Universidad Rafael Landívar
Facultad de Ingeniería
Compiladores Sec. 02
Ing. Moises Antonio Alonso Gonzales

Análisis y Diseño del Lenguaje de Programación

Joaquín Raymundo Choc Salvador - 1280423
Karla Alejandra Palacios Escobar - 1173219
José Pablo Sosa España - 1057623
Josué Daniel Tzul Gochez – 1067123

Guatemala 3 de marzo del 2025

INTRODUCCIÓN

El presente documento contiene el análisis detallado de la gramática y el analizador léxico en Python para la fase 1 del proyecto de Compiladores con el propósito de la creación de un lenguaje de programación propio inspirado en los lenguajes de alto nivel, hecho para ser intuitivo y eficiente.

Para la primera fase se estableció la gramática libre de contexto que cumple con las restricciones de una gramática LL1, es decir, es determinista, no ambigua, y no contiene recursión por la izquierda, permitiendo una estructuración correcta para la declaración de variables. expresiones aritméticas, estructuras de control y la entrada/salida de datos.

Continuando, el analizador léxico fue construido en Python por medio de expresiones regulares, permitiendo generar los tokens de los programas escritos en el lenguaje desarrollado mediante archivos de prueba que contienen los fragmentos de código, incluyendo la impresión de cadenas, operaciones aritméticas, el uso de “if”, definición de una función y su respectiva llamada y la salida de datos con un ciclo while.

Dentro del documento se incluyen las decisiones tomadas durante el diseño del lenguaje y una evaluación de sus características distintivas respecto a los lenguajes de programación actuales.

GRAMÁTICA Y REGLAS LÉXICAS

En esta sección se detallan las especificaciones de la gramática y sus reglas léxicas. El lenguaje se define de manera que resulte familiar para los programadores de lenguajes de alto nivel, adoptando elementos sintácticos presentes en lenguajes como C# pero simplificados para facilitar la escritura de código.

Gramática:

```
// Inicio del programa
<S> ::= <P> $

// P va a ser cada linea de co´digo, ya que produce todo
<P> ::= <Declaracion> <P>
       | <Funcion> <P>
       | <Control> <P>
       | <While> <P>
       | <IN> <P>
       | <OUT> <P>
       | <LlamadaFuncion> <P>
       | <Return> <P>
       | ε

// 1. Declaraciones (D)
<Declaracion> ::= <B> id = <V> ;
<B> ::= i | f | s | b
<V> ::= NUM | FLOAT | STRING | BOOL

// 2. Expresiones Aritme´ticas (E)
<E> ::= <T1> <E'>
<E'> ::= + <T1> <E'>
       | - <T1> <E'>
       | ε
<T1> ::= <F1> <T'>
<T'> ::= * <F1> <T'>
       | / <F1> <T'>
       | ε
<F1> ::= id
       | STRING
       | NUM
       | FLOAT
       | ( <E> )
       | <LlamadaFuncion>
```

```
// 3. Desigualdades (R)
<R> ::= <E> DESI <E>

// 4. Estructuras de Control
<Control> ::= if ( <R> ) { <P> } <Control'>
<Control'> ::= else { <P> }
          | ε
<While> ::= while ( <R> ) { <P> }

// 5. Funciones
<Funcion> ::= def id ( <Params> ) { <P> }
<Params> ::= <B> id <ParamList>
           | ε
<ParamList> ::= , <B> id <ParamList>
              | ε
<LlamadaFuncion> ::= id ( <Args> ) ;
<Args> ::= <E> <ArgList>
         | ε
<ArgList> ::= , <E> <ArgList>
            | ε
<Return> ::= return <E> ;

// 6. Input/Output
<IN> ::= read ( id ) ;
<OUT> ::= write ( <E> ) ;
```

- **Declaraciones de variables** con tipos predefinidos y asignaciones opcionales.

- **Expresiones aritméticas y lógicas** con operadores bien definidos y precedencia adecuada.

- **Estructuras de control** como condicionales y bucles con sintaxis clara.

- **Declaración de funciones** permitiendo la escritura de sus argumentos y la llamada a la función.

- **Manejo de entrada y salida**, facilitando la interacción con el usuario.

Reglas Léxicas:

```
ID      ::= LETRA (LETRA | DIGITO)*
STRING  ::= '"' (LETRA | DIGITO | ESPACIO | "_" | "-" )* '"'
NUM     ::= DIGITO+
FLOAT   ::= NUM "." NUM
DIGITO  ::= [0-9]
BOOL    ::= "True" | "False"
DESI    ::= "<" | ">" | "==" | "=" | "!="
READ    ::= "read"
WRITE   ::= "write"
```

Definición de Tokens como **ID, NUM, FLOAT, STRING**, operadores, palabras reservadas, y la estrategia de reconocimiento basado en expresiones.

En caso de las palabras reservadas (**if, else, def, return**) tienen producciones independientes dentro de la gramática, permitiendo tener sus propia estructura dentro de esta

Esta combinación de reglas sintácticas y léxicas permite que el lenguaje sea estructurado, predecible y fácil de analizar en etapas posteriores del compilador.

CÓDIGO FUENTE DEL ANALIZADOR LÉXICO

El código fuente implementado en Python ha sido diseñado para tokenizar las cadenas de entrada de los archivos de prueba según las reglas léxicas definidas con expresiones regulares.

Código fuente documentado:

```
import re

def load_file(file_path):
    """
     Lee un archivo de texto y devuelve su contenido como una única cadena.
     Se eliminan los espacios en blanco al inicio y final de cada línea antes de concatenarlas.

    :param file_path: Ruta del archivo a leer.
     :return: Contenido del archivo como una sola cadena de texto.
    """
    with open(file_path, 'r', encoding='utf-8') as f:
                return " ".join(line.strip() for line in f.readlines())

# Definición de tokens con prioridad en palabras clave
TOKEN_REGEX = {
     'READ': re.compile(r'\bread\b'),         # Palabra clave 'read'
     'WRITE': re.compile(r'\bwrite\b'),       # Palabra clave 'write'
    'IF': re.compile(r'\bif\b'),           # Palabra clave 'if'
     'ELSE': re.compile(r'\belse\b'),         # Palabra clave 'else'
     'WHILE': re.compile(r'\bwhile\b'),       # Palabra clave 'while'
     'DEF': re.compile(r'\bdef\b'),           # Palabra clave 'def' (definición de función)
      'RETURN': re.compile(r'\breturn\b'),    # Palabra clave 'return'
       'BOOL': re.compile(r'\bTrue\b|\bFalse\b'),    # Valores booleanos 'True' o 'False'
     'DESI': re.compile(r'==|!=|<=|>=|<|>'),   # Operadores de comparación
```

```
    'FLOAT': re.compile(r'\d+\.\d+'),      # Números de punto
flotante
    'NUM': re.compile(r'\b\d+\b'),        # Números enteros
    'STRING': re.compile(r'"[^"]*"'),      # Cadenas de texto
entre comillas dobles
    'TYPE': re.compile(r'\bi\b|\bf\b|\bs\b|\bb\b'),   # Tipos
de datos (i: entero, f: float, s: string, b: booleano)
        'ID':  re.compile(r'\b[a-zA-Z_][a-zA-Z0-9_]*\b'),    #
Identificadores (variables, nombres de funciones)
    'EQUAL': re.compile(r'='),                  # Signo de
asignación '='
    'OP': re.compile(r'[+\-*/]'),              # Operadores
aritméticos '+', '-', '*', '/'
    'LPAREN': re.compile(r'\('),               # Paréntesis
izquierdo '('
    'RPAREN': re.compile(r'\)'),               # Paréntesis
derecho ')'
    'LBRACE': re.compile(r'\{'),            # Llave izquierda
'{'
    'RBRACE': re.compile(r'\}'),           # Llave derecha '}'
    'COMMA': re.compile(r','),              # Coma ','
    'SEMICOLON': re.compile(r';')           # Punto y coma ';'
}

def lexer(input_text):
    """
    Analizador léxico (lexer) que toma un código fuente como
entrada y lo convierte en una lista de tokens.

    :param input_text: Código fuente a analizar.
    :return: Lista de tokens representados como tuplas (tipo
de token, valor).
    """
    tokens = []
    while input_text:
        input_text = input_text.lstrip()  # Eliminar espacios
en blanco iniciales
        matched = False

        # Intentar hacer coincidir la entrada con los patrones
de los tokens
        for token_type, pattern in TOKEN_REGEX.items():
            match = pattern.match(input_text)
            if match:
```

```
                tokens.append((token_type, match.group(0)))
                    input_text = input_text[match.end():]   #
Avanzar en la cadena de entrada
                matched = True
                break

        # Si no hay coincidencia, lanzar un error léxico
        if not matched:
                    raise  SyntaxError(f"Error  léxico  en:
{input_text}")

      tokens.append(('EOF', ''))   # Agregar token de fin de
archivo
    return tokens

def main():
    """
     Función principal que solicita al usuario la ruta de un
archivo,
    lee su contenido y genera los tokens utilizando el lexer.
    """
      file_path = input("Ingrese la ruta del archivo con el
código de prueba: ").strip()
    try:
        code = load_file(file_path)
        tokens = lexer(code)
        print("Tokens generados:")
        for token in tokens:
            print(token)
    except Exception as e:
        print(f"Error: {e}")

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

- **Función** `load_file(filename)` abre el archivo en modo lectura, lee el documento, cierra el archivo y devuelve el contenido como una cadena de texto .

- **Diccionario** `TOKEN_REGEX` define los patrones de las expresiones regulares para identificar los tokens de la cadena devuelta en load_file.

- **Función** `lexer(text)` hace las coincidencias de TOKEN_REGEX con las expresiones regulares definidas, si encuentra un token válido lo almacena en una lista con su tipo y valor. (Ignora espacios en blanco y comentarios).

- `main()` solicita el nombre del archivo de entrada carga el contenido usando load_file(), llama a lexer() para obtener los tokens y muestra los tokens generados.

ARCHIVOS DE PRUEBA

Se incluyen a continuación los archivos de prueba utilizados para validar el correcto funcionamiento del analizador léxico y la gramática del lenguaje.

Prueba 1: Programa que imprime “Hola Mundo”.

```
write("Hola Mundo")
```

Prueba 2: Programa que realiza una operación aritmética básica.

```
i ejemplo1 = 23;
(ejemplo1 + 3) * 5;
```

Prueba 3: Programa que demuestre el uso de el input de datos y de acuerdo a la entrada del usuario decidir el flujo del programa a través de un if.

```
i ejemplo3 = 7;
i valorInput;
if (ejemplo3 < 10) {
    read(valorInput);
    write(valorInput);
}
```

Prueba 4: Definición de una función que reciba parámetros y devuelva un resultado, la función debe ser llamada desde el programa principal.

```
def suma(i a, i c) {
    return a + c;
}
write(suma(5, 10));
```

Prueba 5: Programa que contenga definición de variables, entrada y salida de datos, y que utilice el bucle “while”.

```
i contador = 0;
while (contador < 5) {
    write(contador);
    contador = contador + 1;
}
```

Aunque la gramática definida se centra en la tokenización (análisis léxico), la sintaxis de asignación y actualización de variables se asume válida para el análisis léxico, ya que el analizador reconoce correctamente los tokens involucrados.

ANÁLISIS DE LA GRAMÁTICA Y DECISIONES DE DISEÑO

La gramática propuesta cumple con las características de una LL1, determinista porque cada producción es unívoca mediante el análisis del primer símbolo de la cadena, no ambigua porque la estructura garantiza que la cadena de entrada se pueda derivar de manera única y sin recursión por la izquierda ya que se evitan las producciones recursivas del lado izquierdo facilitando la implementación en las fases futuras del proyecto.

Además, se realizaron comentarios en la gramática para identificar las secciones que manejan declaraciones, expresiones aritméticas, desigualdades, estructuras de control y funciones. Esto favorece la legibilidad y la modularidad del lenguaje.

Para las decisiones de Diseño y Consideraciones del Lenguaje se tuvieron en cuenta las siguientes características:

Sintaxis familiar: Se buscó que el lenguaje sea similar a lenguajes de alto nivel como C#, utilizando palabras reservadas y estructuras (uso de paréntesis, llaves y punto y coma) que resulten familiares para el programador.

Simplificación en la Declaración de Variables: Para evitar la verbosidad en la declaración de tipos, se decidió utilizar únicamente la letra inicial para representar cada tipo de dato:

i para integer (entero)
f para float (flotante)
s para string (cadena de texto)
b para boolean (booleano)

Legibilidad y Funcionalidad: Se mantuvo la indentación y el uso de delimitadores (llaves y paréntesis) de manera similar a lenguajes convencionales, asegurando que el código sea claro y fácil de entender sin perder simplicidad.

Priorización en el Análisis Léxico: Se implementaron expresiones regulares con prioridades específicas para reconocer primero las palabras clave y tipos, lo que ayuda a evitar conflictos durante el proceso de tokenización.

Extensibilidad: La estructura de la gramática y el analizador léxico permiten la incorporación de nuevas funcionalidades (por ejemplo, la definición de funciones o estructuras de control adicionales) sin alterar de forma significativa el diseño inicial.

Diferenciador del Lenguaje: Entre los aspectos positivos se destaca la simplicidad combinada con la potencia funcional. La reducción de la complejidad sintáctica (como la abreviación de tipos) y el uso de estructuras familiares facilitan el aprendizaje y la escritura de código, haciendo que este lenguaje se distinga de otros que requieren mayor verbosidad.

CONCLUSIONES

- El diseño de la gramática es claro y determinista (LL1) facilitando el desarrollo de compiladores.
- El lenguaje cuenta con un diseño simple pero eficaz, amigable con el usuario y es escalable.
- El funcionamiento del analizador léxico se demuestra en el correcto funcionamiento de la tokenización de los archivos de prueba.
- Sintaxis simplificada para la reducción de la curva de aprendizaje sin sacrificar la funcionalidad necesaria para la programación de tareas comunes.