UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Proyecto 2

"Analizador Léxico-Sintáctico"

Compiladores

Grupo 03

Integrantes:

Alcantar Correa Vianey

Sánchez Rosas Alexis Alejandro

Profesor: M.C. LAURA SANDOVAL MONTAÑO

Fecha de entrega:23/Mayo/2023

Índice

Objetivo

Construir, en un mismo programa, los analizadores Léxico y Sintáctico Descendente Recursivo que revisen programas escritos en el lenguaje definido por la gramática del Anexo A del documento de especificaciones del proyecto.

Descripción del problema

El analizador léxico que se diseñó e implementó en el proyecto 1 nos permite identificar que simbolos pertenecen a nuestro lenguaje y cuáles no. Además gracias a las expresiones regulares y la lectura caracter por caracter es capaz de identificar variables, palabras reservadas, operadores, etc. El siguiente componente en el trabajo que nos compete es el analizador sintáctico que se encarga de reconocer la gramática a la que pertenece nuestro lenguaje; con ayuda del analizador sintáctico además de reconocer los componentes léxicos también seremos capaces de reconocer la estructura sintáctica de nuestros códigos fuente.

El primer desafio de para la correcta implementación de nuestro análisis Léxico - Semántico es realizar las correcciones pertinentes a nuestro analizador semántico para que no tenga ningún tipo de error al momento de reconocer cualquier cadena que se ingrese. Para el correcto funcionamiento del analizador semántico es necesario que se reconozcan todos los tokens de manera correcta y se tenga un manejo de errores para las situaciones esperadas en cuanto a un error léxico.

El problema consiste en analizar e identificar los conjuntos de selección de la gramática proporcionada para nuestro lenguaje la cual consta de 79 producciones.

Gramática del lenguaje

1:	<Program> →<Decl>[<Func><otraFunc>]
2:	<otraFunc> →<Func><otraFunc>
3:	<otraFunc> → ξ
4:	<Func> → <Tipo>a(<Param>){<Cuerpo >}
5:	<Param> → <Tipo>a<otroParam>
6:	<Param> → ξ
7:	<otroParam>→,<Tipo>a<otroParam>
8:	<otroParam> → ξ
9:	<Cuerpo> → <Decl>S'
10:	<Decl> → ξ
11:	<Decl> → D<Decl>
12:	D → <Tipo>aVD'
13:	<Tipo>→ e
14:	<Tipo>→ r
15:	<Tipo>→ b
16:	<Tipo>→ le
17:	D'→ ,aVD'
18:	D'→ ;
19:	V → =C
20:	C→ n
21:	C→ c
22:	C→ x
23:	A → a=A';
24:	A' → c
25:	A' → E
26:	E → T E'
27:	E' → + T E'
28:	E' → - T E'
29:	E' → ξ
30:	T → F T'
31:	T' → *F T'
32:	T' → /FT'
33:	T' → %FT'
34:	T' → ξ
35:	F → (E)
36:	F → a
37:	F → n
38:	F → x
39:	F → <Llama>
40:	R → V'R'V'
41:	R' → >
42:	R' → <
43:	R' → q
44:	R' → g
45:	R' → d
46:	R' → p
47:	V' → a
48:	V' → C
49:	S → A
50:	S → B
51:	S → H
52:	S → M
53:	S → P
54:	S → <Llama>
55:	S → <Devuelve>
56:	S → s;
57:	S' → SS'
58:	S' → ξ
59:	M → m(R'')(S')
60:	R'' → R
61:	R'' → ξ
62:	B → i(R)S'B'
63:	B' → oS'f
64:	B' → f
65:	P → t(P')C'
66:	P' → a=E;R;a=E
67:	C' → ;
68:	C' → S;
69:	C' → {S'}
70:	H→ h{S'}H'
71:	H' → m(R'');
72:	<Devuelve> → v(<valor>);
73:	<valor> → V'
74:	<valor> → ξ
75:	<Llama> → [a(<arg>)]
76:	<arg> → ξ
77:	<arg> → V'<otroArg>
78:	<otroArg> → ,V'<otroArg>
79:	<otroArg> → ξ

Cuando se hayan conseguido los conjuntos de selección se debe proceder a modificar el programa del analizador léxico para que, una vez concluido el análisis léxico, se pase el control del programa al analizador sintáctico con una función parser(). Además se debe modificar el programa para crear una cadena de átomos conforme se va creando la tabla de tokens, está cadena de átomos será la única entrada que recibirá el analizador sintáctico.

Cuando llegamos al momento de programar la función Parser() se debe trabajar con los conjuntos de selección para identificar cada escenario y realizar un correcto reconocimiento de la cadena de átomos utilizando la gramática. Se espera que finalmente se tenga un archivo que indique los errores sintácticos que se encuentren e indique una correcta inspección sintáctica en caso de no encontrar errores de esta naturaleza.

Bitácora de actividades

Tarea	Realizado por:
La entrada es un archivo con el programa fuente a analizar que deberá estar escrito en el lenguaje definido por la gramática del Anexo A del documento de requerimientos. Este archivo de entrada se indicará desde la línea de comandos.	Ambos integrantes
El programa realizará tanto el análisis léxico como el sintáctico (en el Anexo B encontrarás cómo incluirlo en el analizador léxico). 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.	Ambos integrantes
Los átomos están definidos en este documento por cada componente léxico y corresponden a los elementos terminales de la gramática	Vianey Alcantar
La tabla de clases de componentes léxicos con sus correspondientes átomos. El valor en los tokens y los átomos se indican en sus respectivas tablas.	Vianey Alcantar
El analizador sintáctico deberá mostrar todos los errores sintácticos que encuentre, indicando qué se esperaba.	Alexis Sánchez
Como resultados, el analizador léxico-sintáctico deberá mostrar el contenido de la tabla de símbolos, las tablas de literales, los tokens y la cadena de átomos. Finalmente deberá indicar si está sintácticamente correcto el programa fuente.	Ambos integrantes

Los errores que vaya encontrando el analizador léxico, los podrá ir mostrando en pantalla o escribirlos 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).	Alexis Sánchez
El programa deberá estar comentado, con una descripción breve de lo que hace (puede ser el objetivo indicado en este documento), el nombre de quienes elaboraron el programa y fecha de elaboración.	Vianey Alcantar

Propuesta de solución y partes del desarrollo del sistema

Generación de cadena de átomos

Para generar las cadenas de átomos conforme se generan las cadenas de tokens es necesario la declaración de dos arreglos que contengan los átomos que solicita el documento de especificaciones. Estos arreglos serán de tipo caracter.

```
//Tabla de átomos para operadores relacionales y palabras reservadas
char    tablaAtomosOpRel [] = {'q','d','g','>','<','p'};
char    tablaAtomosPalRes [] = {'e','f','h','m','l','t','r','v','i','b','o'};
```

Se busca crear la cadena de átomos conforme se crea la tabla de tokens para facilitar el trabajo, entonces en un nuevo apuntador a archivo llamado ‘atomosFile’ se escribe únicamente el carácter correspondiente a cada clase según se requiera, en la imagen se ilustra la creación de átomos para identificadores (átomo carácter ‘a’), símbolos especiales (átomo es el mismo símbolo) y operador relacional que toma el valor ‘value’ que entra a la función ‘writeToken’ para saber que átomo debe colocar en la cadena de átomos, cubriendo de está forma los 3 casos posibles para la creación de átomo.

```
case 0:         //Identificadores
    fprintf(outFile,"identificador: %s con Token: ( %d,%d )\n",yytext,class,value );
    fprintf(atomosFile,"%c",'a');
    break;
case 1:         //Simbolo especial
    fprintf(outFile,"Simbolo especial: '%s' con Token: ( %d,%d )\n",yytext,class,value );
    fprintf(atomosFile,"%s",yytext);
    break;
case 2:         //Operador relacional
    fprintf(outFile,"Operador relacional: %s con Token: ( %d,%d )\n",yytext,class,value );
    fprintf(atomosFile,"%c",tablaAtomosOpRel[value]);
```

Conjuntos de selección

Comenzamos las actividades relacionadas con el analizador sintáctico con la obtención de los conjuntos de selección de cada producción.

Producción	Conjunto de selección
1: <Program> → <Decl>[<Func><otraFunc>]	C.S(1) = {[e r b l}
2: <otraFunc> →<Func><otraFunc>	C.S(2) = {e r b l}
3: <otraFunc> → ξ	C.S(3) = {First(3) U Follow(<otraFunc>) ={} U {]} = {]}
4: <Func> → <Tipo>a(<Param>){<Cuerpo >}	C.S(4) = {e r b l}
5: <Param> → <Tipo>a<otroParam>	C.S(5) = {e r b l}
6: <Param>→ ξ	C.S(6) = {First(6) U {Follow(<Param>)} ={} U {) } ={)}
7: <otroParam>→,<Tipo>a<otroParam>	C.S(7) = {,}
8: <otroParam> → ξ	C.S(8) = {First(8)}U{Follow(<otroParam>)}={}U{)}={) }
9: <Cuerpo> → <Decl>S'	C.S(9) ={First(9)}U{Follow(<Cuerpo>)}={erblaihmt[vs}U{}}={erblaihmt[vs}}
10: <Decl>→ ξ	{First(10)}U{Follow(<Cuerpo>)}={erblaihmt[vs}U{}}={[aihmtvs}}
11: <Decl> → D<Decl>	C.S(11) = {e r b l}
12: D → <Tipo>aVD'	C.S(12) = {e r b l}
13: <Tipo>→ e	C.S(13) = {e}
14: <Tipo>→ r	C.S(14) = {r}
15: <Tipo>→ b	C.S(15) = {b}
16: <Tipo>→ le	C.S(16) = {l}
17: D'→ ,aVD'	C.S(17) = {,}
18: D'→ ;	C.S(18) = {;}
19: V → =C	C.S(19) = {=}
20: C→ n	C.S(20) = {n}
21: C→ c	C.S(21) = {c}
22: C→ x	C.S(22) = {x}
23: A → a=A';	C.S(23) = {a}
24: A' → c	C.S(24) = {c}
25: A' → E	C.S(25) = {a n x ([}
26: E → T E'	C.S(26) = {a n x ([}
27: E' → + T E'	C.S(27) = {+}
28: E' → - T E'	C.S(28) = {-}
29: E'→ ξ	C.S(29) ={First(29)}U{Follow(E')}={}U{;)}={;)}
30: T →F T'	C.S(30) = {a n x ([}
31: T' → *F T'	C.S(31) = {*}
32: T' → /FT'	C.S(32) = {/}
33: T' → %FT'	C.S(33) = {%}
34: T' → ξ	C.S(34) = {First(34)}UFollow(T')={}U{+-;)}={+-;)}
35: F → (E)	C.S(35) = {(}
36: F → a	C.S(36) = {a}
37: F → n	C.S(37) = {n}
38: F → x	C.S(38) = {x}
39: F → <Llama>	C.S(39) = {[}
40: R → V'R'V'	C.S(40) = {a n c x}

41: R' → >	C.S(41) = {>}
42: R'→ <	C.S(42) = {<}
43: R' → q	C.S(43) = {q}
44: R' → g	C.S(44) = {g}
45: R' → d	C.S(45) = {d}
46: R' → p	C.S(46) = {p}
47: V' → a	C.S(47) = {a}
48: V' → C	C.S(48) = {n c x}
49: S →A	C.S(49) = {a}
50: S → B	C.S(50) = {i}
51: S → H	C.S(51) = {h}
52: S → M	C.S(52) = {m}
53: S → P	C.S(53) = {t}
54: S →<Llama>	C.S(54) = {[}
55: S →<Devuelve>	C.S(55) = {v}
56: S → s;	C.S(56) = {s}
57: S' → SS'	C.S(57) = {a i h m t [v s}
58: S' → ξ	C.S(58) = {First(58)}U{Follow(S')}={}U{}of}={}of}
59: M → m(R")}{S'}	C.S(59) = {m}
60: R" → R	C.S(60) = {a n c x}
61: R" → ξ	C.S(61) = {First(61)}UFollow(R")={}U{}}={}}
62: B → i(R)S'B'	C.S(62) = {i}
63: B' → oS'f	C.S(63) = {o}
64: B' → f	C.S(64) = {f}
65: P → t(P')C'	C.S(65) = {t}
66: P' → a=E;R;a=E	C.S(66) = {a}
67: C' → ;	C.S(67) = {;}
68: C'→ S;	C.S(68) = {a i h m t [v s}
69: C'→ {S'}	C.S(69) = {{}
70: H→ h{S'}H'	C.S(70) = {h}
71: H' → m(R");	C.S(71) = {m}
72: <Devuelve> → v(<valor>);	C.S(72) = {v}
73: <valor> → V'	C.S(73) = {a n c x}
74: <valor> → ξ	C.S(74) ={First(74)}U{Follow(<valor>)}={}U{}}={}}
75: <Llama> → [a(<arg>)]	C.S(75) = {[}
76: <arg> → ξ	C.S(76) ={First(76)}U{Follow(arg)}={}U{}}={}}
77: <arg> → V'<otroArg>	C.S(77) = {a n c x}
78: <otroArg> → ,V'<otroArg>	C.S(78) = {,}
79: <otroArg> → ξ	C.S(79) = {First(79)}U{Follow(<otroArg>)}={}U{}}={}}

Llamada a función parser()

Antes de llamar la función es necesario escribir un símbolo de fin de cadena para la cadena de átomos, para ello después de llamar a la función 'yylex()' se imprime directamente el carácter '#' al final del archivo

al que apunta ‘atomosFIle’, con esto se consigue tener el indicador de fin de cadena exactamente donde nos interesa para el análisis.

```
// Ejecuta el analizador léxico (Flex) para analizar l
yylex();
// Imprime la tabla de símbolos generada por el analiz
printSymbolTable();
printStringTable();
printRealTable();
//Agrega fin de cadena antes de análisis sintáctico
fprintf(atomosFile,"%c",'#');
fclose(atomosFile);
//Imprime cadena de atomos
imprimirCadena(atomosFile);
```

Inmediatamente después de cerrar los archivos se manda a llamar a la función ‘parser()’.

```
void parser(){
    //Se abren los archivos de entrada y salida
    atomosFile = fopen("atomos.txt", "r");
    sintaxis = fopen("sintaxis.txt", "w");
    //Contador indica que átomo se esta leyendo
    counter++;
    //Caracter se obtiene directamente desde el archivo con fgetc()
    c = fgetc(atomosFile);
    //Se llama al simbolo inicial de la gramática para comenzar el análisis
    Program();
    //Cuando termina el análisis se tiene el final de cadena con o sin errores sintácticos
    if (c == '#'){
        if(flag)
            fprintf(sintaxis, "El programa es sintacticamente correcto!\n");
        else
            fprintf(sintaxis, "El programa contiene errores sintácticos\n");
    }
    else{
        fprintf(sintaxis, "Se esperaba el fin de cadena \n");
        return;
    }
    //Se cierran los archivos
    fclose(atomosFile);
    fclose(sintaxis);
}
```

Dentro de la función se abren los archivos ‘atomos.txt’ para escritura y ‘sintaxis.txt’ para lectura. Además para el funcionamiento de la función y de todas las funciones relacionadas a las producciones de la gramática se crearon 2 variables globales. La primera de nombre ‘c’ que se encargará de almacenar el carácter actual que se está procesando o comparando y la variable ‘counter’ que servirá de contador para finalmente indicar, en caso de existir, el lugar en donde se encuentra el átomo que ocasionó un error sintáctico. Antes de llamar a la producción inicial de la gramática se incrementa el valor de ‘counter’ y se obtiene el primer carácter de la cadena directamente desde el archivo utilizando la función ‘fgetc()’.

Entonces se manda a llamar la producción inicial, se espera que una vez concluido el análisis la variable ‘c’ tenga el valor del carácter de fin de cadena ‘#’. Si esto es así entonces se leyó correctamente todo el archivo de entrada en caso contrario ocurrió algún error inesperado. Si además existe algún error

sintáctico y llegó el carácter de fin de cadena se muestra que el análisis concluyó de manera satisfactoria pero en algún momento se encontró con algún error sintáctico.

Estructura de programación para las producciones de la gramática

Se elaboran funciones para cumplir con el autómata de pila que nos dirá si una cadena es aceptada o rechazada en el análisis sintáctico.

-

Caso	Tipo producción	Código
1	i : A → b	c= getchar( ) return
2	i : A → bα	c= getchar( ) procesar(α) return
3	i : A → ξ	return
4	i : A → Bα	procesar(Bα) return

Donde, procesar(α) y procesar(Bα) consiste en:

1) Por cada no-terminal en α y para B, llamar a la función que le corresponde.
2) Por cada terminal:

```
if (c == 'x')
    c= getchar( );
else
    rechaza();
```

donde x ∈ Σ_p

Observamos que la producción inicial cae en el caso 4 de nuestra tabla de funciones por lo que primero se compara nuestra variable ‘c’ con los elementos del conjunto de selección de la primera producción en caso de existir una coincidencia podemos comenzar siguiendo las instrucciones de la tabla. Como el primer elemento de la frase es un símbolo no terminal entonces se procesa alfa y posteriormente se realiza una nueva comparación por cada símbolo terminal. Cuando se encuentra un símbolo terminal se actualiza el valor de la variable ‘c’, se incrementa el valor de la variable ‘counter’ y se continúa con el análisis del siguiente símbolo (carácter).

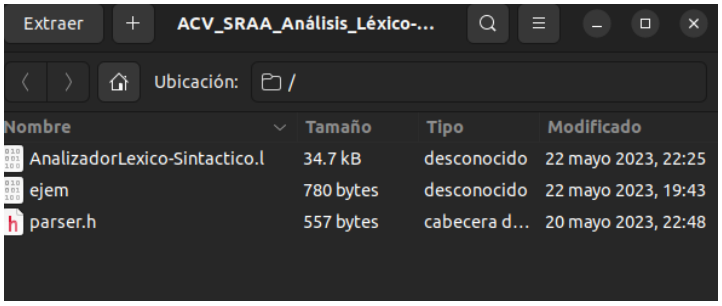
Si se encuentra un elemento no terminal se manda a llamar la función del no terminal correspondiente, el manejo de errores está conformado por una función error que mostrará la información actual de las variables ‘c’ y ‘counter’ y un mensaje que se guarda en el archivo ‘sintaxis.txt’ que almacenará toda la información relevante del análisis.

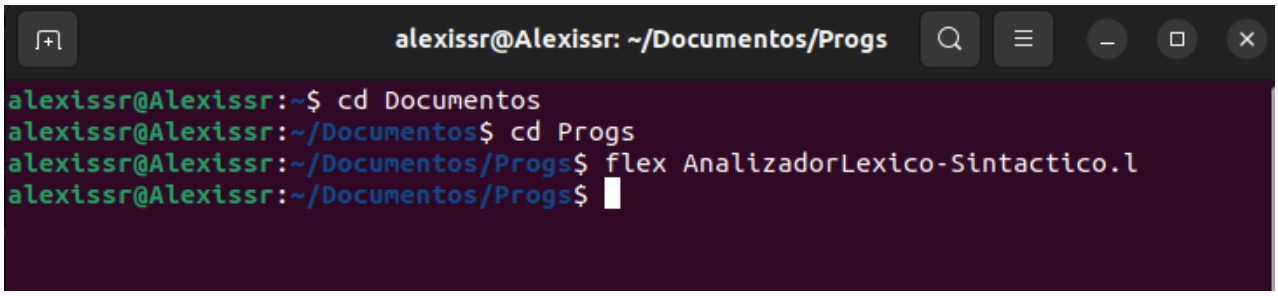
```
//Comienza la programación de las producciones.
void Program(){
    if (c == '[' || c == 'e' || c == 'r' || c == 'b' || c == 'l'){
        Decl();
        if(c == '['){
            c = fgetc(atomosFile);
            counter++;
        }else{
            error(counter);
            fprintf(sintaxis,"Se esperaba el caracter '[' \n");
        }
        Func();
        otraFunc();
        if ( c == ']'){
            c = fgetc(atomosFile);
            counter++;
        }else{
            error(counter);
            fprintf(sintaxis,"Se esperaba el caracter ']' \n");
        }
        return;
    }
    else{
        error(counter);
        fprintf(sintaxis,"Se esperaba el caracter '[' o 'e' o 'r' o 'b' o 'l'\n");
        return;
    }
}
```

Este proceso se repite para cada producción y se debe tener cuidado de manejar correctamente los errores, finalmente el programa se probará en el siguiente apartado con un archivo de prueba para demostrar que reconoce un archivo de entrada sintácticamente bien construido.

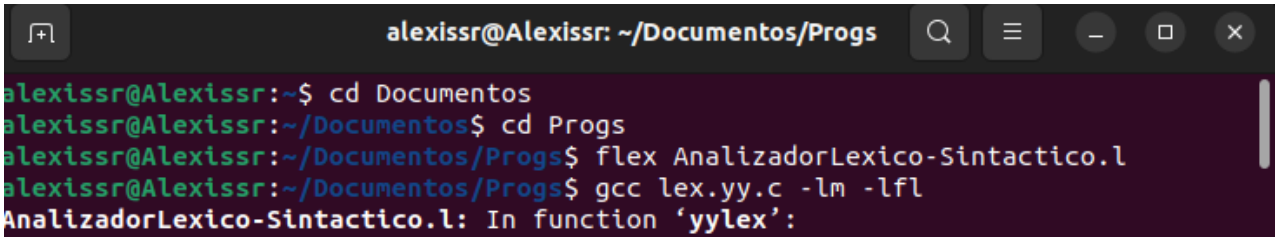
Indicaciones de como correr el programa

Con la documentación se hace entrega de 3 archivos, el código fuente con terminación .l, un archivo de texto de nombre ejem y un header file que almacena los prototipos de función para cada producción de la gramática por lo que es indispensable que estos 3 archivos se encuentren en una misma carpeta.

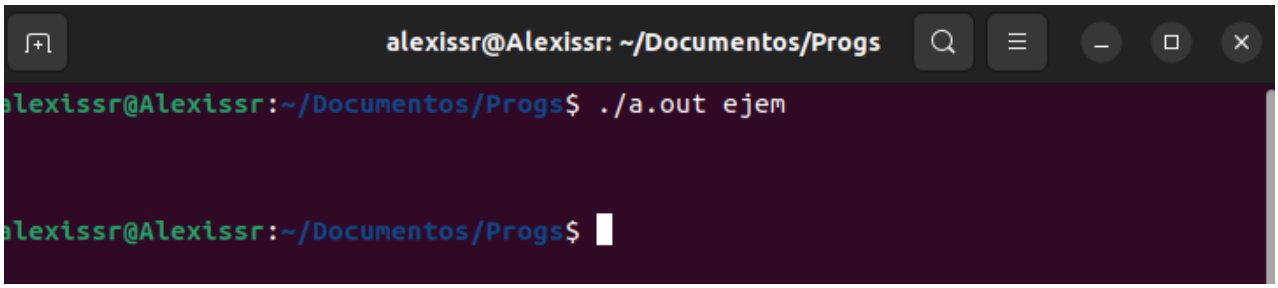




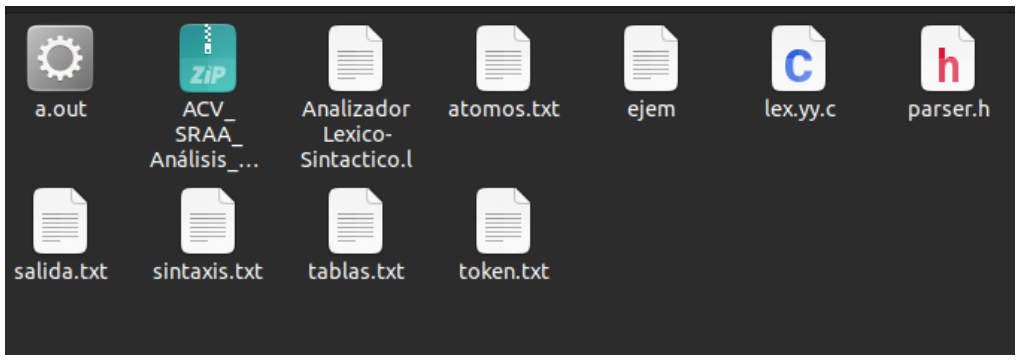
Desde la consola nos ubicamos en la carpeta que contiene los archivos y utilizamos el comando flex para generar el archivo con terminación .c que necesitamos compilar utilizando gcc.



Se utiliza el compilador GCC del lenguaje C con las cláusulas -lm y -lfl que son necesarias para que la operación se realice exitosamente.



Finalmente ejecutamos con el comando de la imagen y se generan archivos .txt que contienen toda la información resultante del análisis léxico-sintáctico.



Conclusiones

Alcantar Correa Vianey

Construir en un mismo programa los analizadores Léxico y Sintáctico Descendente Recursivo para revisar programas con el lenguaje que se definió en clase nos permitió extender la comprensión en el tema del análisis descendente . Aunque enfrentamos dificultades iniciales, logramos superarlas y llevar a cabo la implementación de ambos analizadores de manera efectiva. Durante el desarrollo del proyecto, nos enfrentamos a desafíos que pusieron a prueba nuestra capacidad de análisis y resolución de problemas. Una de las dificultades más destacadas fue la creación de los conjuntos de selección necesarios para el diseño del programa. Sin embargo, con un trabajo en equipo sólido y una colaboración efectiva, pudimos superar estos obstáculos y avanzar en el proceso de implementación. La construcción del analizador léxico nos permitió identificar y clasificar correctamente los tokens presentes en el código fuente, mientras que el analizador sintáctico descendente recursivo verificó la estructura y sintaxis del programa. Esto nos permitió detectar errores y anomalías en el código, brindando una valiosa herramienta de verificación y análisis.

Sánchez Rosas Alexis Alejandro

La realización del trabajo fue de gran utilidad para poner en práctica los conceptos y ejercicios vistos en clase y aplicarlos a un problema de ingeniería como lo es la construcción de un analizador sintáctico descendente recursivo en el cual se basó nuestro programa. Los conjuntos de selección fueron en conjunto con el manejo de errores las tareas más desafiantes a las que nos enfrentamos para cumplir con los rubros del proyecto en su totalidad. Gracias al trabajo en equipo y el repaso de los conceptos teóricos la construcción del analizador se llevó a cabo dentro del tiempo esperado y cumpliendo con la rúbrica presentada por el documento de especificaciones, el proyecto fue entonces de gran utilidad para repasar todos los conceptos que tienen que ver con el análisis sintáctico y además su integración con el análisis léxico nos dió como resultado un programa que puede realizar el reconocimiento de errores de estás dos naturalezas y que son de gran importancia para la construcción de un compilador.