UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROYECTO 2

ANALIZADOR LÉXICO - SINTÁCTICO

NOMBRES:

Piña Rossette Marco Antonio

Reyes Mendoza Miriam Guadalupe

ASIGNATURA:
Compiladores

GRUPO: 3

SEMESTRE: 2023-1

FACULTAD DE INGENIERÍA

ANALIZADOR Léxico - Sintáctico

DESCRIPCIÓN DE PROBLEMA

Primeros debemos conocer las funciones y bases en el funcionamiento del analizador sintáctico, ya que haremos su implementación sobre el analizador léxico realizado con anterioridad. Entonces, sabemos que el análisis sintáctico, es la segunda fase de un compilador, un analizador léxico puede identificar tokens con la ayuda de expresiones regulares y reglas de patrones. Pero un analizador léxico no puede verificar la sintaxis de una oración dada debido a las limitaciones de las expresiones regulares. Las expresiones regulares no pueden verificar tokens de equilibrio, como paréntesis. Por lo tanto, esta fase utiliza gramática libre de contexto, que es reconocida por autómatas push – down.

Una gramática libre de contexto tiene cuatro componentes:

- Un conjunto de no terminales (V). Los no terminales son variables sintácticas que denotan conjuntos de cadenas. Los no terminales definen conjuntos de cadenas que ayudan a definir el lenguaje generado por la gramática.

- Un conjunto de tokens, conocidos como símbolos terminales (Σ). Los terminales son los símbolos básicos a partir de los cuales se forman las cadenas.

- Un conjunto de producciones (P). Las producciones de una gramática especifican la manera en que los terminales y los no terminales pueden combinarse para formar cadenas. Cada producción consta de un no-terminal denominado lado izquierdo de la producción, una flecha y una secuencia de tokens y/o on-terminales , denominado lado derecho de la producción.

- Uno de los no terminales se designa como el símbolo de inicio (S); desde donde comienza la producción.

Las cadenas se derivan del símbolo de inicio reemplazando repetidamente un no terminal el lado derecho de una producción, para ese no terminal. Un analizador sitactico, a fin de cuentas, toma la entrada de un analizador léxico en forma de flujos de tokens y analiza el código fuente contra las reglas de producción para detectar cualquier error en el código.

Además, se espera que los analizadores analicen todo el código incluso si existen algunos errores en el programa. De esta forma, se busca construir, en un mismo programa, los analizadores léxico y sintáctico descendente recursivo que revisen programas escritos en el lenguaje definido por la gramática presentadas más adelante.

SENTENCIA DECLARATIVA

$$D \rightarrow < Tipo > K; \qquad Q \rightarrow = NC$$
$$< Tipo > \rightarrow b \qquad Q \rightarrow , K$$
$$< Tipo > \rightarrow g \qquad N \rightarrow n$$
$$< Tipo > \rightarrow \# \qquad N \rightarrow r$$
$$< Tipo > \rightarrow y \qquad N \rightarrow s$$
$$< Tipo > \rightarrow x \qquad C \rightarrow \xi$$
$$K \rightarrow iQ \qquad C \rightarrow , K$$
$$Q \rightarrow \xi \qquad Q \rightarrow = NC$$

SENTENCIA DE ASIGNACIÓN

$$A \rightarrow i = A';$$
$$A' \rightarrow s$$
$$A' \rightarrow E$$

EXPRESIÓN ARITMÉTICA

$$E \rightarrow T\ E' \qquad T' \rightarrow \%FT'$$
$$E' \rightarrow + T\ E' \qquad T' \rightarrow \hat{}FT'$$
$$E' \rightarrow - T\ E' \qquad T' \rightarrow \xi$$
$$E' \rightarrow \xi \qquad F \rightarrow (\ E\ )$$
$$T \rightarrow F\ T' \qquad F \rightarrow i$$
$$T' \rightarrow * F\ T' \qquad F \rightarrow n$$
$$T' \rightarrow /FT' \qquad F \rightarrow r$$
$$T' \rightarrow \backslash FT' \qquad F \rightarrow < Llama >$$

EXPRESIÓN RELACIONAL

$$R \rightarrow iR'V \qquad V \rightarrow i$$
$$R \rightarrow nR'V' \qquad V \rightarrow n$$
$$R \rightarrow rR'V'' \qquad V \rightarrow r$$
$$R \rightarrow sR'V''' \qquad V \rightarrow s$$
$$R' \rightarrow > \qquad V' \rightarrow n$$
$$R' \rightarrow < \qquad V' \rightarrow i$$
$$R' \rightarrow l \qquad V'' \rightarrow r$$
$$R' \rightarrow e \qquad V'' \rightarrow i$$
$$R' \rightarrow d \qquad V''' \rightarrow s$$
$$R' \rightarrow u \qquad V''' \rightarrow i$$

PROPOSICIONES

$$P \rightarrow A$$
$$P \rightarrow I \qquad P \rightarrow < Llama >$$
$$P \rightarrow H \qquad P \rightarrow < Devuelve >$$
$$P \rightarrow W \qquad P \rightarrow c$$
$$P \rightarrow J$$

LISTA DE 0 O MÁS PROPOSICIONES

$$< listaP > \rightarrow \xi$$
$$< listaP > \rightarrow P < listaP >$$

SENTENCIA LOOP

$$W \rightarrow w(R)m\{< listaP >\}$$

SENTENCIA EVALUATE

$$I \rightarrow f(R) < listaP > I':$$
$$I' \rightarrow t < listaP >$$
$$I' \rightarrow \xi$$

SENTENCIA REPEAT

$$J \rightarrow j(YXZ\{< listaP >\}$$
$$Y \rightarrow i = E;$$
$$Y \rightarrow ;$$
$$X \rightarrow R;$$
$$X \rightarrow ;$$
$$Z \rightarrow i = E)$$
$$Z \rightarrow)$$

SENTENCIA SELECT

$$H \rightarrow h(i)\{C'O'\}$$
$$C' \rightarrow an:< listaP > UC'$$
$$C' \rightarrow \xi$$
$$O' \rightarrow o:< listaP >$$
$$O' \rightarrow \xi$$
$$U \rightarrow q$$
$$U \rightarrow \xi$$

SENTENCIA THROW

$$< Devuelve > \rightarrow z(< valor >);$$
$$< valor > \rightarrow V$$
$$< valor > \rightarrow \xi$$

LLAMADA A UNA FUNCIÓN

$$< Llama > \rightarrow [i(< arg >)]$$
$$< arg > \rightarrow \xi$$
$$< arg > \rightarrow V < otroArg >$$
$$< otroArg > \rightarrow ,V < otroArg >$$
$$< otroArg > \rightarrow \xi$$

FUNCIONES

$$< Func > \rightarrow < Tipo > i(< Param >)\{< Cuerpo >\}$$
$$< Param > \rightarrow < Tipo > i < otroParam >$$
$$< Param > \rightarrow \xi$$
$$< otroParam > \rightarrow ,< Tipo > i < otroParam >$$
$$< otroParam > \rightarrow \xi$$
$$< Cuerpo > \rightarrow < Decl >< listaP >$$
$$< Decl > \rightarrow \xi$$
$$< Decl > \rightarrow D < Decl >$$

ESTRUCTURA DEL PROGRAMA

$$< Program > \rightarrow < Func >< otraFunc >$$
$$< otraFunc > \rightarrow < Func >< otraFunc >$$
$$< otraFunc > \rightarrow \xi$$

De forma más concreta, se pretende realizar un analizador léxico – sintáctico funcional que haga uso de las gramáticas que definen la sintaxis de nuestro lenguaje, es importante, saber que en el proyecto encontramos algunas limitaciones, pero se pretende realizar un analizador lo más completo posible para entender su funcionamiento y aplicación.

- La entrada es un archivo con el programa fuente a analizar que deberá estar escrito en el lenguaje definido por la gramática definida anteriormente en este documento. Este archivo de entrada se indicará desde la línea de comandos.

- El programa realizará tanto el análisis léxico como el sintáctico. El analizador léxico deberá generar además de los tokens, la cadena de átomos que será la entrada del analizador sintáctico. Los átomos se pueden ir generando a la par que los tokens, pero irlos almacenando en una sola cadena.

- Los átomos están definidos en este documento por cada componente léxico y correspondena los elementos terminales de la gramática.

- La tabla de clases de componentes léxicos con sus correspondientes átomos es:

CLASE	DESCRIPCIÓN	ÁTOMO
0	Palabras reservadas (ver tabla).	(ver tabla)
1	Identificadores. Iniciar con $ y le sigue al menos una letra minúscula o mayúscula. Ejemplos: $ejemplo, $Varible, $OtraVariable, $XYZ	i
2	Constantes numéricas enteras. En base 10 (secuencia de dígitos del 0-9 sin 0's a la izquierda, excepto para el número 0), en base 8 (inicien con O u o y le sigan dígitos del 0 al 7).	n
3	Constantes numéricas reales. Siempre deben llevar parte decimal y es opcional la parte entera. Ejemplos: 73.0, .0, 10.2 No aceptados: . , 12 , 4.	r
4	Constantes cadenas. Encerrado entre comillas (“) cualquier secuencia de más de un carácter que no contenga “ ni ‘. Para cadenas de un solo carácter, encerrarlo entre apóstrofos (‘). La cadena de unas comillas debe ser encerrada entre apóstrofos: ‘“’ . La cadena de un apóstrofo debe ser encerrada por comillas: “‘” . No se aceptan cadenas vacías. Ejemplos NO válidos: “ejemplo no “valido” , “” , “”” , “hola ‘mundo”	s
5	Símbolos especiales [ ] ( ) { } , : ;	mismo símbolo
6	Operadores aritméticos + - * / % \ ^	mismo símbolo
7	Operadores relacionales (ver tabla)	(ver tabla)
8	Operador de asignación =	=

- El valor en los tokens y los átomos se indican en las siguientes tablas.

VALOR	PALABRA RESERVADA	EQUIVALE EN C	ÁTOMO
0	alternative	case	a
1	big	long	b
2	evaluate	if	f
3	instead	else	t
4	large	double	g
5	loop	while	w
6	make	do	m

VALOR	EQUIVALE EN C	ÁTOMO
0	<	<
1	>	>
2	<=	l
3	>=	u
4	= =	e
5	! =	d

7	number	int	#
8	other	default	o
9	real	float	x
10	repeat	for	j
11	select	switch	h
12	small	short	p
13	step	continue	c
14	stop	break	q
15	symbol	char	y
16	throw	return	z

- El analizador sintáctico deberá mostrar todos los errores sintácticos que encuentre, indicandoqué se esperaba.

- Como resultados, el analizador léxico-sintáctico deberá mostrar el contenido de la tablade símbolos, las tablas de literales, los tokens y la cadena de átomos. Finalmente deberáindicar si está sintácticamente correcto el programa fuente.

- Los errores que vaya encontrando el analizador léxico, los podrá ir mostrando en pantalla oescribirlos en un archivo, así como él o los errores sintácticos. Es conveniente que cuando encuentre un error sintáctico se indique en qué átomo de la cadena se encontró (con ubicación).

- El programa deberá estar comentado, con una descripción breve de lo que hace (puede ser elobjetivo indicado en este documento), el nombre de quienes elaboraron el programa y fechade elaboración.

Por último debemos tener en consideración que los analizadores sintácticos reciben sus entradas, en forma de tokens, de los analizadores léxicos. Los analizadores léxicos son responsables de la validez de un token proporcionado por el analizador de sintaxis. Los analizadores de sintaxis tienen los siguientes inconvenientes:

- No puede determinar si un token es válido.

- No puede determinar si un token se declara antes de que se use.

- No puede determinar si un token se inicializa antes de que se use.

- No puede determinar si una operación realizada en un tipo de token es válida o no.

CONJUNTOS DE SELECCIÓN

Para hallar los respectivos conjuntos de selección debemos saber que:

Sea $\boldsymbol{i}: \boldsymbol{A} \longrightarrow \boldsymbol{\alpha}$

$$\boldsymbol{C.S.(i)} = \begin{cases} \boldsymbol{First(i)} & \boldsymbol{si\ i\ no\ es\ anulable} \\ \boldsymbol{Follow(A)} & \boldsymbol{si\ i\ es\ anulable} \end{cases}$$

GRAMÁTICA	CONJUNTO DE SELECCIÓN
1. $< Program > \rightarrow < Func >< otraFunc >$	$C.S.(1) = \{bg\#yx\}$
2. $< otraFunc > \rightarrow < Func >< otraFunc >$	$C.S.(2) = \{bg\#yx\}$
3. $< otraFunc > \rightarrow\ \xi$	$C.S.(3) = \{\dashv\}$
4. $< Func > \rightarrow < Tipo > i(< Param >)\{< Cuerpo\ >\}$	$C.S.(4) = \{bg\#yx\}$
5. $< Param > \rightarrow < Tipo > i < otroParam >$	$C.S.(5) = \{bg\#yx\}$
6. $< Param > \rightarrow\ \xi$	$C.S.(6) = \{)\}$
7. $< otroParam > \rightarrow, < Tipo > i < otroParam >$	$C.S.(7) = \{,\}$
8. $< otroParam > \rightarrow\ \xi$	$C.S.(8) = \{)\}$
9. $< Cuerpo > \rightarrow < Decl >< listaP >$	$C.S.(9) = \{i\}bg\#yxcwfjhz[\}$
10. $< Decl > \rightarrow\ \xi$	$C.S.(10) = \{i\}cwfjhz[\}$
11. $< Decl > \rightarrow\ D < Decl >$	$C.S.(11) = \{bg\#yx\}$
12. $D\ \rightarrow < Tipo > K;$	$C.S.(12) = \{bg\#yx\}$
13. $< Tipo > \rightarrow\ b$	$C.S.(13) = \{b\}$
14. $< Tipo > \rightarrow\ g$	$C.S.(14) = \{g\}$
15. $< Tipo > \rightarrow\ \#$	$C.S.(15) = \{\#\}$
16. $< Tipo > \rightarrow\ y$	$C.S.(16) = \{y\}$
17. $< Tipo > \rightarrow\ x$	$C.S.(17) = \{x\}$
18. $K\ \rightarrow\ iQ$	$C.S.(18) = \{i\}$
19. $Q\ \rightarrow\ \xi$	$C.S.(19) = \{;\}$
20. $Q\ \rightarrow = NC$	$C.S.(20) = \{=\}$
21. $Q\ \rightarrow, K$	$C.S.(21) = \{,\}$
22. $N\ \rightarrow\ n$	$C.S.(22) = \{n\}$
23. $N\ \rightarrow\ r$	$C.S.(23) = \{r\}$

24. $N \rightarrow s$	$C.S.(24) = \{s\}$
25. $C \rightarrow \xi$	$C.S.(25) = \{;\}$
26. $C \rightarrow ,K$	$C.S.(26) = \{,\}$
27. $A \rightarrow i = A';$	$C.S.(27) = \{i\}$
28. $A' \rightarrow s$	$C.S.(28) = \{s\}$
29. $A' \rightarrow E$	$C.S.(29) = \{(inr[\}$
30. $E \rightarrow T\ E'$	$C.S.(30) = \{(inr[\}$
31. $E' \rightarrow +\ T\ E'$	$C.S.(31) = \{+\}$
32. $E' \rightarrow -\ T\ E'$	$C.S.(32) = \{-\}$
33. $E' \rightarrow \xi$	$C.S.(33) = \{;)\}$
34. $T \rightarrow F\ T'$	$C.S.(34) = \{(inr[\}$
35. $T' \rightarrow *\ F\ T'$	$C.S.(35) = \{*\}$
36. $T' \rightarrow /FT'$	$C.S.(36) = \{/\}$
37. $T' \rightarrow \backslash FT'$	$C.S.(37) = \{\backslash\}$
38. $T' \rightarrow \%FT'$	$C.S.(38) = \{\%\}$
39. $T' \rightarrow \hat{}FT'$	$C.S.(39) = \{\hat{}\}$
40. $T' \rightarrow \xi$	$C.S.(40) = \{;) + -\}$
41. $F \rightarrow (E)$	$C.S.(41) = \{(\}$
42. $F \rightarrow i$	$C.S.(42) = \{i\}$
43. $F \rightarrow n$	$C.S.(43) = \{n\}$
44. $F \rightarrow r$	$C.S.(44) = \{r\}$
45. $F \rightarrow < Llama >$	$C.S.(45) = \{[\}$
46. $R \rightarrow iR'V$	$C.S.(46) = \{i\}$
47. $R \rightarrow nR'V'$	$C.S.(47) = \{n\}$
48. $R \rightarrow rR'V''$	$C.S.(48) = \{r\}$
49. $R \rightarrow sR'V''''$	$C.S.(49) = \{s\}$
50. $R' \rightarrow >$	$C.S.(50) = \{>\}$
51. $R' \rightarrow <$	$C.S.(51) = \{<\}$
52. $R' \rightarrow l$	$C.S.(52) = \{l\}$
53. $R' \rightarrow e$	$C.S.(53) = \{e\}$

54. $R' \rightarrow d$	$C.S.(54) = \{d\}$
55. $R' \rightarrow u$	$C.S.(55) = \{u\}$
56. $V \rightarrow i$	$C.S.(56) = \{i\}$
57. $V \rightarrow n$	$C.S.(57) = \{n\}$
58. $V \rightarrow r$	$C.S.(58) = \{r\}$
59. $V \rightarrow s$	$C.S.(59) = \{s\}$
60. $V' \rightarrow n$	$C.S.(60) = \{n\}$
61. $V' \rightarrow i$	$C.S.(61) = \{i\}$
62. $V'' \rightarrow r$	$C.S.(62) = \{r\}$
63. $V'' \rightarrow i$	$C.S.(63) = \{i\}$
64. $V''' \rightarrow s$	$C.S.(64) = \{s\}$
65. $V''' \rightarrow i$	$C.S.(65) = \{i\}$
66. $P \rightarrow A$	$C.S.(66) = \{i\}$
67. $P \rightarrow I$	$C.S.(67) = \{f\}$
68. $P \rightarrow H$	$C.S.(68) = \{h\}$
69. $P \rightarrow W$	$C.S.(69) = \{w\}$
70. $P \rightarrow J$	$C.S.(70) = \{j\}$
71. $P \rightarrow < Llama >$	$C.S.(71) = \{[\}$
72. $P \rightarrow < Devuelve >$	$C.S.(72) = \{z\}$
73. $P \rightarrow c;$	$C.S.(73) = \{c\}$
74. $< listaP > \rightarrow \xi$	$C.S.(74) = \{\}: taoq\}$
75. $< listaP > \rightarrow P < listaP >$	$C.S.(75) = \{icwfjhz[\}$
76. $W \rightarrow w(R)m\{< listaP >\}$	$C.S.(76) = \{w\}$
77. $I \rightarrow f(R) < listaP > I'$:	$C.S.(77) = \{f\}$
78. $I' \rightarrow t < listaP >$	$C.S.(78) = \{t\}$
79. $I' \rightarrow \xi$	$C.S.(79) = \{:\}$
80. $J \rightarrow j(YXZ\{< listaP >\}$	$C.S.(80) = \{j\}$
81. $Y \rightarrow i = E;$	$C.S.(81) = \{i\}$
82. $Y \rightarrow ;$	$C.S.(82) = \{;\}$
83. $X \rightarrow R;$	$C.S.(83) = \{inrs\}$

84. $X \rightarrow ;$	$C.S.(84) = \{;\}$
85. $Z \rightarrow i = E)$	$C.S.(85) = \{i\}$
86. $Z \rightarrow)$	$C.S.(86) = \{)\}$
87. $H \rightarrow h(i)\{C'O'\}$	$C.S.(87) = \{h\}$
88. $C' \rightarrow an: < listaP > UC'$	$C.S.(88) = \{a\}$
89. $C' \rightarrow \xi$	$C.S.(89) = \{\}o\}$
90. $O' \rightarrow o: < listaP >$	$C.S.(90) = \{o\}$
91. $O' \rightarrow \xi$	$C.S.(91) = \{\}\}$
92. $U \rightarrow q$	$C.S.(92) = \{q\}$
93. $U \rightarrow \xi$	$C.S.(93) = \{\}ao\}$
94. $< Devuelve > \rightarrow z(< valor >);$	$C.S.(94) = \{z\}$
95. $< valor > \rightarrow V$	$C.S.(95) = \{inrs\}$
96. $< valor > \rightarrow \xi$	$C.S.(96) = \{)\}$
97. $< Llama > \rightarrow [i(< arg >)]$	$C.S.(97) = \{[\}$
98. $< arg > \rightarrow \xi$	$C.S.(98) = \{)\}$
99. $< arg > \rightarrow V < otroArg >$	$C.S.(99) = \{inrs\}$
100. $< otroArg > \rightarrow ,V < otroArg >$	$C.S.(100) = \{,\}$
101. $< otroArg > \rightarrow \xi$	$C.S.(101) = \{)\}$

Podemos ver que los conjuntos de selección para las diferentes gramáticas realizadas son disjuntos, de este modo, podemos decir que cumple con ser una gramática LL(1), entonces, debemos conocer que:

- La primera L significa escanear la entrada de izquierda a derecha.
- La segunda L representa la producción de una derivación más a la izquierda.
- El 1 representa el uso de un símbolo de entrada de anticipación en cada paso para tomar una decisión de acción de análisis.

De este modo, se puede demostrar que las gramáticas LL(1) no son ambiguas y no recursivas a la izquierda. Entonces, que una gramática cuya tabla de análisis sintáctico no tiene entradas definidas de forma múltiple es LL(1), lo que significa escanear la entrada de izquierda a derecha produciendo una derivación más a la izquierda y usando 1 símbolo de entrada de anticipación en cada paso para tomar decisiones de acción de análisis.

SOLUCIÓN Y FASES DE DESARROLLO

Tomando en cuenta que el análisis de sintaxis es una segunda fase del proceso de diseño del compilador en el que se comprueba la cadena de entrada dada para confirmar las reglas y la estructura de la gramática formal. Analiza la estructura sintáctica y verifica si la entrada dada está en la sintaxis correcta del lenguaje de programación o no.

El análisis de sintaxis en el proceso de diseño del compilador viene después de la fase de análisis léxico. También se conoce como árbol de análisis o árbol de sintaxis. El Parse Tree se desarrolla con la ayuda de la gramática predefinida del idioma. El analizador de sintaxis también comprueba si un programa determinado cumple las reglas implícitas en una gramática libre de contexto. Si satisface, el analizador crea el árbol de análisis de ese programa fuente. De lo contrario, mostrará mensajes de error.

El desarrollo de este proyecto, pretende entonces:

- Comprobar si el código es gramaticalmente válido.
- El analizador sintáctico te ayuda a aplicar reglas al código.
- Le ayuda a asegurarse de que cada llave de apertura tenga un saldo de cierre correspondiente.
- Cada declaración tiene un tipo y ese tipo debe existir.
- Un análisis también verifica que la cadena de entrada esté bien formada y, de no ser así, la rechaza.

ANÁLISIS

Las siguientes son tareas importantes y las tomamos en cuenta para realizar el analizador en el diseño del compilador:

- Le ayuda a detectar todo tipo de errores de sintaxis
- Encuentre la posición en la que se ha producido el error
- Descripción clara y precisa del error.
- Recuperación de un error para continuar y encontrar más errores en el código.
- No debería afectar la compilación de programas "correctos".
- El análisis debe rechazar textos inválidos informando errores de sintaxis.

Una gramática es un conjunto de reglas estructurales que describen una lengua. Las gramáticas asignan estructura a cualquier oración. Este término también se refiere al estudio de estas reglas, y este archivo incluye la morfología, la fonología y la sintaxis. Es capaz de describir muchas, de la sintaxis de los lenguajes de programación.

Una gramática libre de contexto es una gramática recursiva por la izquierda que tiene al menos una producción del tipo. Las reglas en una gramática libre de contexto son principalmente recursivas. Un analizador de sintaxis verifica que un programa específico cumpla o no con todas las reglas de la gramática libre de contexto. Posteriormente, la derivación gramatical es una secuencia de reglas gramaticales que transforma el símbolo de inicio en la cadena. Una derivación prueba que la cadena pertenece al lenguaje de la gramática.

El análisis léxico y sintáctico se puede simplificar a una máquina que toma algún código de programa y luego devuelve errores de sintaxis y estructuras de datos. Es justo aquí que la principal diferencia entre el análisis léxico y el análisis de sintáctico, el léxico lee el código fuente un carácter a la vez y lo convierte en tokens, mientras que el sintáctico toma esos tokens y produce un análisis sintáctico como salida. La fase léxica es la primera fase en el proceso de compilación. Toma el código fuente como entrada. Además, escanea el programa fuente y convierte un carácter a la vez en tokens significativos. El resultado del análisis léxico pasa a la fase de análisis sintáctico.

La segunda fase del proceso de compilación es el análisis de sintaxis. Toma los tokens producidos por el analizador léxico como entrada. En esta fase, las organizaciones de tokens se comparan con la gramática del código fuente. Además, el analizador realiza análisis de sintaxis, comprobando si la expresión hecha por los tokens es sintácticamente correcta. Además de los dos pasos anteriores, el proceso de compilación también incluye fases como el análisis semántico, la generación de código intermedio, la generación de código y la optimización de código.

PLANIFICACIÓN

Con base en el analisis, fue como decidimos dividir la creación del analizador sintáctico en diferentes etapas de diseño. Las cuales se mencionan a continuación, junto con la persona encargada de su implementación.

- Correciones del analizador léxico → Miriam
- Reconocimiento de los conjuntos de seleccion → Miriam
- Codificación de funciones de la gramática → Miriam
- Generación de atómos → Antonio
- Almacenamiento de los atómos → Antonio
- Muestra de los errores sintácticos → Antonio
- Ver si es sintacticamente correcto → Antonio

DISEÑO E IMPLEMENTACIÓN

Flex es una herramienta para generar analaizadores. Un analizador lo podriamos resumir como un programa que reconoce patrones léxicos en el texto. El programa flex lee los archivos de entrada dados, o su entrada estándar si no se dan nombres de archivos, para una descripción de un escáner a generar. La descripción está en forma de pares de expresiones regulares y código C, llamados reglas.

Flex genera como salida un fichero fuente C, $lex.yy.c$ por defecto, que define una rutina $yylex()$. Este fichero puede compilarse y enlazarse con la biblioteca de tiempo de ejecución de flex para producir un ejecutable. Cuando el ejecutable se ejecuta, analiza su entrada en busca de ocurrencias de las expresiones regulares. Cuando encuentra una, ejecuta el código C correspondiente.

Se decidió iniciar con la definiución de los atomos Se decidió iniciar con la definición de los átomos establecidos en la tabla de requerimientos, posteriormente una vez definidos, se hizo la declaración de la cadena de átomos para que esta se pudiera visualizar como se solicita. Lo más importante fueron las funciones, primeramente, se definieron de acuerdo con los nombres establecidos en la tabla de gramáticas.

Justamente, este fue el proceso más laborioso puesto que implico definir las funciones para la gramática, pues implicó el análisis y la obtención de los conjuntos de selección, bueno lo cual fue un proceso tardado ya que debemos obtener un conjunto para las 101 producciones que teníamos, una vez hecho esto solamente nos quedó crear las funciones de acuerdo con su producción.

Posteriormente mandamos a imprimir la cadena de átomos, los cuales se han generado a partir del reconocimiento de las diferentes funciones, recorriendo cada una de las posiciones dentro de la cadena que estamos generando y por último, reconociendo los errores generados al momento de llamar y hacer uso de una función que no cumple con la sintaxis esperada.

INDICACIONES PARA CORRER EL PROGRAMA

Primero se debe realizar la trducción de flex a C, por lo que este recibe nuestro archivo tipo lex.

$$\$\ flex\ lexico.l$$

Una vez traducido, se generará un archivo $\boldsymbol{lex.yy.c}$, este se compila con el compilador de C usando el siguiente comando:

$$\$\ gcc\ -g\ lex.yy.c\ SLL/SLL.c\ -lfl\ -o\ lexico$$

En el comando anterior se está tomando en cuenta a la biblioteca externa de la lista ligada simple, ya que sin ella no funciona el programa. Esto nos generará un archivo ejecutable, el cuál deberá recibir la dirección del archivo que queremos analizar:

```
$ ./lexico test.txt
```

CONCLUSIONES

PIÑA ROSSETTE MARCO ANTONIO

Logramos, mediante la aplicación del conocimiento de teoría de las gramáticas LL(1), construir el analizador sintáctico recursivo descendente en el mismo programa del analizador léxico y estos juntos logran revisar los programas que estén escritos en el lenguaje definido por la gramática realizada en clase del lenguaje de programación que definimos.

Igualmente, al aplicar los conocimientos teóricos comprendí de mejor manera el funcionamiento de un analizador recursivo descendente y cómo es que hace el reconocimiento de una cadena de átomos.

REYES MENDOZA MIRIAM GUADALUPE

La implementación del analizador sintáctico, fue un proceso bastante complejo, ya que implico una gran cantidad de código para la definición de las diferente funciones, más aun porque teníamos demasiada producciones definidas, sin embargo esto hace que nuestro compilador sea más completo pero implica mucho tiempo al ver qué tipo de producción esta generando cada una y aplicar la regla correspondiente, además, de ir reconociendo los átomos dentro de cada una aunque esto no fue tan complicado ya teniendo las gramáticas definidas.

Otro de los puntos más complejos fue la definición de errores, sin embargo, se encontró la solución más optima al indicar el tipo de error. Sabemos que se necesita del analizador sintáctico para determinar si la serie de tokens dados son apropiados en un lenguaje, es decir, si la cadena tiene o no la forma correcta. Sin embargo, no todas las cadenas sintácticamente válidas son significativas, se debe aplicar un análisis semántico adicional para esto. Para el análisis sintáctico, las gramáticas independientes del contexto y las técnicas de análisis sintácticas asociadas son lo suficientemente poderosas como para ser utilizadas.