    return result.replace('\t', '')
```

Con esto programado procedemos a programar los más importante, nuestro analizador léxico, el cuál estará compuesto por 3 partes:

1. Clase Position para identificar la ubicación exacta de un error
2. Clase Token para definir nuestros tokens
3. Clase Lexer para representar nuestro analizador léxico

Clase Position

```
"""
                        **********************    POSITION    ***************
******
"""


class Position:
    def __init__(self, idx, ln, col, fn, ftxt):
```

```
        self.idx = idx
        self.ln = ln
        self.col = col
        self.fn = fn
        self.ftxt = ftxt

    def advance(self, current_char=None):
        self.idx += 1
        self.col += 1

        if current_char == '\n':
            self.ln += 1
            self.col = 0
        return self

    def back(self, current_char=None):
        self.idx += -1
        self.col += -1

        if current_char == '\n':
            self.ln -= 1
            self.col = 0
        return self

    def copy(self):
        return Position(self.idx, self.ln, self.col, self.fn, self.ftxt)
```

Clase Token

```
"""
                        **********************   TOKEN   ******************
***
"""

class Token:

    # El constructor recibe el tipo de token del que se trata: int, float, o
perador
    # Y de ser un numero o nombre de variable se asigna, de lo contrario es
nulo

    def __init__(self, type, value=None, pos_start=None, pos_end=None):
        self.type = type
        self.value = value
```

```
        if pos_start:
            self.pos_start = pos_start.copy()
            self.pos_end = pos_start.copy()
            self.pos_end.advance()

        if pos_end:
            self.pos_end = pos_end

    # Se sobreescribe la manera en la que se imprime el objeto

    def __repr__(self):
        if self.value:
            return f'{self.type} -> {self.value}'
        return f'{self.type}'

    # Regresa el valor del Token

    def Value(self):
        if self.value != None:
            return self.value

    # Regresa el tipo del Token

    def Type(self):
        return self.type
```

Clase Lexer

```
"""
                        **********************    LEXER    ******************
***
"""

class Lexer:
    def __init__(self, fn, text):
        self.fn = fn
        self.text = text
        self.pos = Position(-1, 0, -1, fn, text)
        self.current_char = None
        self.advance()

    def advance(self):
        self.pos.advance(self.current_char)
        self.current_char = self.text[self.pos.idx] if self.pos.idx < len(
            self.text) else None
```

```
def analyze(self):
    tokens = []

    while self.current_char != None:
        if self.current_char == '\t':
            self.advance()
        elif self.current_char in DIGITS:
            tokens.append(self.make_number())
        elif self.current_char in LETTERS:
            tokens.append(self.make_word())
        elif self.current_char == '+':
            tokens.append(Token(TT_PLUS, pos_start=self.pos))
            self.advance()
        elif self.current_char == '-':
            tokens.append(Token(TT_MINUS, pos_start=self.pos))
            self.advance()
        elif self.current_char == '*':
            tokens.append(Token(TT_MUL, pos_start=self.pos))
            self.advance()
        elif self.current_char == '/':
            tokens.append(Token(TT_DIV, pos_start=self.pos))
            self.advance()
        elif self.current_char == '(':
            tokens.append(Token(TT_LPAREN, pos_start=self.pos))
            self.advance()
        elif self.current_char == ')':
            tokens.append(Token(TT_RPAREN, pos_start=self.pos))
            self.advance()
        elif self.current_char == '=':
            tokens.append(self.make_equality())
            self.advance()
        elif self.current_char == ';':
            tokens.append(Token(TT_SEMI, pos_start=self.pos))
            self.advance()
        elif self.current_char == ',':
            tokens.append(Token(TT_COMA, pos_start=self.pos))
            self.advance()
        elif self.current_char == '{':
            tokens.append(Token(TT_LCBRACK, pos_start=self.pos))
            self.advance()
        elif self.current_char == '}':
            tokens.append(Token(TT_RCBRACK, pos_start=self.pos))
            self.advance()
        elif self.current_char == '&':
```

```
                tokens.append(self.make_logical_operator())
                self.advance()
            elif self.current_char == '|':
                tokens.append(self.make_logical_operator())
                self.advance()
            elif self.current_char == '#':
                tokens.append(Token(DIRECTIVE, pos_start=self.pos))
                self.advance()
            elif self.current_char == '"':
                tokens.append(Token(TT_DQUOTATION, pos_start=self.pos))
                self.advance()
            elif self.current_char == "'":
                tokens.append(Token(TT_SQUOTATION, pos_start=self.pos))
                self.advance()
            elif self.current_char == '<':
                tokens.append(self.make_relational_operator())
                self.advance()
            elif self.current_char == '>':
                tokens.append(self.make_relational_operator())
                self.advance()
            elif self.current_char == '%':
                tokens.append(Token(TT_MODULE, pos_start=self.pos))
                self.advance()
            elif self.current_char == '.':
                tokens.append(Token(TT_DOT, pos_start=self.pos))
                self.advance()
            elif self.current_char == '!':
                tokens.append(self.make_equality())
                self.advance()
            elif self.current_char == '$':
                tokens.append(Token(TT_DOLLAR, pos_start=self.pos))
                self.advance()
            elif self.current_char == '\n':
                self.advance()
            elif self.current_char == ' ':
                self.advance()
            else:
                pos_start = self.pos.copy()
                char = self.current_char
                self.advance()
                return [], IllegalCharError(pos_start, self.pos, "'" + char
+ "'")

        tokens.append(Token(TT_EOF, pos_start=self.pos))
        return tokens, None
```

```
    def make_number(self):
        num_str = ''
        dot_count = 0
        pos_start = self.pos.copy()

        while self.current_char != None and self.current_char in DIGITS + '.
':
            if self.current_char == '.':
                if dot_count == 1:
                    break
                dot_count += 1
                num_str += '.'
            else:
                num_str += self.current_char
            self.advance()

        if dot_count == 0:
            return Token(TT_INT, int(num_str), pos_start, self.pos)
        else:
            return Token(TT_FLOAT, float(num_str), pos_start, self.pos)

    def make_word(self):
        word = ''
        dot_count = 0
        pos_start = self.pos.copy()

        while self.current_char != ' ' and self.current_char != None and (se
lf.current_char in LETTERS or dot_count <= 1):
            if dot_count == 0:
                if self.current_char in LETTERS:
                    word += self.current_char
                elif self.current_char == '.':
                    word += self.current_char
                    dot_count += 1
            else:
                if self.current_char == 'h':
                    word += self.current_char
            self.advance()
        if word in KEYWORD:
            return Token(TT_KEYWORD, word, pos_start, self.pos)
        elif word.endswith('.h'):
            return Token(LIBRARY, word, pos_start, self.pos)
        else:
            return Token(IDENTIFIER, word, pos_start, self.pos)
```

```
def make_equality(self):
    pos_start = self.pos.copy()

    if self.current_char == '=':
        self.advance()
        if self.current_char != None and self.current_char == '=':
            return Token(TT_EQUALITY, '==', pos_start, self.pos)
        return Token(ASING, '=', pos_start, self.pos)
    else:
        self.advance()
        if self.current_char != None and self.current_char == '=':
            return Token(TT_EQUALITY, '!=', pos_start, self.pos)
        return Token(TT_NEGATION, '!', pos_start, self.pos)

def make_logical_operator(self):
    pos_start = self.pos.copy()

    if self.current_char == '&':
        self.advance()
        if self.current_char != None and self.current_char == '&':
            return Token(TT_AND, '&&', pos_start, self.pos)
        return Token(TT_AMPERSON, '&', pos_start, self.pos)
    else:
        self.advance()
        if self.current_char != None and self.current_char == '|':
            return Token(TT_OR, '||', pos_start, self.pos)
        return Token(TT_BAR, '|', pos_start, self.pos)

def make_relational_operator(self):
    pos_start = self.pos.copy()

    if self.current_char == '<':
        self.advance()
        if self.current_char != None and self.current_char == '=':
            return Token(TT_RELATIONAL, '<=', pos_start, self.pos)
        return Token(TT_RELATIONAL, '<', pos_start, self.pos)
    else:
        self.advance()
        if self.current_char != None and self.current_char == '=':
            return Token(TT_RELATIONAL, '>=', pos_start, self.pos)
        return Token(TT_RELATIONAL, '>', pos_start, self.pos)
```

Una vez programado esto procederemos a realizar diversas pruebas.

Resultados

Se ejecutaron diversas pruebas para corroborar el correcto funcionamiento del programa.

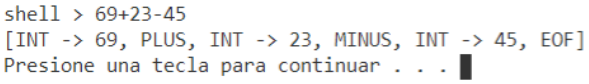


Fig. 1. Se comprueban los identificadores y operaciones de suma y resta

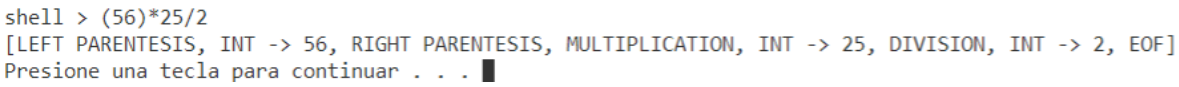


Fig. 2. Se comprueban los paréntesis y operaciones de división y multiplicación

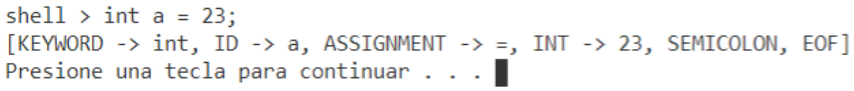


Fig. 3. Se comprueba la palabra reservada “int”, un identificador, asignación y punto y coma

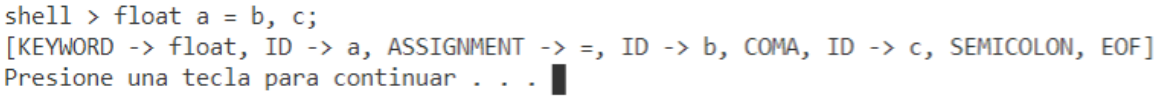
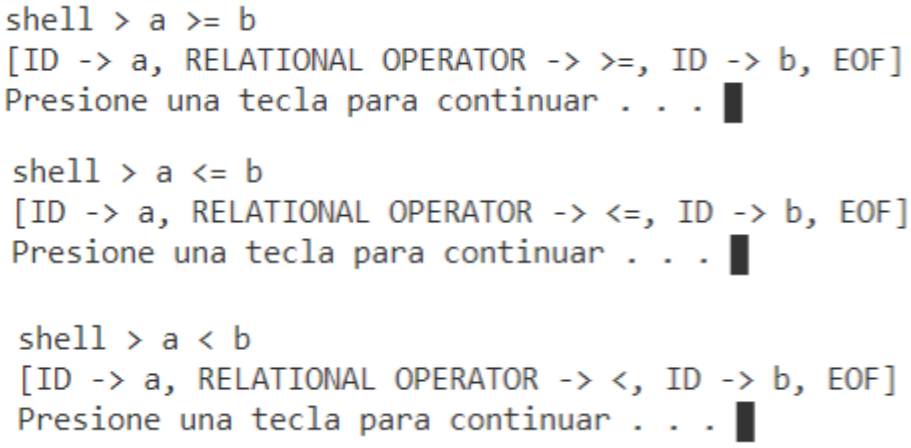


Fig. 4. Se comprueba la palabra reservada “float”, múltiples identificadores, asignación, coma y punto y coma



```
shell > a > b
[ID -> a, RELATIONAL OPERATOR -> >, ID -> b, EOF]
Presione una tecla para continuar . . .
```

Fig. 5. Se comprueban identificadores y operadores relacionales

```
shell > if (a == b) && (c == b) || (a != c){ return 0; }
[KEYWORD -> if, LEFT PARENTESIS, ID -> a, EQUALITY OPERATOR -> ==, ID -> b, AND OPERATOR -> &&, LEFT PARENTESIS, ID -> c, EQUALITY OPERATOR -> ==, ID -> b, OR OPERATO
R -> ||, LEFT PARENTESIS, ID -> a, EQUALITY OPERATOR -> !=, ID -> c, KEYWORD -> return, INT, SEMICOLON, RIGHT CURVY BRACKET, EOF]
Presione una tecla para continuar . . .
```

Fig. 6. Se comprueba la sentencia if, paréntesis, operación AND, operación OR, desigualdad, igualdad, y palabra reservada return

```
shell > while(true){ if(!false){return false;} else{return true};
[ID -> whiletrue, ID -> iffalsereturn, ID -> false, SEMICOLON, RIGHT CURVY BRACKET, ID -> elsereturn, ID -> true, SEMICOLON, EOF]
Presione una tecla para continuar . . .
```

Fig. 7. Se comprueba el ciclo while, palabra reservada true y false, sentencias if y else y palabra reservada return.