TECNOLÓGICO
DE MONTERREY®

Project I: Scanner
Compiladores
Paula Sofía Soto Ayala
A01620155

Índice

1. Introducción

1.1. Resumen

El desarrollo de un analizador léxico es un componente esencial en la construcción de un compilador, y este reporte detalla la implementación de tal analizador para el lenguaje Pascal–. El analizador léxico, también conocido como lexer, es responsable de la tokenización del código fuente, proceso mediante el cual el texto de entrada se divide en una secuencia de tokens que serán utilizados en las etapas subsiguientes del análisis sintáctico.

En este reporte, se presenta la metodología empleada para la identificación y clasificación de los distintos tokens definidos en la gramática de Pascal–. Se discuten los desafíos encontrados, como el manejo de literales, identificadores y palabras reservadas, así como la implementación de autómatas finitos para el reconocimiento de patrones en el código fuente.

1.2. Notación

Las máquinas de estado finito determinista (DFA) son modelos de computación que realizan cálculos en respuesta a una secuencia de entradas y cambios de estado. Las expresiones regulares son patrones que describen conjuntos de cadenas y se utilizan para identificar los tokens del lenguaje. Las tablas de transición representan cómo el lexer cambia de un estado a otro al procesar la entrada.

1.2.1. Modelo utilizado para fase de análisis y diseño

El modelo seleccionado para el desarrollo de las fases de análisis y diseño es el de FSM debido a su eficiencia y claridad en la representación de los lexemas del lenguaje.

1.2.2. ¿Por qué utilizar este modelo?

La justificación de este modelo radica en su capacidad para descomponer el proceso de análisis léxico en componentes manejables y su facilidad de implementación.

1.2.3. ¿Por qué utilizar este lenguaje de programación (Python)?

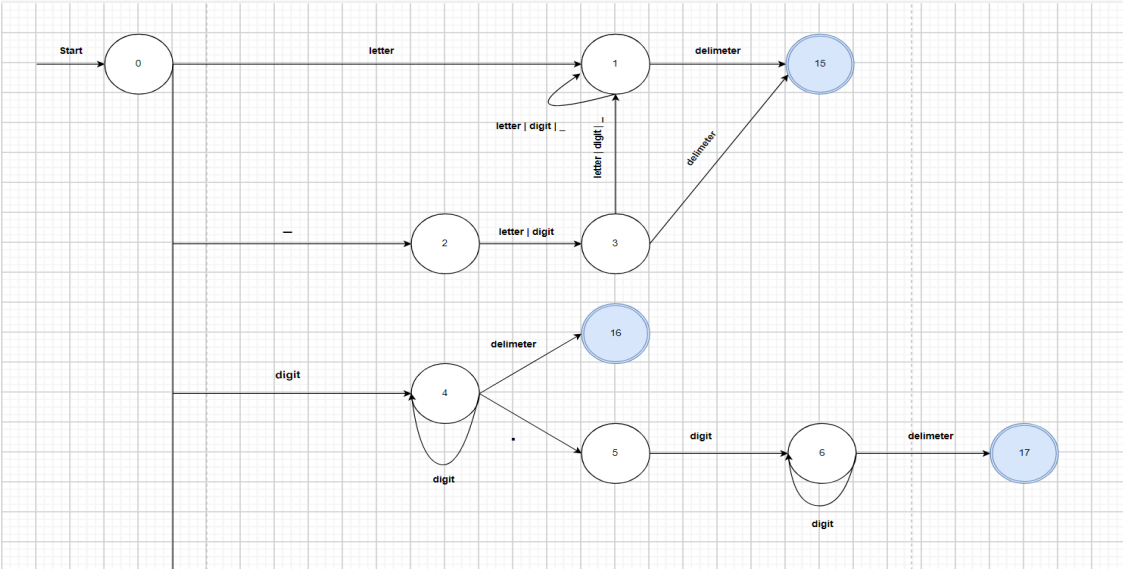
El lenguaje de programación elegido para la implementación es Python, ya que ofrece flexibilidad, manejo dinámico de las estructuras de datos y adicionalmente es un lenguaje que he usado por un buen tiempo y me

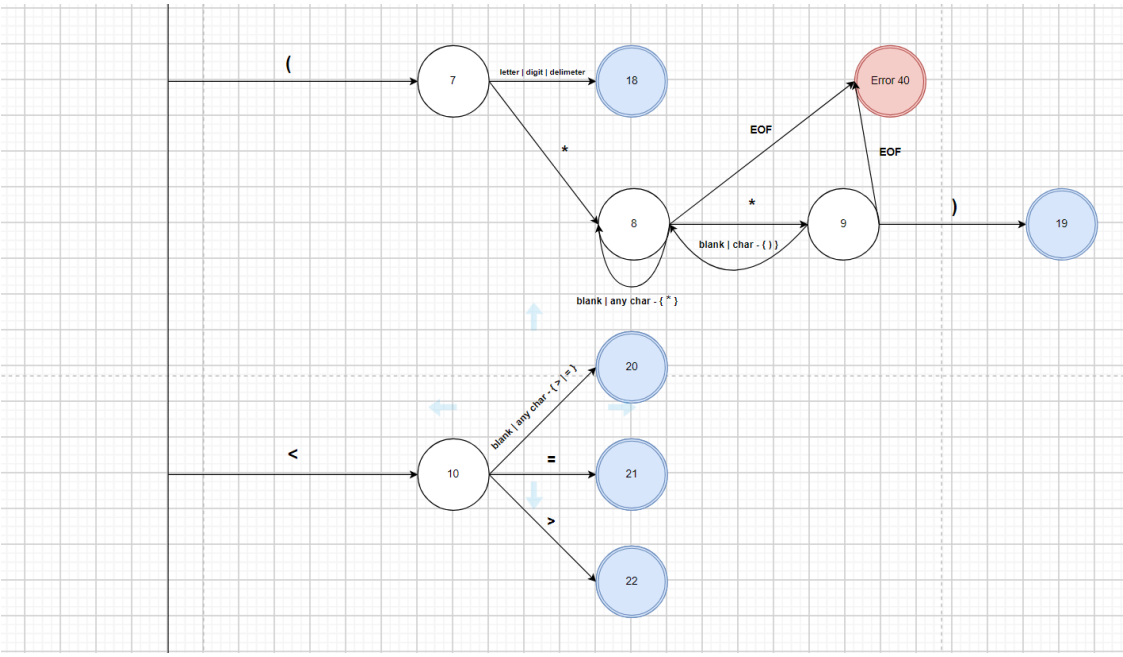
siento cómoda usándolo,todo lo anterior sumado facilita el desarrollo y la depuración del lexer.

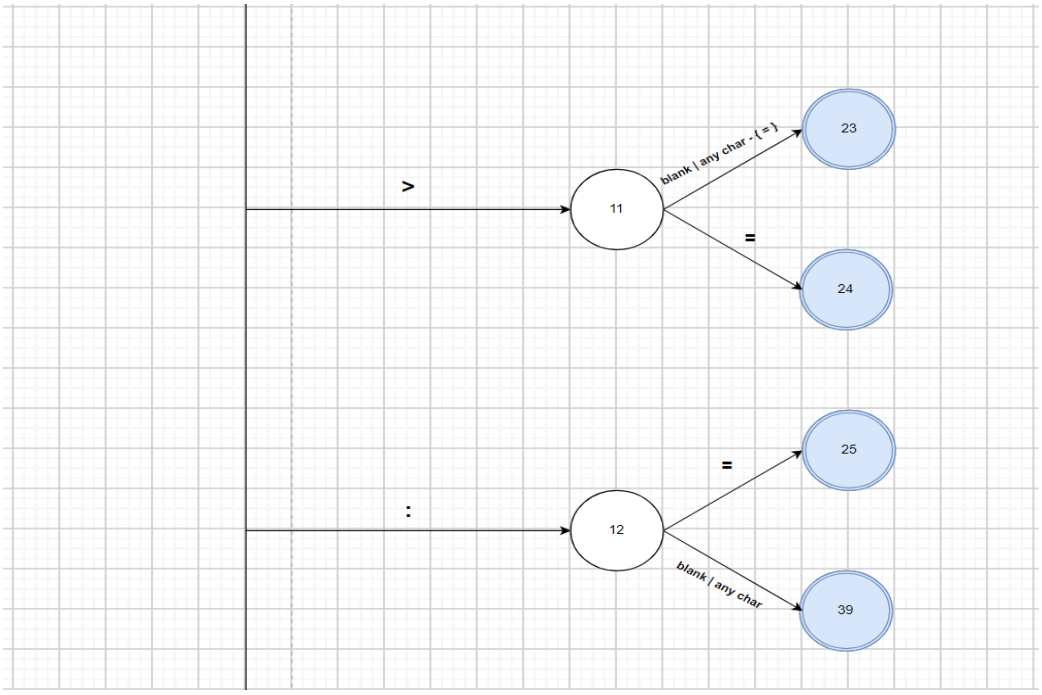
2. Análisis

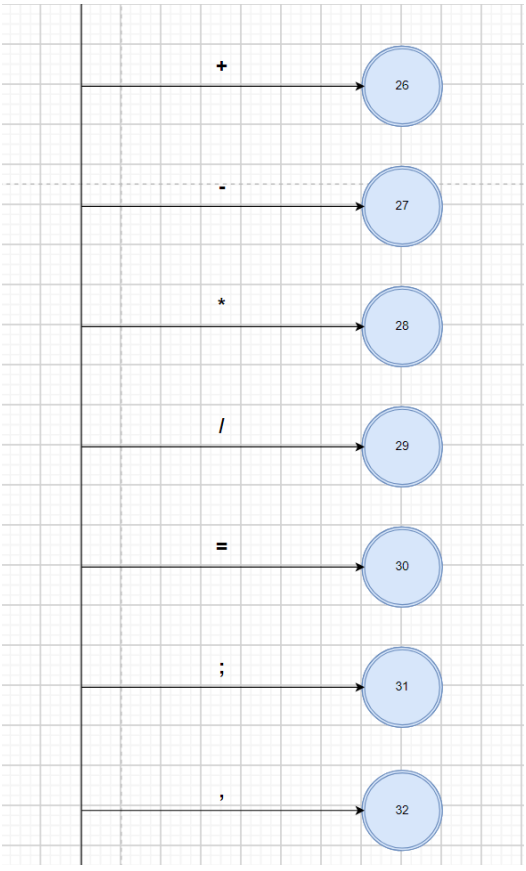
2.1. DFA

Link a DFA: [DFA.drawio]






>
11
blank | any char - { = }
23
=
24
:
12
=
25
blank | any char
39


+
26
-
27
*
28
/
29
=
30
;
31
,
32

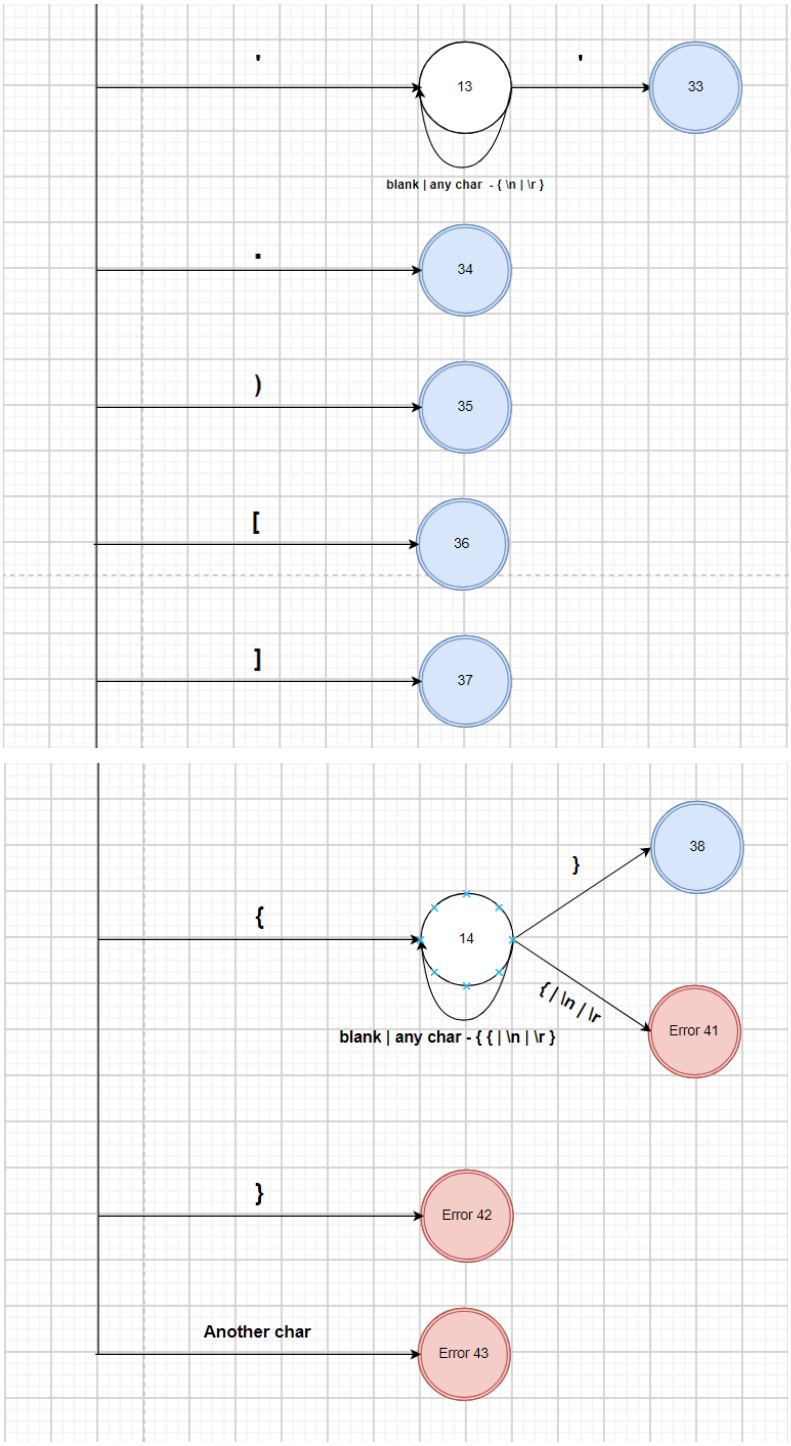

'
13
'
33
blank | any char - { \n | \r }
.
34
)
35
[
36
]
37
{
14
}
38
{ | \n | \r
Error 41
blank | any char - { { | \n | \r }
}
Error 42
Another char
Error 43

Figura 2.1.1

2.2. Tabla de transiciones

Link a tabla de transiciones: TRANSITION TABLE

STATE / INPUT CHAR	Letter	Digit	_	.	(	*	)	<	=	>	:	+	-	/	;	,	'	.	[	]	{	}	\n	\r	Blank	EOF	Another char
0	1	4	2	34	7	28	35	10	30	11	12	26	27	29	31	32	13	34	36	37	14	42	0	0	0	0	43
1	1	1	1	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	43
2	3	3	43	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	43
3	1	1	1	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	43
4	16	4	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	5	16	16	16	16	16	16	16	16	43
5	17	6	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	43
6	17	6	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	43
7	18	18	18	18	18	8	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	43
8	8	8	8	8	8	9	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	40	8
9	8	8	8	8	8	8	19	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	40	8
10	20	20	20	20	20	20	20	20	21	22	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	43
11	23	23	23	23	23	23	23	23	24	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	43
12	39	39	39	39	39	39	39	39	25	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	43
13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	33	13	13	13	13	13	13	13	13	13	13
14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	41	38	41	41	14	41	14

Figura 2.2.1

Estados aceptores:

15	Identifier
16	Integer
17	Real
18	(
19	Multiline comment
20	<
21	<=
22	<>
23	>
24	>=
25	:=
26	+
27	-
28	*
29	/
30	=
31	;
32	,
33	'
34	.
35	)
36	[
37	]
38	Single line comment
39	Type assigment =

Figura 2.2.2

Estados de error:

40	EOF
41	Single line comment error
42	Open curly brace missing error
43	Invalid char error

Figura 2.2.3

2.3. Tabla de tokens ID

Para la tabla de tokens opté por usar strings como ID para facilitar la parte de debug en la fase de implementación del algoritmo para procesar la tabla de transiciones.

```
keywords = {
    'array': 'array',
    'begin': 'begin',
    'do': 'do',
    'else': 'else',
    'end': 'end',
    'for': 'for',
    'function': 'function',
    'if': 'if',
    'integer': 'integer',
    'of': 'of',
    'procedure': 'procedure',
    'program': 'program',
    'readLn': 'readLn',
    'real': 'real',
    'repeat': 'repeat',
    'string': 'string',
    'then': 'then',
    'to': 'to',
    'until': 'until',
    'var': 'var',
    'writeLn': 'writeLn',
}

operators = {
    '-': 'minus',
    ':': 'type_assign',
    ':=': 'val_assign',
    '*': 'mult',
    '/': 'div',
    '+': 'plus',
    '<': 'less_than',
    '<=': 'less_eq_than',
    '<>': 'not_eq',
    '=': 'eq',
```

```
    '>': 'more_than',
    '>=': 'more_eq_than',
}

punctuation = {
    ',': 'comma',
    ';': 'semi-colon',
    '.': 'dot',
    '(': 'open_paren',
    ')': 'close_paren',
    '[': 'open_brackets',
    ']': 'close_brackets',
    "'": 'single_quote',
}
```

Figura 2.3.1

2.4. Expresiones regulares

Identificadores:

```
([A-Za-z]|_)([A-Za-z0-9_])*
```

Números:

```
[0-9]+.[0-9]+|[0-9]+
```

Strings:

```
\'([^\\\']|\\.)*\'
```

Single line comment:

```
\/\/.*$
```

Multi line comment

```
Python
\(\*[\s\S]*?\*\)
```

2.5. Tabla de símbolos

2.5.1. Descripción de las tablas usadas

La implementación hace uso de un **hashmap** para guardar los cuatro tipos de tablas de símbolos, las cuales incluyen identificadores, strings, integers y reales. Una diferencia significativa es que la tabla de símbolos para los identificadores hace uso de un **Set**, para evitar identificadores duplicados.

Los elementos guardados en las tablas son de clase **Symbol**, cuya implementación hace override de los métodos hash y equal para revisar que dos símbolos son equivalentes si su tipo y contenido son equivalentes, ignorando sus IDs, ya que estos pueden ser distintos, como cuando aparece un identificador usado múltiples veces.

Las tablas utilizadas para los integers, strings y reales son simples listas de Python, ya que en este caso no hay necesidad de evitar duplicados, ya que sus valores pueden aparecer en múltiples partes del mismo código fuente.

2.5.2. Método de gestión de las tablas

No se realiza un manejo especial de las tablas más allá de distinguir si su implementación usa un set, como en el caso de los identificadores, o una simple lista. La clase **TokenizerResult** contiene el stream de tokens y la tabla de símbolos.

Esta última está tipada como un diccionario con llaves de tipo string y valores de tipo **Set[Symbol]** o **List[Symbol]**, como una unión de tipos. Cada vez que se va a insertar un nuevo símbolo se requiere hacer una revisión de tipo en tiempo de ejecución, usando la función isinstance para asegurarse que los métodos correctos se usan para la inserción, como add y append según sea el caso.

2.6. Elementos del lenguaje de programación (Pascal--)

```
Unset
### Palabras Reservadas
     Las palabras reservadas del lenguaje son:
          - 'program'
          - 'procedure'
          - 'function`
          - 'begin'
          - 'end'
          - 'var'
          - 'integer'
          - 'real'
          - 'string'
          - 'array'
          - 'of'
          - 'if'
          - 'then'
          - 'else'
          - 'repeat'
          - 'until'
          - 'for'
          - 'to'
          - 'do'
          - 'readLn'
          - 'writeLn'

     Estas palabras pueden escribirse en minúsculas o
     mayúsculas indistintamente.

### Símbolos Especiales
     Los símbolos especiales incluyen:
          - '+' : adición
          - '-' : sustracción
          - '*' : multiplicación
          - '/' : división
          - '<' : menor que
```

- '<=' : menor o igual que
- '>' : mayor que
- '>=' : mayor o igual que
- '=' : igual
- '<>' : diferente
- ':=' : asignación
- ':' : dos puntos
- ';' : punto y coma
- ',' : coma
- ''' : comilla simple
- '.' : punto
- '(' : paréntesis que abre
- ')' : paréntesis que cierra
- '[' : corchete que abre
- ']' : corchete que cierra
- '{' : comienza comentario de una línea
- '}' : termina comentario de una línea
- '(*' : comienza comentario de múltiples líneas
- '*)' : termina comentario de múltiples líneas

Identificadores
Los identificadores se componen de letras, dígitos y guiones bajos (_), con las siguientes restricciones:

- No son sensibles a mayúsculas y minúsculas.
- El primer carácter debe ser una letra o un guión bajo. Si comienza con un guión bajo, debe seguir al menos una letra o un dígito.
- No pueden comenzar con un dígito.
- Ejemplos válidos: `abc`, `ABC`, `_a9B2`, `_123`, `abc_123`, `_a12_ _23`
- Ejemplos no válidos: `8b23abc`, `__abc`

Números

Los números pueden ser enteros o reales, como `1234` o `1234.5678`. Los números reales deben tener parte entera y parte decimal.

Cadenas de Texto
Las cadenas de texto son secuencias de caracteres entre comillas simples (' ') y no son sensibles a mayúsculas y minúsculas. Ejemplos: `'aB12%$*cv'`, `'*?._23s#@we'`

Espacios en Blanco
Los espacios en blanco incluyen espacios, saltos de línea y tabulaciones. Se ignoran pero deben ser reconocidos y actúan como delimitadores junto con identificadores, cadenas, números, palabras reservadas y símbolos especiales.

Comentarios
Los comentarios pueden aparecer donde sea posible el espacio en blanco y no pueden estar dentro de tokens. Pueden incluir cualquier carácter y ser de más de una línea. Hay dos formas de escribir comentarios:

1. Para comentarios de una sola línea, se pueden encerrar con (* ... *) o { ... }. Si se usan { ... }, no pueden incluir llaves dentro.
- Ejemplo válido: {esto es un comentario válido de una sola línea}
- Ejemplo no válido: {esto no es un comentario válido de una sola{ línea}

2. Para comentarios de múltiples líneas, deben encerrarse solo con (* ... *).
- Ejemplo válido: (* Esto es un comentario válido de múltiples líneas. *)

3. Diseño

En la (Figura 3.1) se puede apreciar el pseudocódigo genérico para procesar una tabla de transiciones y lo utilicé como base para el desarrollo del código que se muestra en la sección 4. Implementación, subsección 4.1. Código fuente(p. 15)

```
ch = next_input_character();
while (!EOF) {
    state = 0;  /* start DFA */
    while (!Accept[state] && !Error[state] ) {
        state = T[state, ch];
        if (Advance[state, ch])              /* See if scanner can get new char */
            ch = next_input_character();     /* given current state and char.   */
    } /* end of DFA */
    if (Accept[state] )
        record_Token();
    else
        error_message();
} /* end of FILE */
```

Figura 3.1

3.1. Implementación del DFA

El DFA(Figura 2.1.1) fue creado a partir de la simplificación de un lenguaje de programación, en este caso, de Pascal– y posteriormente se creó la equivalencia del DFA en una tabla de transiciones(Figura 2.2.1, 2.2.2 y 2.2.3), la tabla de transiciones si tuvo una implementación en el código fuente, se utilizó la estructura de datos: tuplas para representarla:

```
# (Current State, Input Character Class): Next State
transition_table = {
    (0, 'letter'): 1,
    (0, 'digit'): 4,
    (0, '_'): 2,
    (0, '.'): 34,
#etc...
}
```

3.2. Implementación de la tabla de tokens ID

Como se mencionó anteriormente en la sección 2.3. Tabla de tokens ID (Figura 2.3.1) opté por usar strings como ID para facilitar la parte de debug en la fase de implementación del algoritmo para procesar la tabla de transiciones y adicionalmente, implementé estas tablas con la estructura de datos: diccionario para representarla (para visualizarla se invita a ir a la sección y figura mencionada el inicio del párrafo).

3.3. Implementación de la tabla de símbolos

Así como se mencionó en la sección 2.5, la tabla de símbolos consiste en una hashmap con llaves de tipo string y valores de tipo List[Symbol] o Set[Symbol] según sea necesario. En el caso de los identificadores se utiliza un Set, así como implementaciones personalizadas de hash y equals para que los identificadores no se dupliquen en la tabla final.

Sin embargo en el caso de los integers, reales y strings estos son solo listas regulares de Python que hace uso de un array dinámico para crecer.

4. Implementación

4.1. Scanner,py documentación

Clase Symbol

La clase Symbol representa una entrada en la tabla de símbolos para un compilador o intérprete. Cada símbolo tiene un identificador único (id), un nombre (name) y un tipo (type). Los métodos especiales __hash__, __eq__ y __str__ permiten que los objetos Symbol se utilicen en estructuras de datos como conjuntos y diccionarios, y proporcionan una representación de cadena legible.

Diccionarios de Tokens

keywords: Contiene las palabras clave del lenguaje y sus nombres de tokens correspondientes.
operators: Mapea los operadores a sus nombres de tokens.
punctuation: Asocia caracteres de puntuación con sus nombres de tokens.

Tabla de Transición

transition_table: Define una tabla de transición para un autómata finito que se utiliza en el análisis léxico. Mapea pares de estado actual y clase de carácter de entrada al siguiente estado.

Estados Finales y de Error

final_states: Mapea los estados finales del autómata a los tipos de tokens que representan.
error_states: Enumera los estados de error y los tipos de errores asociados.

Funciones de Utilidad
is_keyword: Verifica si una palabra es una palabra clave.
is_operator: Verifica si una palabra es un operador.
is_punctuation: Verifica si una palabra es un carácter de puntuación.
is_digit: Verifica si una palabra es un dígito.
is_whitespace: Verifica si una palabra es un espacio en blanco.
get_char_type: Determina el tipo de carácter (espacio en blanco, letra, operador, etc.).
read_file: Lee el contenido de un archivo.
create_token: Crea un token basado en el lexema y el estado.

Clase TokenizerResult
La clase TokenizerResult almacena el resultado del proceso de tokenización. Contiene una lista de tokens y una tabla de símbolos que mapea los tipos de tokens a listas o conjuntos de objetos Symbol.

Función tokenize
La función tokenize convierte el código fuente en tokens y entradas de la tabla de símbolos. Utiliza la tabla de transición para procesar cada carácter y determinar el estado siguiente. Si se encuentra en un estado de error, imprime un mensaje de error y termina la ejecución. Si alcanza un estado final, crea un token y lo agrega a la lista de tokens y a la tabla de símbolos correspondiente.

Función reset_lexeme
La función reset_lexeme determina si el lexema actual debe reiniciarse al hacer la transición de un estado a otro en la máquina de estados. Esto es necesario para manejar correctamente los comentarios, operadores de dos caracteres y números reales.

Función lexer
La función lexer abre un archivo, tokeniza su contenido y devuelve el resultado. Utiliza la función read_file para leer el contenido del archivo y tokenize para realizar la tokenización.

Función main
La función main es el punto de entrada del programa. Toma argumentos de la línea de comandos y, si se proporciona un nombre de archivo, llama a la función lexer para tokenizar ese archivo. Luego imprime el flujo de tokens resultante.

4.2. Código fuente

```
import os
import sys
```

```python
from typing import List

# Symbol table entry class
class Symbol:
    def __init__(self, id: int, name: str, type: str):
        self.id = id
        self.name = name
        self.type = type
    def __hash__(self) -> int:
        return hash((self.name, self.type))
    def __eq__(self, other) -> bool:
        if not isinstance(other, Symbol):
            return False
        return (self.name, self.type) == (other.name, other.type)
    def __str__(self) -> str:
        return f"{self.id}: {self.name} - {self.type}"

# Keywords and their corresponding token names.
keywords = {
    'array': 'array',
    'begin': 'begin',
    'do': 'do',
    'else': 'else',
    'end': 'end',
    'for': 'for',
    'function': 'function',
    'if': 'if',
    'integer': 'integer',
    'of': 'of',
    'procedure': 'procedure',
    'program': 'program',
    'readLn': 'readLn',
    'real': 'real',
    'repeat': 'repeat',
    'string': 'string',
    'then': 'then',
    'to': 'to',
    'until': 'until',
    'var': 'var',
```

```
    'writeLn': 'writeLn',
}

# Operators and their corresponding token names.
operators = {
    '-': 'minus',
    ':': 'type_assign',
    ':=': 'val_assign',
    '*': 'mult',
    '/': 'div',
    '+': 'plus',
    '<': 'less_than',
    '<=': 'less_eq_than',
    '<>': 'not_eq',
    '=': 'eq',
    '>': 'more_than',
    '>=': 'more_eq_than',
}

# Punctuation characters and their corresponding token names.
punctuation = {
    ',': 'comma',
    ';': 'semi-colon',
    '.': 'dot',
    '{': 'open_curly',
    '}': 'close_curly',
    '(': 'open_paren',
    ')': 'close_paren',
    '[': 'open_brackets',
    ']': 'close_brackets',
    "'": 'single_quote',
}

# Transition table --> (Current State, Input Character Class): Next
State.
transition_table = {
    (0, 'letter'): 1,
    (0, 'digit'): 4,
    (0, '_'): 2,
```

```
    (0, '.'): 34,
    (0, '('): 7,
    (0, '*'): 28,
    (0, ')'): 35,
    (0, '<'): 10,
    (0, '='): 30,
    (0, '>'): 11,
    (0, ':'): 12,
    (0, '+'): 26,
    (0, '-'): 27,
    (0, '/'): 29,
    (0, ';'): 31,
    (0, ','): 32,
    (0, "'"): 13,
    (0, '.'): 34,
    (0, '['): 36,
    (0, ']'): 37,
    (0, '{'): 14,
    (0, '}'): 42,
    (0, 'whitespace'): 0,
    (0, 'eof'): 0,
    (0, 'other'): 43,

    (1, 'letter'): 1,
    (1, 'digit'): 1,
    (1, '_'): 1,
    (1, '.'): 15,
    (1, '('): 15,
    (1, '*'): 15,
    (1, ')'): 15,
    (1, '<'): 15,
    (1, '='): 15,
    (1, '>'): 15,
    (1, ':'): 15,
    (1, '+'): 15,
    (1, '-'): 15,
    (1, '/'): 15,
    (1, ';'): 15,
    (1, ','): 15,
```

```
    (1, "'"): 15,
    (1, '.'): 15,
    (1, '['): 15,
    (1, ']'): 15,
    (1, '{'): 15,
    (1, '}'): 15,
    (1, 'whitespace'): 15,
    (1, 'eof'): 15,
    (1, 'other'): 43,

    (2, 'letter'): 3,
    (2, 'digit'): 3,
    (2, '_'): 43,
    (2, '.'): 15,
    (2, '('): 15,
    (2, '*'): 15,
    (2, ')'): 15,
    (2, '<'): 15,
    (2, '='): 15,
    (2, '>'): 15,
    (2, ':'): 15,
    (2, '+'): 15,
    (2, '-'): 15,
    (2, '/'): 15,
    (2, ';'): 15,
    (2, ','): 15,
    (2, "'"): 15,
    (2, '.'): 15,
    (2, '['): 15,
    (2, ']'): 15,
    (2, '{'): 15,
    (2, '}'): 15,
    (2, 'whitespace'): 15,
    (2, 'eof'): 15,
    (2, 'other'): 43,

    (3, 'letter'): 1,
    (3, 'digit'): 1,
    (3, '_'): 1,
```

```
    (3, '.'): 15,
    (3, '('): 15,
    (3, '*'): 15,
    (3, ')'): 15,
    (3, '<'): 15,
    (3, '='): 15,
    (3, '>'): 15,
    (3, ':'): 15,
    (3, '+'): 15,
    (3, '-'): 15,
    (3, '/'): 15,
    (3, ';'): 15,
    (3, ','): 15,
    (3, "'"): 15,
    (3, '.'): 15,
    (3, '['): 15,
    (3, ']'): 15,
    (3, '{'): 15,
    (3, '}'): 15,
    (3, 'whitespace'): 15,
    (3, 'eof'): 15,
    (3, 'other'): 43,

    (4, 'letter'): 16,
    (4, 'digit'): 4,
    (4, '_'): 16,
    (4, '.'): 16,
    (4, '('): 16,
    (4, '*'): 16,
    (4, ')'): 16,
    (4, '<'): 16,
    (4, '='): 16,
    (4, '>'): 16,
    (4, ':'): 16,
    (4, '+'): 16,
    (4, '-'): 16,
    (4, '/'): 16,
    (4, ';'): 16,
    (4, ','): 16,
```

```
    (4, "'"): 16,
    (4, '.'): 5,
    (4, '['): 16,
    (4, ']'): 16,
    (4, '{'): 16,
    (4, '}'): 16,
    (4, 'whitespace'): 16,
    (4, 'eof'): 16,
    (4, 'other'): 43,

    (5, 'letter'): 17,
    (5, 'digit'): 6,
    (5, '_'): 17,
    (5, '.'): 17,
    (5, '('): 17,
    (5, '*'): 17,
    (5, ')'): 17,
    (5, '<'): 17,
    (5, '='): 17,
    (5, '>'): 17,
    (5, ':'): 17,
    (5, '+'): 17,
    (5, '-'): 17,
    (5, '/'): 17,
    (5, ';'): 17,
    (5, ','): 17,
    (5, "'"): 17,
    (5, '.'): 17,
    (5, '['): 17,
    (5, ']'): 17,
    (5, '{'): 17,
    (5, '}'): 17,
    (5, 'whitespace'): 17,
    (5, 'eof'): 17,
    (5, 'other'): 43,

    (6, 'letter'): 17,
    (6, 'digit'): 6,
    (6, '_'): 17,
```

```
    (6, '.'): 17,
    (6, '('): 17,
    (6, '*'): 17,
    (6, ')'): 17,
    (6, '<'): 17,
    (6, '='): 17,
    (6, '>'): 17,
    (6, ':'): 17,
    (6, '+'): 17,
    (6, '-'): 17,
    (6, '/'): 17,
    (6, ';'): 17,
    (6, ','): 17,
    (6, "'"): 17,
    (6, '.'): 17,
    (6, '['): 17,
    (6, ']'): 17,
    (6, '{'): 17,
    (6, '}'): 17,
    (6, 'whitespace'): 17,
    (6, 'eof'): 17,
    (6, 'other'): 43,

    (7, 'letter'): 18,
    (7, 'digit'): 18,
    (7, '_'): 18,
    (7, '.'): 18,
    (7, '('): 18,
    (7, '*'): 8,
    (7, ')'): 18,
    (7, '<'): 18,
    (7, '='): 18,
    (7, '>'): 18,
    (7, ':'): 18,
    (7, '+'): 18,
    (7, '-'): 18,
    (7, '/'): 18,
    (7, ';'): 18,
    (7, ','): 18,
```

```
    (7, "'"): 18,
    (7, '.'): 18,
    (7, '['): 18,
    (7, ']'): 18,
    (7, '{'): 18,
    (7, '}'): 18,
    (7, 'whitespace'): 18,
    (7, 'eof'): 18,
    (7, 'other'): 43,

    (8, 'letter'): 8,
    (8, 'digit'): 8,
    (8, '_'): 8,
    (8, '.'): 8,
    (8, '('): 8,
    (8, '*'): 9,
    (8, ')'): 8,
    (8, '<'): 8,
    (8, '='): 8,
    (8, '>'): 8,
    (8, ':'): 8,
    (8, '+'): 8,
    (8, '-'): 8,
    (8, '/'): 8,
    (8, ';'): 8,
    (8, ','): 8,
    (8, "'"): 8,
    (8, '.'): 8,
    (8, '['): 8,
    (8, ']'): 8,
    (8, '{'): 8,
    (8, '}'): 8,
    (8, 'whitespace'): 8,
    (8, 'eof'): 40,
    (8, 'other'): 8,

    (9, 'letter'): 8,
    (9, 'digit'): 8,
    (9, '_'): 8,
```

```
    (9, '.'): 8,
    (9, '('): 8,
    (9, '*'): 9,
    (9, ')'): 19,
    (9, '<'): 8,
    (9, '='): 8,
    (9, '>'): 8,
    (9, ':'): 8,
    (9, '+'): 8,
    (9, '-'): 8,
    (9, '/'): 8,
    (9, ';'): 8,
    (9, ','): 8,
    (9, "'"): 8,
    (9, '.'): 8,
    (9, '['): 8,
    (9, ']'): 8,
    (9, '{'): 8,
    (9, '}'): 8,
    (9, 'whitespace'): 8,
    (9, 'eof'): 40,
    (9, 'other'): 8,

    (10, 'letter'): 20,
    (10, 'digit'): 20,
    (10, '_'): 20,
    (10, '.'): 20,
    (10, '('): 20,
    (10, '*'): 20,
    (10, ')'): 20,
    (10, '<'): 20,
    (10, '='): 21,
    (10, '>'): 22,
    (10, ':'): 20,
    (10, '+'): 20,
    (10, '-'): 20,
    (10, '/'): 20,
    (10, ';'): 20,
    (10, ','): 20,
```

```
    (10, "'"): 20,
    (10, '.'): 20,
    (10, '['): 20,
    (10, ']'): 20,
    (10, '{'): 20,
    (10, '}'): 20,
    (10, 'whitespace'): 20,
    (10, 'eof'): 20,
    (10, 'other'): 43,

    (11, 'letter'): 23,
    (11, 'digit'): 23,
    (11, '_'): 23,
    (11, '.'): 23,
    (11, '('): 23,
    (11, '*'): 23,
    (11, ')'): 23,
    (11, '<'): 23,
    (11, '='): 24,
    (11, '>'): 23,
    (11, ':'): 23,
    (11, '+'): 23,
    (11, '-'): 23,
    (11, '/'): 23,
    (11, ';'): 23,
    (11, ','): 23,
    (11, "'"): 23,
    (11, '.'): 23,
    (11, '['): 23,
    (11, ']'): 23,
    (11, '{'): 23,
    (11, '}'): 23,
    (11, 'whitespace'): 23,
    (11, 'eof'): 23,
    (11, 'other'): 43,

    (12, 'letter'): 39,
    (12, 'digit'): 39,
    (12, '_'): 39,
```

```
    (12, '.'): 39,
    (12, '('): 39,
    (12, '*'): 39,
    (12, ')'): 39,
    (12, '<'): 39,
    (12, '='): 25,
    (12, '>'): 39,
    (12, ':'): 39,
    (12, '+'): 39,
    (12, '-'): 39,
    (12, '/'): 39,
    (12, ';'): 39,
    (12, ','): 39,
    (12, "'"): 39,
    (12, '.'): 39,
    (12, '['): 39,
    (12, ']'): 39,
    (12, '{'): 39,
    (12, '}'): 39,
    (12, 'whitespace'): 39,
    (12, 'eof'): 39,
    (12, 'other'): 43,

    (13, 'letter'): 13,
    (13, 'digit'): 13,
    (13, '_'): 13,
    (13, '.'): 13,
    (13, '('): 13,
    (13, '*'): 13,
    (13, ')'): 13,
    (13, '<'): 13,
    (13, '='): 13,
    (13, '>'): 13,
    (13, ':'): 13,
    (13, '+'): 13,
    (13, '-'): 13,
    (13, '/'): 13,
    (13, ';'): 13,
    (13, ','): 13,
```

```
    (13, "'"): 33,
    (13, '.'): 13,
    (13, '['): 13,
    (13, ']'): 13,
    (13, '{'): 13,
    (13, '}'): 13,
    (13, 'whitespace'): 13,
    (13, 'eof'): 13,
    (13, 'other'): 13,

    (14, 'letter'): 14,
    (14, 'digit'): 14,
    (14, '_'): 14,
    (14, '.'): 14,
    (14, '('): 14,
    (14, '*'): 14,
    (14, ')'): 14,
    (14, '<'): 14,
    (14, '='): 14,
    (14, '>'): 14,
    (14, ':'): 14,
    (14, '+'): 14,
    (14, '-'): 14,
    (14, '/'): 14,
    (14, ';'): 14,
    (14, ','): 14,
    (14, "'"): 34,
    (14, '.'): 14,
    (14, '['): 14,
    (14, ']'): 14,
    (14, '{'): 41,
    (14, '}'): 38,
    (14, 'whitespace'): 14,
    (14, 'eof'): 41,
    (14, 'other'): 14,
}

# Final states of transition table --> State: Token Type.
final_states = {
```

```
    15: 'identifier',
    16: 'integer',
    17: 'real',
    18: 'open_parens',
    19: 'multiline_comment',
    20: 'less_than',
    21: 'less_eq_than',
    22: 'not_eq',
    23: 'more_than',
    24: 'more_eq_than',
    25: 'val_assign',
    26: 'plus',
    27: 'minus',
    28: 'mult',
    29: 'div',
    30: 'eq',
    31: 'semicolon',
    32: 'comma',
    33: 'string',
    34: 'point',
    35: 'closed_parens',
    36: 'open_sqre_bracket',
    37: 'closed_sqre_bracket',
    38: 'single_line_comment',
    39: 'type_assignment',
}

# Error states of the transition tables --> State: Error type.
error_states = {
    40: 'eof',
    41: 'single_line_comment_error',
    42: 'open_curly_brace_error',
    43: 'invalid_char_error',
}

# Function to check if a word is a keyword.
def is_keyword(word: str):
    return word in keywords.keys()
```

```
# Function to check if a word is an operator.
def is_operator(word: str):
    return word in operators.keys()

# Function to check if a word is a punctuation character.
def is_punctuation(word: str):
    return word in punctuation.keys()

# Function to check if a word is a digit.
def is_digit(word: str):
    return str.isnumeric(word)

# Function to check if a word is whitespace.
def is_whitespace(word: str):
    return word.isspace()

# Function to clean the source file from comments and excessive
whitespace.
# (Removes comments and replace multiple whitespaces with a single
space)
def clean_file(source: str) -> str:
    # Remove comments
    comment_starters = ['{', '(*']
    comment_enders = ['}', '*)']
    for start, end in zip(comment_starters, comment_enders):
        while start in source:
            start_index = source.find(start)
            end_index = source.find(end, start_index)
              if end_index == -1:  # If the end of the comment is not
found
                    source = source[:start_index]  # Remove from start
to the end
            else:
                    source = source[:start_index] + source[end_index +
len(end):]

    # Replace multiple whitespaces with a single space
    result = []
    in_whitespace = False
```

```
    for char in source:
        if char.isspace():
            if not in_whitespace:
                result.append(' ')
                in_whitespace = True
        else:
            in_whitespace = False
            result.append(char)

    return ''.join(result)

# Function to determine the type of character (whitespace, letter,
operator, etc.).
def get_char_type(char: str):
    if is_whitespace(char):
        return "whitespace"
    if str.isalpha(char):
        return "letter"
    if is_operator(char):
        return char
    if is_punctuation(char):
        return char
    if is_digit(char):
        return "digit"

    return "other"

# Function to read the contents of a file.
def read_file(file_path):
    with open(file_path, 'r') as file:
        return file.read()

# Function to create a token based on the lexeme and state.
def create_token(lexeme: str, state: int):
    # Ignores comment final states
    if state in [38, 19]:
        return None
    if is_whitespace(lexeme):
        return None
```

```
    if is_keyword(lexeme):
        return (lexeme, 'keyword')
    if state in final_states:
        return (lexeme, final_states[state])
    else:
        return None

# Class to hold the result of the tokenization process.
class TokenizerResult:
       def  __init__(self,  tokens:  list[tuple[str,  str]],  symbols:
dict[str, List[Symbol] | set[Symbol]]):
        self.tokens = tokens
        self.symbols = symbols

# Function to tokenize the source code into tokens and symbol table
entries.
def tokenize(source: str) -> TokenizerResult:
     # Initialization of symbol table, state, lexeme, tokens, and
index.
    symbol_table: dict[str, List[Symbol] | set[Symbol]] = {
        'identifier': set([]),
        'real': [],
        'integer': [],
        'string': []
    }

    current_state = 0
    lexeme = ''
    tokens = []

    index = 0

    # Processing each character in the source code
    while (index < len(source)):
        char = source[index]
        char_type = get_char_type(char)
        transition = (current_state, char_type)
        next_state = transition_table[transition]
```

```
        # If in an error state print error
        if next_state in error_states:
            print(f"Invalid state found: {transition}")
            print(f"Error: {error_states[next_state]}")
            exit(1)

        if next_state in final_states:
                # hack that closes quotes, reason: missing delimiter
state
            if lexeme and char == "'":
                lexeme += char
                index += 1

            # two char operators, i.e: >= or :=
            if is_operator(lexeme) and is_operator(char):
                lexeme += char
                index += 1

            token = create_token(lexeme or char, next_state)
            if token:
                (name, type) = token

                symbol_list = symbol_table.get(type)
                if symbol_list is not None:
                    id = len(symbol_list) + 1
                    symbol = Symbol(id, name, type)
                    token = (name, type, id)

                    if isinstance(symbol_list, set):
                        symbol_list.add(symbol)
                    else:
                        symbol_list.append(symbol)

                tokens.append(token)

            # if lexeme was delimiter move to the next char
            # needed for operators and punctuation
            # if we just closed a comment move up as well
            if not lexeme:
```

```
                index += 1
            if next_state in [38, 19]:
                print("Comment: " + lexeme + char)
                index += 1

            lexeme = ''
            current_state = 0
            continue

        if reset_lexeme(current_state, next_state):
            lexeme = char
        else:
            lexeme += char

        current_state = next_state
        index += 1

    return TokenizerResult(tokens, symbol_table)

# Function to determine whether the current lexeme should be reset
when transitioning from one state to another in the state machine.
def reset_lexeme(current_state: int, next_state: int):
    # Moving from ( to * inside a comment
    if current_state == 7 and next_state == 8:
        return False
    # Moving from comment to *
    if current_state == 8 and next_state == 9:
        return False
    # Moving from * to ) in a comment
    if current_state == 9 and next_state == 19:
        return False
    # Moving digit to '.'
    if current_state == 4 and next_state == 5:
        return False
    # Moving from '.' to digit
    if current_state == 5 and next_state == 6:
        return False

    return current_state != next_state
```

```
# Opens file, tokenizes it and returns result
def lexer(filename) -> TokenizerResult:
    dir_path = os.path.dirname(os.path.realpath(__file__))
    file_path = os.path.join(dir_path, filename)
    source = read_file(file_path)
    result = tokenize(source)

    return result

def main():
    args = sys.argv
    if len(args) > 1:
        result = lexer(args[1])

        # Print the tokens in the token stream
        print("\nTOKEN STREAM")
        for token in result.tokens:
            print(token)

        # Print the contents of the symbol table
        print("\nSYMBOL TABLE")
        for type, symbols in result.symbols.items():
            print(f"\n{type.upper()}(s)")
            print("\nID | CONTENT | TYPE")

            for symbol in symbols:
                print(symbol)
    else:
        print("No file provided for lexing.")

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

5. Verificación y Validación

5.1. Verificación (“are we building the product right?”) y Validación (“are we building the right product?”)

Prueba 1:

```
Python
PS C:\Users\sofo-\OneDrive\Documentos\Compiladores\compiladores> python
.\Scanner.py .\Test1.txt
Comment: { Example #1 }
Comment: {  This is the typical "Hello World" }
Comment: (* This is the main program block *)
Comment: (* This is the end of the main
program block *)

TOKEN STREAM
('program', 'keyword')
('HelloWorld', 'identifier', 1)
(';', 'semicolon')
('+', 'plus')
('begin', 'keyword')
('writeLn', 'keyword')
('(', 'open_parens')
("' Hello World '", 'string', 1)
(')', 'closed_parens')
(';', 'semicolon')
('end', 'keyword')
('.', 'point')

SYMBOL TABLE

IDENTIFIER(s)

ID | CONTENT | TYPE
1: HelloWorld - identifier

REAL(s)

ID | CONTENT | TYPE
```

```
INTEGER(s)

ID | CONTENT | TYPE

STRING(s)

ID | CONTENT | TYPE
1: ' Hello World ' - string
```

Prueba 2:

```
Python
PS C:\Users\sofo-\OneDrive\Documentos\Compiladores\compiladores> python
.\Scanner.py .\Test2.txt
Comment: { Example #2 }
Comment: (* This is a procedure block*)
Comment: (* This is the main program block *)
Comment: (* This is the end of the main
program block *)

TOKEN STREAM
('program', 'keyword')
('Ejemplo2', 'identifier', 1)
(';', 'semicolon')
('var', 'keyword')
('a', 'identifier', 2)
(':', 'type_assignment')
('integer', 'keyword')
(';', 'semicolon')
('b', 'identifier', 3)
(':', 'type_assignment')
('real', 'keyword')
(';', 'semicolon')
('procedure', 'keyword')
('assign', 'identifier', 4)
('(', 'open_parens')
('x', 'identifier', 5)
(':', 'type_assignment')
('integer', 'keyword')
```

```
(';', 'semicolon')
('y', 'identifier', 6)
(':', 'type_assignment')
('real', 'keyword')
(')', 'closed_parens')
(';', 'semicolon')
('begin', 'keyword')
('a', 'identifier', 7)
(':=', 'val_assign')
('x', 'identifier', 7)
(';', 'semicolon')
('b', 'identifier', 7)
(':=', 'val_assign')
('y', 'identifier', 7)
(';', 'semicolon')
('end', 'keyword')
(';', 'semicolon')
('begin', 'keyword')
('assign', 'identifier', 7)
('(', 'open_parens')
('27', 'integer', 1)
(',', 'comma')
('3.1416', 'real', 1)
(')', 'closed_parens')
(';', 'semicolon')
('writeLn', 'keyword')
('(', 'open_parens')
("' a = '", 'string', 1)
(',', 'comma')
('a', 'identifier', 7)
(')', 'closed_parens')
(';', 'semicolon')
('writeLn', 'keyword')
('(', 'open_parens')
("' b = '", 'string', 2)
(',', 'comma')
('b', 'identifier', 7)
(')', 'closed_parens')
(';', 'semicolon')
('end', 'keyword')
('.', 'point')

SYMBOL TABLE
```

```
IDENTIFIER(s)

ID | CONTENT | TYPE
3: b - identifier
4: assign - identifier
1: Ejemplo2 - identifier
2: a - identifier
5: x - identifier
6: y - identifier

REAL(s)

ID | CONTENT | TYPE
1: 3.1416 - real

INTEGER(s)

ID | CONTENT | TYPE
1: 27 - integer

STRING(s)

ID | CONTENT | TYPE
1: ' a = ' - string
2: ' b = ' - string
```

Prueba 3:

```
Python
PS C:\Users\sofo-\OneDrive\Documentos\Compiladores\compiladores> python
.\Scanner.py .\Test3.txt
Comment: { Example #3 }
Comment: (* Var declaration section*)

TOKEN STREAM
('program', 'keyword')
('Ejemplo3', 'identifier', 1)
(';', 'semicolon')
('var', 'keyword')
```

```
('a', 'identifier', 2)
(',', 'comma')
('b', 'identifier', 3)
(':', 'type_assignment')
('integer', 'keyword')
(';', 'semicolon')
('x', 'identifier', 4)
(',', 'comma')
('y', 'identifier', 5)
(':', 'type_assignment')
('real', 'keyword')
(';', 'semicolon')
('n', 'identifier', 6)
(':', 'type_assignment')
('array', 'keyword')
('[', 'open_sqre_bracket')
('1.', 'real', 1)
('.', 'point')
('10', 'integer', 1)
(']', 'closed_sqre_bracket')
('of', 'keyword')
('integer', 'keyword')
(';', 'semicolon')
('s', 'identifier', 7)
(':', 'type_assignment')
('string', 'keyword')
(';', 'semicolon')
('function', 'keyword')
('calc', 'identifier', 8)
('(', 'open_parens')
('w', 'identifier', 9)
(',', 'comma')
('z', 'identifier', 10)
(':', 'type_assignment')
('real', 'keyword')
(')', 'closed_parens')
(':', 'type_assignment')
('integer', 'keyword')
(';', 'semicolon')
('begin', 'keyword')
('if', 'keyword')
('(', 'open_parens')
('w', 'identifier', 11)
('>=', 'more_eq_than')
```

```
('z', 'identifier', 11)
(')', 'closed_parens')
('then', 'keyword')
('calc', 'identifier', 11)
(':=', 'val_assign')
('5', 'integer', 2)
('else', 'keyword')
('calc', 'identifier', 11)
(':=', 'val_assign')
('0', 'integer', 3)
(';', 'semicolon')
('end', 'keyword')
(';', 'semicolon')
('procedure', 'keyword')
('arrayInit', 'identifier', 11)
('(', 'open_parens')
('w', 'identifier', 12)
(':', 'type_assignment')
('integer', 'keyword')
(';', 'semicolon')
('z', 'identifier', 12)
(':', 'type_assignment')
('real', 'keyword')
(')', 'closed_parens')
(';', 'semicolon')
('begin', 'keyword')
('for', 'keyword')
('i', 'identifier', 12)
(':=', 'val_assign')
('1', 'integer', 4)
('to', 'keyword')
('10', 'integer', 5)
('do', 'keyword')
('begin', 'keyword')
('n', 'identifier', 13)
('[', 'open_sqre_bracket')
('i', 'identifier', 13)
(']', 'closed_sqre_bracket')
(':=', 'val_assign')
('1', 'integer', 6)
('*', 'mult')
('5', 'integer', 7)
(';', 'semicolon')
('writeLn', 'keyword')
```

```
('(', 'open_parens')
("'n['", 'string', 1)
(',', 'comma')
('i', 'identifier', 13)
(',', 'comma')
("'] ='", 'string', 2)
(',', 'comma')
('n', 'identifier', 13)
('[', 'open_sqre_bracket')
('i', 'identifier', 13)
(']', 'closed_sqre_bracket')
(')', 'closed_parens')
(';', 'semicolon')
('end', 'keyword')
(';', 'semicolon')
('end', 'keyword')
('.', 'point')

SYMBOL TABLE

IDENTIFIER(s)

ID | CONTENT | TYPE
11: arrayInit - identifier
12: i - identifier
9: w - identifier
4: x - identifier
6: n - identifier
10: z - identifier
1: Ejemplo3 - identifier
7: s - identifier
3: b - identifier
8: calc - identifier
2: a - identifier
5: y - identifier

REAL(s)

ID | CONTENT | TYPE
1: 1. - real

INTEGER(s)

ID | CONTENT | TYPE
```

```
1: 10 - integer
2: 5 - integer
3: 0 - integer
4: 1 - integer
5: 10 - integer
6: 1 - integer
7: 5 - integer

STRING(s)

ID | CONTENT | TYPE
1: 'n[' - string
2: '] =' - string
```

Prueba 4:

```
Python
PS C:\Users\sofo-\OneDrive\Documentos\Compiladores\compiladores> python
.\Scanner.py .\Test4.txt

TOKEN STREAM
('procedure', 'keyword')
('assign', 'identifier', 1)
('(', 'open_parens')
('w', 'identifier', 2)
(',', 'comma')
('z', 'identifier', 3)
(':', 'type_assignment')
('real', 'keyword')
(')', 'closed_parens')
(';', 'semicolon')
('var', 'keyword')
('temp', 'identifier', 4)
(':', 'type_assignment')
('real', 'keyword')
(';', 'semicolon')
('begin', 'keyword')
('temp', 'identifier', 5)
(':=', 'val_assign')
('w', 'identifier', 5)
```

```
(';', 'semicolon')
('repeat', 'keyword')
('temp', 'identifier', 5)
(':=', 'val_assign')
('temp', 'identifier', 5)
('-', 'minus')
('z', 'identifier', 5)
(';', 'semicolon')
('until', 'keyword')
('(', 'open_parens')
('temp', 'identifier', 5)
('<=', 'less_eq_than')
('0', 'integer', 1)
(')', 'closed_parens')
(';', 'semicolon')
('if', 'keyword')
('(', 'open_parens')
('temp', 'identifier', 5)
('=', 'eq')
('0', 'integer', 2)
(')', 'closed_parens')
('then', 'keyword')
('begin', 'keyword')
('a', 'identifier', 5)
(':=', 'val_assign')
('10', 'integer', 3)
(';', 'semicolon')
('b', 'identifier', 6)
(':=', 'val_assign')
('20', 'integer', 4)
(';', 'semicolon')
('end', 'keyword')
('else', 'keyword')
('begin', 'keyword')
('a', 'identifier', 7)
(':=', 'val_assign')
('0', 'integer', 5)
(';', 'semicolon')
('b', 'identifier', 7)
(':=', 'val_assign')
('0', 'integer', 6)
(';', 'semicolon')
('end', 'keyword')
(';', 'semicolon')
```

```
('end', 'keyword')
(';', 'semicolon')
('begin', 'keyword')
('s', 'identifier', 7)
(':=', 'val_assign')
("'The end'", 'string', 1)
(';', 'semicolon')
('writeLn', 'keyword')
('(', 'open_parens')
("' x = '", 'string', 2)
(')', 'closed_parens')
(';', 'semicolon')
('readLn', 'keyword')
('(', 'open_parens')
('x', 'identifier', 8)
(')', 'closed_parens')
(';', 'semicolon')
('writeLn', 'keyword')
('(', 'open_parens')
("' y = '", 'string', 3)
(')', 'closed_parens')
(';', 'semicolon')
('readLn', 'keyword')
('(', 'open_parens')
('y', 'identifier', 9)
(')', 'closed_parens')
(';', 'semicolon')
('if', 'keyword')
('(', 'open_parens')
('calc', 'identifier', 10)
('(', 'open_parens')
('x', 'identifier', 11)
(',', 'comma')
('y', 'identifier', 11)
(')', 'closed_parens')
('=', 'eq')
('5', 'integer', 7)
(')', 'closed_parens')
('then', 'keyword')
('assign', 'identifier', 11)
('(', 'open_parens')
('x', 'identifier', 11)
(',', 'comma')
('y', 'identifier', 11)
```

```
(')', 'closed_parens')
('else', 'keyword')
('writeLn', 'keyword')
('(', 'open_parens')
('s', 'identifier', 11)
(')', 'closed_parens')
(';', 'semicolon')
('end', 'keyword')
('.', 'point')

SYMBOL TABLE

IDENTIFIER(s)

ID | CONTENT | TYPE
10: calc - identifier
1: assign - identifier
6: b - identifier
7: s - identifier
4: temp - identifier
3: z - identifier
9: y - identifier
5: a - identifier
2: w - identifier
8: x - identifier

REAL(s)

ID | CONTENT | TYPE

INTEGER(s)

ID | CONTENT | TYPE
1: 0 - integer
2: 0 - integer
3: 10 - integer
4: 20 - integer
5: 0 - integer
6: 0 - integer
7: 5 - integer

STRING(s)

ID | CONTENT | TYPE
```

```
1: 'The end' - string
2: ' x = ' - string
3: ' y = ' - string
```