# **Especificación de requerimientos de software**

**C—Compilador Lexer & Parser**

**Versión 1.0**

por Cesar Eduardo Flores Palacios

Tabla de contenido

[**Especificación de requerimientos de software** 1](#_Toc43245087)

[1. Introducción 3](#_Toc43245088)

[1.1 Notaciones 3](#_Toc43245089)

[2. Análisis 5](#_Toc43245090)

[2.1 Objetivos formales 5](#_Toc43245091)

[2.2 Especificaciones formales del lenguaje 5](#_Toc43245092)

[3. Diseño 7](#_Toc43245093)

[4. Implementación 10](#_Toc43245094)

[5. Verificación y validación 11](#_Toc43245095)

[References 12](#_Toc43245096)

# Introducción

En el siguiente reporte se describe el proceso de desarrollo de software para un compilador, primeramente, la fase del analizador léxico y posteriormente el analizador sintáctico correspondiente, desarrollando la especificación de las fases de análisis, diseño, implementación y pruebas para el software en cuestión.

### 1.1 Notaciones

El desarrollo de los analizadores y los conceptos escritos en esta sección, en su mayor parte están basados o se usan citas del libro *“Compiladores: principios, técnicas y herramientas”* o mejor conocido como el libro del dragón en el mundo de los compiladores, escrito por Alfred V. Aho, Ravi Sethi y Jeffrey D. Ullman. Otro de los recursos empleados para definiciones será el libro digital *de “Teoría de autómatas, lenguajes y computación.”* por John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman.

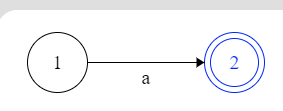
Para la primera fase que es el en analizador léxico es necesario entender los conceptos básicos que se implican, tales como una máquina de estados finitos, un diagrama de transiciones, expresiones regulares, y gramática independiente de contexto.

Analizador léxico: la fase de un compilador que se encarga de extraer los tokens de un programa fuente y los separa en tablas para su uso posterior.

Analizador sintáctico: recibe un grupo de elementos léxicos (tokens) que provienen de la fase previa al análisis sintáctico y comprueba si el conjunto de tokens recibidos puede ser generado por la estructura gramatical definida para ese lenguaje.

##### Máquina de estados finitos: es un programa que toma como entrada una cadena de caracteres y responde si la cadena ingresada cumple con las reglas de la máquina o no.

##### Diagrama de transiciones: es un modelo o diagrama de flujo usado para representar las reglas de respuesta de una máquina de estados finitos ante una entrada y, más enfocado en el diseño de un compilador “Los diagramas de transiciones representan las acciones que tienen lugar cuando el analizador léxico es llamado por el analizador sintáctico para obtener el siguiente componente léxico.” (Alfred V. Aho, Ravi Sethi y Jeffrey D. Ullman). Un ejemplo de diagrama de transiciones que reconoce la letra a como entrada y solo esa letra en la figura 1.



*Figura 1: diagrama de transiciones que reconoce la letra a.*

Expresiones regulares: son una notación para describir o aceptar patrones en una cadena de caracteres. Con una definición más técnica “las expresiones regulares ofrecen algo que los autómatas no proporcionan: una forma declarativa para expresar las cadenas que deseamos aceptar. Por tanto, las expresiones regulares sirven como lenguaje de entrada de muchos sistemas que procesan cadenas”. (John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman.)

Tablas de transiciones: es una representación mediante el uso de una tabla en la que la primera columna lista todos los estados de una maquina y en la primera fila se listan todas las posibles transiciones o símbolos de entrada aceptados por esa máquina.

Gramática independiente del contexto: “describe de forma natural la estructura jerárquica de la construcción de un lenguaje de programación” (John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman.). Como ejemplo de esta definición, Ullman y autores conjuntos en su libro ejemplifican la gramática usando la regla de producción para la estructura de la sentencia “if” y la ilustran de la siguiente manera.

*prop* -> **if** (*expr*) *prop* **else***prop*

El lenguaje de programación usado para el desarrollo del analizador léxico fue Python, por el motivo de que es un lenguaje en el que me siento más cómodo y siendo un proyecto académico fue la alternativa más viable.

# 2. Análisis

Durante el desarrollo del compilador se realiza un análisis de los requerimientos para tener una visión más objetiva de lo que debería ser desarrollado y cuales aspectos o características serian considerados como esenciales para su funcionamiento.

Como objetivo principal se estableció que el software a desarrollar debe de leer una cadena de caracteres y lograr identificar y separar los tokens adecuadamente en tablas que puedan ser interpretada por el analizador sintáctico. Luego el sistema deberá analizar cada token y verificar que la cadena concuerde con las reglas establecidas para la gramática libre de contexto. También detectará errores en el análisis y dará la correspondiente sentencia descriptiva ante el error.

### 2.1 Objetivos formales

Una vez definido el objetivo principal se define el conjunto de características más formales para el desarrollo.

#### 2.1.1 Objetivos léxicos

1. Se creará un autómata que identifique los símbolos establecidos en los requerimientos.
2. Se crearán las tablas y describirán los símbolos y su tipo.
3. Basado en el autómata creado se realizará la tabla de transiciones para representar el autómata del lenguaje.
4. Estructuración del código para interpretar la tabla de transiciones

#### 2.1.2 Objetivos sintácticos

1. En base a las especificaciones del lenguaje se creará una gramática independiente de contexto para validad el lenguaje.
2. Se obtendrán los first, follow y first+ sets para cada regla de la gramática.
3. Se generará el código usando la metodología de recursive procedures adaptandolo a las reglas y first+ sets.
4. Se listarán los mensajes de error posibles por el escáner léxico y el analizador sintáctico.

### 2.2 Especificaciones formales del lenguaje

Las especificaciones formales del lenguaje se definen a continuación.

#### 2.2.1 Especificaciones para escáner léxico

Las palabras reservadas que el analizador debe de reconocer son else, if, int, return, void, while, input y output.

Los símbolos que deberán de ser aceptados se listan a continuación:

1. + suma
2. - resta
3. \* factor
4. / division
5. < mayor que
6. < menor que
7. >= mayor o igual que
8. >= menor igual que
9. == igual
10. != diferente que
11. = asignacion
12. ; punto y coma
13. , coma
14. ( parentesis de apertura
15. ) parentesis de cierre
16. [ corchete de apertura
17. ] corchete de cierre
18. { llave de apertura
19. } llave de cierre
20. /\* comentario de apertura
21. \*/ comentario de cierre

Además de 2 tokens extras para reconocer un identificador y un numero bajo las siguientes características:

1. Los identificadores están conformados por un conjunto de letras de la A a la Z, minúsculas o mayúsculas.
2. Los números solo están conformados por la unión de los números del 0 al 9.

**Las expresiones regulares para distinguir identificadores y números serán:**

*identificador = letra+ y letra =* [a-zA-Z]

*numero = digito+ y digito = [0-9]*

El contenido dentro de los tokens de comentario debe de ser ignorado por el escáner, asimismo como espacios en blanco y derivados de este, como tabulaciones y saltos de línea.

#### 2.2.2 Especificaciones para analizador sintáctico

Para generar la gramática hay que tomar en consideración las especificación del lenguaje,

La declaración de una función consiste en definir un tipo de retorno, un identificador y una lista de parámetros separados por una coma encerrados por un grupo de paréntesis. Luego un compound statement que contendrá el código de la función encerrado entre llaves. Las funciones pueden ser declaradas con tipo de retorno void y de ser así la función no deberá de contener un retorno. Un compound statement será un conjunto de declaración de variables locales y de statements.

Las llamadas a funciones consistirán de un identificador que represente el nombre de la función, seguido de un conjunto de parámetros separados por coma y encerrados en un grupo de paréntesis.

Para las expresiones aritméticas es necesario tomar en consideración la asociatividad y la precedencia establecida comúnmente por los operadores.

Para la validación del cumplimento del objetivo se preparan algunos casos de prueba que se alinean a las reglas del lenguaje y otros de modo contrario para validad que los casos erróneos son detectados exitosamente.

# 3. Diseño

### 3.1 Escáner léxico

Para la implementación de autómata se usará la convención básica en donde:

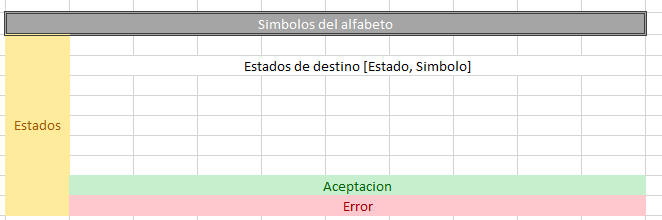
Los círculos representaran los estados en los cuales podrá estar el autómata y un doble circulo los estados de aprobación

*Figura 2: Representación de estado no final y final*

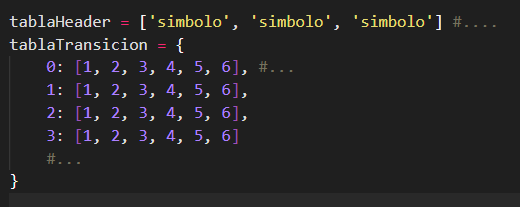
Las flechas indicarán las transiciones posibles entre estados, estas líneas estarán acotadas por el símbolo mediante el cual se realiza la transición y serán unidireccionales.

*símbolo*

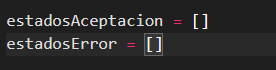
*Figura 3: Representación de estado no final y final*

La tabla de transiciones estará basada en el autómata generado y será añadida al código del sistema media una estructura de datos para guardar cada símbolo y transición entre los estados. La tabla tendrá la siguiente estructura:

*Figura 4: Representación de tabla de transiciones*

La representación en código de la tabla será mediante el uso de diccionarios de la siguiente manera:

*Figura 5: Representación en código de tabla de transiciones*

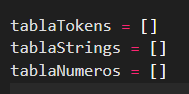
Para definir los estados de aceptación y de error se usará una lista conteniendo dichos estados

*Figura 6: Representación en código de tabla de estados*

Las palabras reservadas según los requerimientos se agregaran al código en forma de un arreglo de la siguiente manera:  *Figura 7: Representación en código de tabla para palabras reservadas*

Para el cumplir con el requerimiento de definir las tablas de símbolos y su definición se crearan 3 tablas de símbolos, una para los identificadores, otra para número y finalmente para los tokens.

representadas como arreglos en el código y tendrán una estructura similar a:

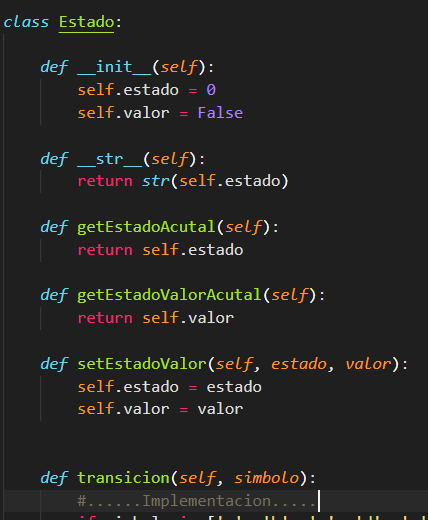


*Figura 8: Representación en código de tabla tokens*

Para el caso de la tabla de tokens, almacenara diccionarios conteniendo el tipo de token y asociado un valor en la forma *{“tipo”: valor}*

La estructura principal del sistema será un loop que leerá la cadena de caracteres de entrada y una clase “Estado” para representar los estados del autómata y contara con las funciones para set y get para cada uno de sus atributos, lo cuales serán su valor actual y su propiedad de aceptación.

Incluirá la función de transición la cual se encargará de obtener el nuevo estado recibiendo como parámetro el siguiente símbolo y cotejándolo con la tabla de transición.

La representación en código tendrá la siguiente estructura:

*Figura 8: Estructura base del estado en código*

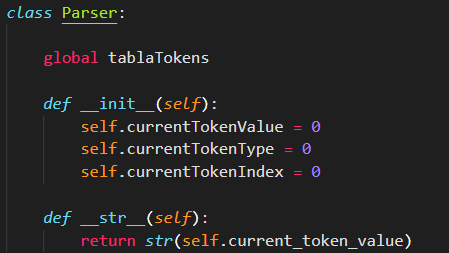
### 3.2 Analizador sintáctico

Para definir la gramática del lenguaje se empleará la notación de una gramática libre de contexto en la que del lado derecho se encuentran los símbolos no terminales, luego una flecha que separara los 2 lados de la producción y finalmente del lado derecho las derivaciones de las reglas que puede ser un conjunto de símbolos terminales (no derivan nada) y no terminales. Sera necesario recordar las notaciones para entender cómo se estructurará la gramática.

*prop* -> **if** (*expr*) *prop* **else***prop*

Al definir todas las reglas gramáticas se procederá a caluclar los first, follow y first+ sets respectivamente.

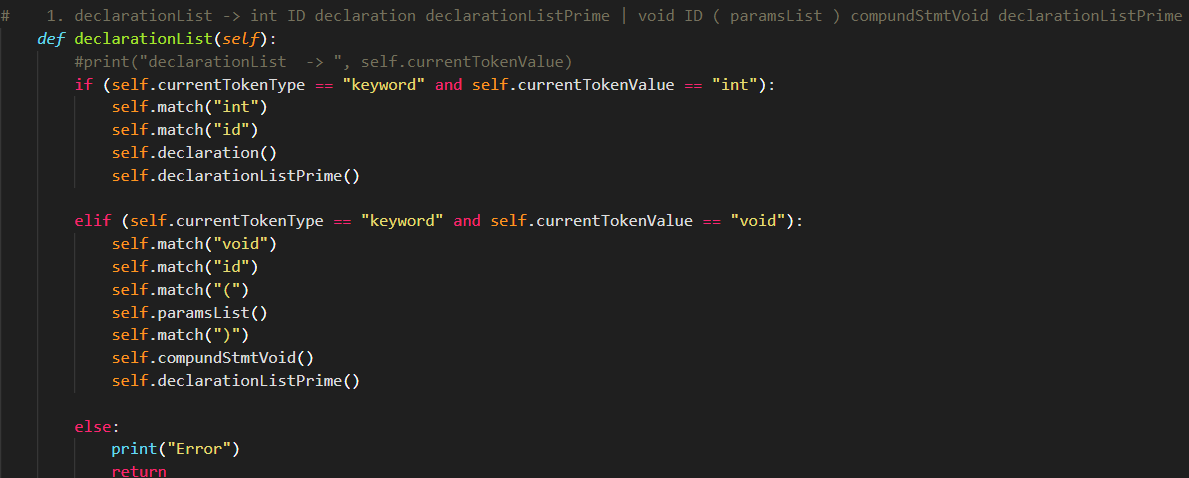
La estructura básica del código respecto al analizador será una clase que alojará el token actual, su tipo y su valor como en la *figura 9.*



*Figura 9: Estructura base del parser en código*

Luego se usará el modelo de recursive procedures y se creará una función para cada regla o producción gramatical. Es decir, para la siguiente regla se genera la funcion descrita en la *figura 10*.

*declarationList* -> **int ID** *declaration declarationListPrime* | **void ID** **(** *paramsList* **)** *compundStmtVoid declarationListPrime*

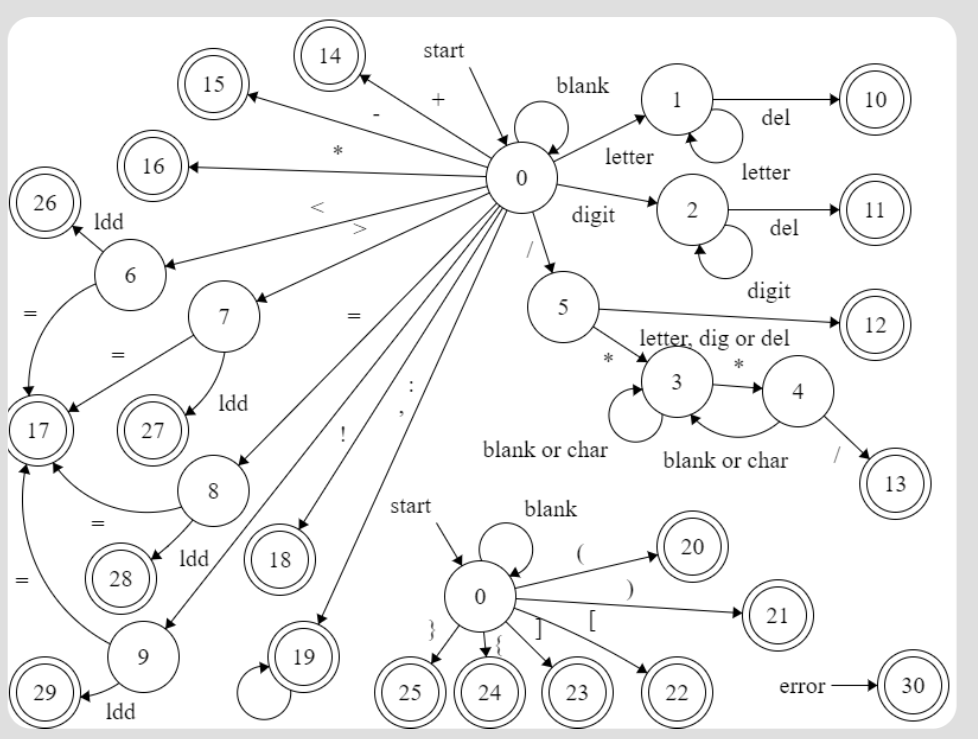


*Figura 10: Estructura básica para un recursive procedure en codigo*

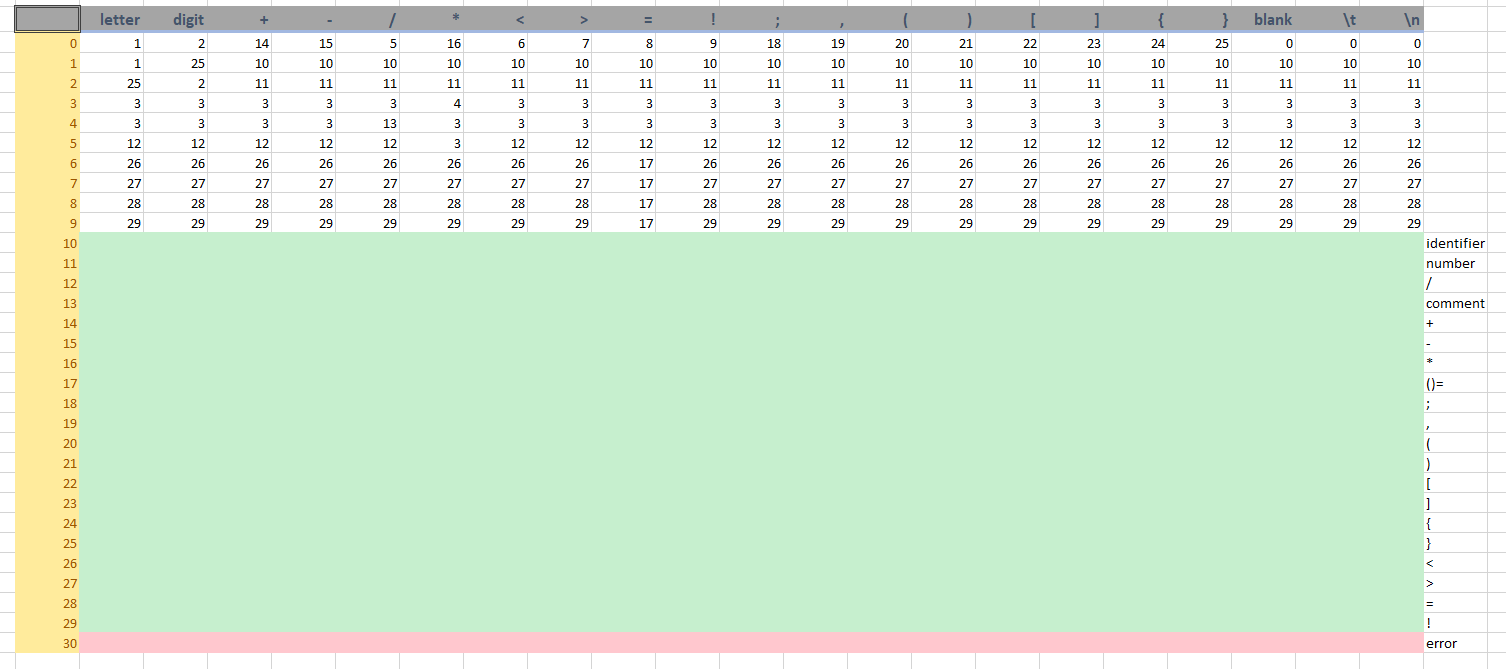
# 4. Implementación

### 4.1 Escáner léxico

#### 4.1.1 Autómata del lenguaje



#### 4.1.2 Tabla de transiciones



### 4.2 Analizador sintáctico

#### 4.2.1 gramática libre de contexto

1. declarationList -> int ID declaration declarationListPrime | void ID ( paramsList ) compundStmtVoid declarationListPrime

2. declarationListPrime -> int ID declaration declarationListPrime | void ID ( paramsList ) compundStmtVoid declarationListPrime | epsilon

3. declaration -> varOrArrayDeclaration ; | ( paramsList ) compountStmtNoVoidReturn

4. varOrArrayDeclaration -> [arithmeticExpression] | epsilon

5. paramsList -> int ID paramVarOrArray paramsListPrime | void

6. paramsListPrime -> , int ID param\_array paramsListPrime | epsilon

7. paramVarOrArray -> [ ] | epsilon

8. compundStmtVoid -> { localDeclarations stmtListVoid }

9. compountStmtNoVoidReturn -> { localDeclarations stmtListNoVoid returnStmtNoVoid }

10. compundStmtNoVoidNoReturn -> { localDeclarations stmtListNoVoid }

11. localDeclarations -> int ID varOrArrayDeclaration; localDeclarations | epsilon

13. stmtListVoid -> statement\_void stmtListVoid | epsilon

14. stmtListNoVoid -> stmtNoVoid stmtListNoVoid | epsilon

15. statement\_void -> assignmentCallStmt | compundStmtVoid | selectionStmtVoid | loop\_stmt\_void | ioStmt

16. stmtNoVoid -> assignmentCallStmt | compundStmtNoVoidNoReturn | selectionStmtNoVoid | loop\_stmt\_int | returnStmtNoVoid | ioStmt

17. ioStmt -> inputStmt | outputStmt

18. assignmentCallStmt -> ID assignmentCallStmtPrime ;

19. assignmentCallStmtPrime -> varOrArrayDeclaration = expression | ( args )

20. selectionStmtVoid -> if ( expression ) statement\_void selectionStmtNoVoidElsePart

21. selectionStmtNoVoidElsePart -> else statement\_void | epsilon

22. selectionStmtNoVoid -> if ( expression ) stmtNoVoid selectionStmtNoVoidElsePart

23. selectionStmtNoVoidElsePart -> else stmtNoVoid | epsilon

24. loopStmtVoid -> while ( expression ) statement\_void

25. loopStmtNoVoid -> while ( expression ) stmtNoVoid

26. returnStmtNoVoid -> return returnStmtNoVoidPrime;

27. returnStmtNoVoidPrime -> expression | epsilon

28. inputStmt -> input ID varOrArrayDeclaration ;

29. outputStmt -> output expression ;

30. expression -> arithmeticExpression expressionPrime

31. expressionPrime -> relop arithmeticExpression | epsilon

32. relop -> <= | < | > | >= | == | !=

33. arithmeticExpression -> term expPrime

34. expPrime -> + term exp' | - term exp' | epsilon

35. term -> factor termPrime

36. termPrime -> \* factor termPrime | / factor termPrime | epsilon

37. factor -> ( arithmeticExpression ) | ID varOrCall | NUM

38. varOrCall -> varOrArrayDeclaration | ( args )

39. args -> arithmeticExpression argsList | epsilon

40. argsList -> , arithmeticExpression argsList | épsilon

#### 4.2.1 Calculo de first, follow y first+

Imagen que contiene captura de pantalla, pista, tren, luz

Descripción generada automáticamenteImagen que contiene captura de pantalla, computadora, laptop, estacionamiento

Descripción generada automáticamenteImagen que contiene captura de pantalla

Descripción generada automáticamente

El código fuente del programa puede ser encontrado a través del siguiente link, tanto para el escáner como par el analizador:

<https://github.com/CesarFloresPraim/CompilerC--.git>

# Verificación y validación

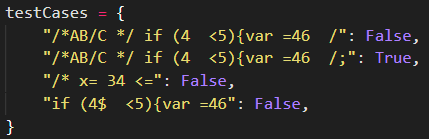
Para verificar que el sistema cumple realmente el propósito con el que fue diseñado se debe de presentar 3 salidas para el caso del escáner léxico: tabla de tokens, de cadena de caracteres y de números.

Estas salidas serán tablas que contendrán cada una respectivamente los tokens que el analizador léxico vaya encontrando conforme recorre la tabla de transiciones

Se proveen 4 cadenas de caracteres para su validación con el analizador léxico y así aprobar que lo que cumpla con las especificaciones.

Para el caso en donde un error encontrado por el analizador léxico sea encontrado se deberá de dar información al usuario acerca del error.

Los casos de uso son agregados al analizador dentro del código en la variable global llamada testCases.



Las salidas para los casos de pruebas se las tablas generadas por el analizador y en caso de error un mensaje.

# References

1. Alfred V. Aho, Ravi Sethi y Jeffrey D. Ullman. *Compiladores: principios, técnicas y herramientas.*
2. John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman *. Teoría de autómatas, lenguajes y computación.*