Facultad de Ingeniería UNAM

Universidad Nacional Autónoma de México

Analizador Léxico

M.I.A Laura Sandoval Montaño

Compiladores

Morales Ortega Carlos
Vélez Grande Cinthya

Fecha de entrega: 03/10/2023
Semestre 2024-1
Grupo 03

Universidad Nacional Autónoma de México

Objetivo

Elaborar un analizador léxico en lex/flex a partir de la definición de clases de componentes léxicos acordados en el grupo.

Descripción del Problema

Se deberá realizar un analizador léxico en lex/flex que reconozca las siguientes clases:

CLASE	DESCRIPCIÓN
0	Operadores aritméticos
1	Operadores lógicos
2	Operadores relacionales
3	Constantes numéricas enteras
4	Palabras reservadas
5	Identificadores
6	Símbolos especiales
7	Operadores de asignación
8	Constantes de cadenas
9	Operadores sobre cadenas

El analizador tendrá como entrada un archivo con el programa fuente, dicho programa será indicado desde la línea de comandos al momento de ejecutar el analizador léxico.

Cada componente léxico está delimitado por espacios, saltos de líneas, tabuladores o bien otro componente léxico.

Los tokens generados tendrán la siguiente estructura con dos campos:

Campo 1	Campo 2
CLASE	VALOR

El **valor** para el token de *identificador* es la posición dentro de la tabla de símbolos. El **valor** para los tokens de *palabras reservadas, operadores lógicos, operadores sobre cadenas,* y *operadores de asignación;* será la posición dentro de su correspondiente tabla (catálogo). Para las *constantes de cadenas* el **valor** será la posición dentro de la tabla de literales. Para *operadores aritméticos* y *símbolos especiales,* su **valor** será el mismo carácter o su correspondiente ASCII. Finalmente, para las *constantes enteras* como **valor** será el mismo valor de la constante entera. Si es negativo se pondrá el signo – si es con el signo + se omitirá.

Al detectar un error léxico, deberá seguir reconociendo a partir del siguiente símbolo.

El analizador deberá crear la Tabla de Símbolos para almacenar los identificadores. Esta tabla manejará los campos: posición, nombre del identificador y tipo (por defecto tendrá valor de -1).

La tabla de literales cuenta con dos campos: posición y cadena.

Al finalizar, el analizador léxico deberá mostrar tabla de símbolos, literales y tokens
Los errores que vaya encontrando puede mostrarlos en pantalla o escribirlos en un archivo. Deberá recuperarse de los errores encontrados para continuar con el reconocimiento de todos los componentes léxicos del archivo de entrada.

Características de componentes léxicos

Clase 0 – Operadores aritméticos

Los operadores aritméticos válidos dentro del lenguaje que se desarrolla son los siguientes:

- + (suma)
- - (resta)
- / (división)
- * (multiplicación)
- % (módulo)

Debido a que un operador aritmético puede ser cualquiera de estos símbolos de manera individual, la expresión regular para representar esta clase de componentes léxicos en lex/flex es la siguiente:

(\+|-|\/|*|%)

Clase 1 – Operadores Lógicos

Los operadores lógicos válidos dentro del lenguaje que se desarrolla son los siguientes:

- && (and)
- || (or)
- ! (not)

Debido a que un operador lógico puede ser cualquiera de estos símbolos de manera individual, la expresión regular para representar esta clase de componentes léxicos en lex/flex es la siguiente:

(&&|\|\||!)

Clase 2 – Operadores relacionales

Los operadores relacionales que se han considerado dentro del lenguaje que se desarrollará son los que se enlistan a continuación:

- == (igual)
- != (distinto)
- > (mayor que)
- < (menor que)
- >= (mayor o igual que)
- <= (menor o igual que)

La expresión regular que describe esta clase de componente léxico en lex/flex es:

(>|<|=|!)=|>|<

Clase 3 – Constantes numéricas enteras

Las constantes numéricas válidas dentro del lenguaje que se desarrolla tienen las siguientes características:

- Base 10
- Si esta con signo (+ o -) encerrarlo entre "paréntesis". Ejemplo: (-389), (+100)

La expresión regular para representar esta clase de componentes léxicos en lex/flex es la siguiente:

({dig})*|(\((\+|-)({dig})*\))

Donde {dig} = [0-9]

Clase 4 – Palabras reservadas

Las palabras reservadas que se acordaron para la implementación del lenguaje serán las siguientes:

- assinado (void)
- caso (case)
- enquanto (while)
- fazer (do)
- flutuador (float)
- inteiro (int)
- para (for)
- quebrar (break)
- Retorno (return)
- se (if)
- trocar (switch)

Para la expresión regular que representa esta clase de componente léxico, se colocan estas palabras separadas mediante barras verticales para indicar que cualquiera de estas, que sea reconocida de manera individual, es parte de las palabras reservadas:

inteiro|enquanto|quebrar|retorno|para|trocar|assinado|se|fazer|caso|flutuador

Clase 5 – Identificadores

Los identificadores válidos dentro del lenguaje que se desarrolla tienen las siguientes características:

- Inicio con _ (guion bajo)
- Le sigue una letra minúscula o mayúscula
- Después pueden tener letras minúsculas, mayúsculas, dígitos y _ (guion bajo)

La expresión regular para representar esta clase de componentes léxicos en lex/flex es la siguiente:

({mayus}|{minus})(|{mayus}|{minus}|{dig})*

Donde:
{dig} = [0-9]
{minus}=[a-z]

{mayus}=[A-Z]

Clase 6 – Símbolos especiales

Dentro del lenguaje en desarrollo se consideran símbolos especiales los siguientes:

- ( ) (paréntesis)
- { } (llaves)
- ; (punto y coma)
- , (coma)
- [ ] (corchetes)
- : (dos puntos)
- # (numeral)

La expresión regular considera la presencia de cualquiera de estos de forma individual para su reconocimiento, por lo que la expresión en lex/flex será:

(\(|\)|;|\{|\}|\[|\]|:|#|,)

Clase 7 – Operadores de asignación

Los operadores de asignación válidos dentro del lenguaje que se desarrolla son los siguientes:

- =
- +=
- -=
- *=
- /=
- %=

Debido a que un operador de asignación puede ser cualquiera de estos símbolos de manera individual, la expresión regular para representar esta clase de componentes léxicos en lex/flex es la siguiente:

(\+|-|*|\/|%)?=

Clase 8 – Constantes de cadenas

Las constantes cadenas serán aquellas secuencias de caracteres que estén encerradas entre comillas, estas deberán incluir los saltos de línea. Dado que las comillas son el carácter que delimita el inicio y el final de una cadena, este será el único carácter que no podrá estar contenido dentro de la cadena en sí, por esta razón la expresión regular que describa a las cadenas deberá considerar las comillas como símbolo inicial y final; y entre estas dos se considerarán caracteres que no sean comillas; esto se especifica de la siguiente forma:

\"([^"])*\"

Clase 9 – Operadores sobre cadenas

Los operadores sobre cadenas válidos dentro del lenguaje que se desarrolla son los siguientes:

- &
- like

Debido a que un operador sobre cadena puede ser cualquiera de estos símbolos de manera individual, la expresión regular para representar esta clase de componentes léxicos en lex/flex es la siguiente:

(&|like)

Propuesta de solución

Se plantea un programa en lex/flex que con base en las expresiones regulares que satisfacen cada clase de componente léxico, al momento de reconocer un componente léxico, en las reglas de operación realice ciertas funciones de búsqueda e inserción. De búsqueda, si se trata de un componente léxico que se encuentra en un catálogo o tabla, de tal forma que devuelva su posición; de inserción, si se trata de un componente léxico que deba agregarse a la tabla de literales o símbolos.

Por otro lado, cuando un componente léxico es reconocido, luego de ser agregado o encontrado su valor en su respectiva tabla, catálogo o de acuerdo con lo planteado en el problema; se agrega su clase y valor a una lista que contendrá todos los tokens de acuerdo con el orden en que fueron encontrados.

Lo anterior se logró a partir del uso de **estructuras, listas ligadas, arreglos de estructuras, funciones inserción y búsqueda, constructores, funciones auxiliares, variables globales, definición de expresiones regulares y reglas de traducción.**

Análisis

De forma conjunta, ambos integrante del equipo, realizamos el análisis de la problemática a resolver, una vez comprendida, propusimos soluciones para su resolución de forma modular, subdividiendo el problema de forma que pudiésemos realizar avances en el programa de manera independiente hasta cierto punto; para ello, primeramente resolvimos aquellos aspectos básicos del programa que sí debían atenderse conjuntamente, como el diseño de las expresiones regulares y la definición de estructuras básicas y catálogos.
Para ello, cada integrante desarrolló cinco expresiones regulares, las compartimos, intercambiamos ideas y realizamos las correcciones pertinentes para que aceptaran todos los casos.

A la par, Carlos investigó e implementó la lectura de un archivo de entrada sobre el cual se realizaría el análisis; pudiendo comprobar los resultados de las expresiones sobre un programa contenido en un archivo de entrada. Así mismo, en las funciones de activación especificamos que se imprimiera el tipo de componente léxico reconocido.

Una vez que determinamos que las expresiones eran correctas, Cinthya se encargó de la creación e implantación de los catálogos, diseñando tanto las estructuras, como las funciones necesarias para su almacenamiento.

Intercambiamos ideas respecto a la forma de trabajar las tablas de símbolos, de literales y de tokens, concluyendo que lo mejor sería implantarlas mediante listas ligadas; por lo que, ya teniendo como base las expresiones regulares y los catálogos, Cinthya desarrolló las tablas de símbolos y de literales. Al mismo tiempo, Carlos creó las funciones necesarias para transformar las constantes enteras del tipo de dato *char* a *int* y para obtener el valor en ASCII de los símbolos especiales y aritméticos.

Posterior a ello, Carlos se encargó del desarrollo de la tabla de tokens, considerando este paso la unión de todos los elementos desarrollados anteriormente.

Realizamos las pruebas pertinentes para evaluar el correcto funcionamiento del programa y realizar las adecuaciones necesarias dependiendo de los problemas que encontráramos. Ya que el programa pasó las pruebas planteadas, acordamos que la implantación fue adecuada; siendo esta la versión final del programa que presentamos.

Diseño e implementación

Definición catálogos

Los catálogos son tablas estáticas que contendrán información respecto a algunas clases de componentes léxicos que se manejen en el lenguaje. Como se especifica en la descripción del problema, en el programa se contará con cinco catálogos pertenecientes a los componentes de palabras reservadas, operadores relacionales, lógicos, de asignación y operadores sobre cadenas.
Para su implementación se decidió crear arreglos de estructuras, puesto que estos no modificarán su tamaño en tiempo de ejecución. Las estructuras que se crearon para la implementación de los componentes léxicos que contendrán los catálogos son *reservada* y *operador*, que representan las abstracciones de las palabras reservadas y los operadores, respectivamente. Ambas estructuras cuentan con un miembro llamado *valor*, correspondiente a su posición dentro de los catálogos y otro miembro que será su representación, este es *palabra* en el caso de las palabras reservadas y *simbolo* en el caso de los operadores; implementado mediante un arreglo de caracteres.

Los catálogos son arreglos de las estructuras ya descritas, y se manejan de forma global en el programa. Se cuenta con los siguientes:

- struct reservada catalogoPalReservadas[11];
- struct operador catalogoOpRelacionales[6];
- struct operador catalogoOpLogicos[3];
- struct operador catalogoOpCadenas[2];
- struct operador catalogoOpAsignacion[6];

De igual forma se diseñaron las funciones necesarias para su creación, al interior de estas se realiza el almacenamiento en un archivo nombrado “catalogo.txt”, en el cual se pueden consultar los catálogos que se crean. Dichas funciones son las siguientes:

- ***crearCatalogos()***: función que llama a todas las funciones correspondientes para la creación de todos los catálogos. Recibe el apuntador al archivo en el que se imprimirá el contenido de los catálogos.
- ***creaReservadas()***: crea el catálogo de las palabras reservadas. Recibe la estructura en donde almacenará las palabras y el archivo al cual agregarlas.
- ***agregarPalabra()***: agrega una palabra, que se pasa como parámetro, al archivo, igual enviado como parámetro, con un cierto formato de impresión.
- ***creaOpSobreCadenas()***: crea el catálogo de los operadores sobre cadenas. Recibe la estructura en donde almacenará los símbolos y el archivo al cual agregarlos.
- ***creaOpLogicos()***: crea el catálogo de los operadores lógicos.
- ***creaOpRelacionales()***: crea el catálogo de los operadores relacionales.
- ***creaOpAsignacion()***: crea el catálogo de los operadores de asignación.
- ***agregarOperador():*** agrega un operador, que se pasa como parámetro, al archivo, igual enviado como parámetro, con un cierto formato de impresión.

Estas estructuras serán consultadas durante el proceso de análisis léxico, por lo que su adecuada representación es imprescindible.

Definición tabla de símbolos

Como se ha estudiado, la tabla de símbolos es una tabla dinámica en la cual el compilador irá insertando los identificadores que vaya detectando durante el análisis del programa, únicamente en caso de que estos no se encuentren en la tabla, si ya se han registrado no debe colocarlos.

Partiendo de la premisa anterior, para la implementación de una tabla dinámica se consideró pertinente emplear una lista ligada, de forma que cada identificador que el analizador reconozca conformaría un nodo de la misma.

Se crearon las estructuras y funciones necesarias para la adecuada implementación de la tabla de símbolos; las estructuras declaradas para su funcionamiento son: *identificador*, *nodoIdentificador* y *listaIdentificadores.*

La estructura *identificador* es la abstracción de un identificador con las características necesarias para almacenarlo dentro de la tabla de símbolos, los elementos con los que cuenta son: *posicion, nombre* y *tipo*; el primero es una variable de tipo *int*, el segundo es un arreglo de cien elementos de tipo *char* y el tercero una variable de tipo *short.* Cada miembro de la estructura almacenará el valor correspondiente a las características que conformen al identificador que represente.

Por su parte, la estructura *nodoIdentificador* es un nodo perteneciente a la lista ligada, es decir, será un elemento de la tabla de símbolos, cuyo valor será un elemento de tipo *identificador* y contendrá una referencia (apuntador) a otro elemento del mismo tipo, *nodoIdentificador*, de tal forma que este es su nodo siguiente dentro de la lista. Estos elementos se implementan mediante miembros de la estructura que son *valor* y *next,* con los tipos ya mencionados.

La estructura *listaIdentificadores* será la representación de la tabla de símbolos como tal, pues es la lista ligada que estará conformada por nodos de tipo *nodoIdentificador*, que como se ha mencionado, contendrán los identificadores que se detecten. Esta cuenta con un apuntador del tipo

mencionado, pues es la referencia al primer nodo que la compone, y se denomina *head*, así como una variable llamada *tamanio* que indicará la cantidad de nodos que conforman la lista.

Las funciones desarrolladas para la implementación de la lista de símbolos son las que se describen a continuación.

- ***crearIdentificador()***

Devuelve una estructura de tipo *identificador*, asignando a sus miembros la posición y el nombre del identificador indicados como parámetros. El miembro *tipo* se inicializa con -1.

- ***crearListaIdentificadores()***:

Devuelve una estructura de tipo *listaIdentificadores*, con sus miembros *head* y *tamanio* inicializados con *null* y 0, respectivamente; ya que se desconoce cuál será el primer nodo que conforme la lista y al ser creada no contine ningún nodo aún. La función se encarga de crear una lista de nodos que tendrán la abstracción de identificadores como valor.

- ***addIdentificAListA()***:

Añade el identificador con el nombre que se pasa como parámetro a una lista de identificadores, es decir, a una lista de tipo *listaIdentificadores* que se especifique como parámetro. Únicamente realiza la inserción del identificador en caso de que este no se encuentre dentro de la lista, para lo cual llama a la función ***buscarIdentificador()***, que permite hacer la búsqueda; en caso de detectar que este ya se encuentra almacenado no lo agrega.

La función devuelve la posición en la que se encuentre el identificador dentro de la lista, ya sea que lo haya insertado o encontrado dentro de la lista.

- ***buscarIdentificador()***

Busca la posición en la que se ubica el identificador, cuyo nombre se pasa como parámetro, dentro de una lista, que también se indica como parámetro. En caso de estar vacía la lista, se indica y retorna un 0, en caso de haberlo encontrado regresa la posición en la que lo encontró y en caso de no haberlo localizado retorna un -1.

- ***imprimirListaIdent()***

Muestra en pantalla los valores de la posición, el nombre y el tipo para cada identificador almacenado en la lista de identificadores que reciba como parámetro.

En la función principal del programa se crea una instancia de tipo *listaIdentificadores*, mediante esta se impleenta la tabla de símbolos en el programa; este elemento se inicializa con la función correspondiente.

Posterior a la finalización del análisis léxico, se llama a la función ***imprimirListaIdent()***, para mostrar en pantalla el contenido de la tabla de símbolos; de manera que el usuario visualice el resultado con los identificadores detectados.

Técnica de inserción

Cada vez que el analizador léxico detecte un componente de tipo identificador, en las funciones de activación especificadas para el componente se llama a la función ***addIdentificALista()***, pasando como parámetros la dirección de la tabla de símbolos creada y el nombre del identificador a insertar, referido mediante *yytext.* El valor que devuelva la función se almacenará en la variable global *valorToken*, como este representa la posición del identificador dentro de la tabla será a su vez el valor que se enviará como parámetro a la función ***addFinalListaToken()***, que nos permitirá crear el token correspondiente al identificador, enviando a su vez el número de clase que es la cinco.

La función ***addIdentificALista()*** consultara primeramente si el identificador que se desea insertar ya se encuentra dentro de la lista, haciendo un llamado a la funcion ***buscarIdentificador()***, dependiendo del valor devuelto por esta función puede determinar si debe insertarlo dentro de la lista o no.

En caso de que la lista se encuentre vacía realizará la inserción del identificador, creando un nuevo nodo de tipo *nodoIdentificador* al cual le asociará a su miembro *valor* la referencia al elemento de tipo *identificador* que resulte de llamar a la función ***crearIdentificador()***, su referencia *next* la inicializa con *null* y la referencia *head* de la lista la asocia a este nodo creado, indicando que será el primer elemento de la lista. Por último, incrementa en uno el tamaño de la lista y devuelve la posición del identificador, que en este caso es cero.

En caso de que el valor devuelto por la función ***buscarIdentificador()*** sea –1, significa que el identificador no se encuentra en la tabla, por lo que realiza la creación de un nuevo nodo de tipo *nodoIdentificador* de la misma forma descrita con anterioridad, pero en este caso, recorre todos los nodos de la lista hasta llegar al último y su referencia *next* la asocia con la dirección del nuevo nodo creado, indicando que este nodo se ha insertado al final de la lista. Nuevamente, se incrementa en uno el tamaño de la lista y se devuelve la posición del identificador insertado.

Si la dirección que devuelve ***buscarIdentificador()*** es distinta de cero o –1, la propia función ***addIdentificALista()*** devolverá este valor y no realiza la inserción, entendiendo que el identificador ya se encontraba en la lista.

Técnica de búsqueda

Para implementar la búsqueda de los identificadores dentro de la tabla de símbolos se cuenta con la función ***buscarIdentificador()***, como se mencionó anteriormente, esta indicará si el identificador ya se encuentra dentro de la lista de identificadores dependiendo del valor entero retornado.

Esta función recibe como parámetros la lista de tipo *listaIdentificadores* en la cual se realizará la búsqueda, y el nombre del identificador que se busca. Con una estructura *if* se verifica si la lista está vacía, en tal caso se retorna un cero; en caso contrario, se recorren uno a uno los elementos que conforman la lista, comparando con cada identificador almacenado, si es idéntico a alguno se

retorna el valor del contador que lleva la posición evaluada. Si llega al final de la lista y no lo encuentra retornará un –1 indicando que no se ha encontrado dentro de la lista.

Definición tabla de literales

Una de las funcionalidades del analizador léxico será crear y actualizar la tabla de literales; de acuerdo al planteamiento del problema, esta tabla contendrá todas las constantes de tipo cadena que el analizador reconozca durante la lectura del programa; a las cuales deberá asignarles su correspondiente posición dentro de la tabla, de tal manera que sea posible identificarlas.

Dado que la tabla de literales también es una tabla dinámica, al igual que en el caso de la tabla de símbolos, se consideró adecuado realizar la implementación mediante una lista ligada de nodos, donde cada nodo contendrá una cadena, que cuenta con dos campos: la posición de la cadena dentro de la tabla y el dato que será la cadena en sí, es decir, la secuencia de caracteres correspondiente.

Para realizar la abstracción adecuada, se consideró la declaración de tres estructuras de datos: *cadena*, *nodoCadena* y *listaCadenas*.

La primera estructura, *cadena*, es la abstracción de una cadena con los elementos que debe considerar para ser almacenada en la lista, en este caso, dichos elementos, especificados como miembros de la estructura son: *posicion*, de tipo *int*, y *dato*, que es un apuntador a carácter.

La estructura *nodoCadena* será la abstracción de un nodo que forma parte de la lista ligada y que almacenará una cadena; cuenta con dos campos: *valor* y *next*, el primero es de tipo *struct cadena*, mientras que el segundo es un apuntador a un elemento del mismo tipo, *nodoCadena*; es decir, el valor del nodo será un elemento de tipo *cadena* y este nodo a su vez se relaciona con otro del mismo tipo mediante un apuntador, indicando que, dentro de la lista, será su elemento consecutivo. La tercera estructura desarrollada, *listaCadenas*, corresponde a la lista ligada que representa la tabla de literales, cuenta con dos elementos: *head* y *tamanio*, el primero es un apuntador a un elemento de tipo *nodoCadena*, y será el primer elemento de la lista, mientras que el segundo es un entero que almacenará el número de nodos que forman parte de la lista.

Las funciones desarrolladas para el manejo de la tabla de literales son las siguientes:

- ***crearCadena()***

Devuelve una estructura de tipo *cadena*, asignando a sus miembros la posición y el contenido indicados como parámetros de entrada.

Para el manejo del espacio que se debe reservar para el almacenamiento correcto de la cadena, se emplea la función ***malloc()***, a partir del tamaño de la cadena, pasado como parámetro, se puede obtener la cantidad de bytes necesarios para su almacenamiento, multiplicándolo por el tamaño de un carácter.

- ***crearListaCadenas()***

Devuelve una estructura de tipo *listaCadenas*, con sus miembros *head* y *tamanio* inicializados; es decir, crea una lista de nodos de tipo *nodoCadena*, asociando a la dirección del primer nodo el valor *null*, puesto que al momento de su creación se desconoce cuál será el primer valor de la lista; y a su miembro *tamanio* el valor de cero, al no contener ningún nodo.

- ***addCadenaALista()***

Añade al final de la lista de tipo *listaCadenas*, indicada como primer parámetro, un nodo que contenga la cadena que se pasa como segundo parámetro, también recibe el tamaño de dicha cadena como tercer parámetro. Devuelve la posición dentro de la lista en la que se añadió la cadena. Esta función permite realizar la inserción de las cadenas dentro de la tabla de literales.

- ***imprimirListaCadenas()***

Muestra en pantalla los valores de la posición y el contenido para cada cadena almacenada en la lista de tipo *listaCadenas* recibida como parámetro, sobre la cual itera mediante un ciclo *while*, recuperando la dirección de cada nodo y consultando el valor de los miembros de la cadena que contenga.

Dentro de la función principal del programa se crea una instancia de tipo *listaCadenas* y se inicializa mediante las funciones correspondientes; esta será la estructura mediante la cual se implementa dentro del programa la tabla de símbolos.

Una vez terminado el análisis, se muestran en pantalla las cadenas que se han almacenado en la tabla junto con su posición, haciendo un llamado a la función ***imprimirListaCadenas()*** descrita anteriormente, de manera que el usuario pueda visualizar el contenido resultante de la tabla de literales.

Técnica de inserción

Una vez que el analizador detecta un componente de tipo cadena, dentro de las funciones de activación especificadas para el componente, se almacena en la variable global *numChar* el número de caracteres que componen la cadena, obtenidos mediante la función ***strlen()*** y colocando como parámetro la referencia a la cadena, *yytext.*

Posteriormente, se llama a la función ***addCadenaALista()***, pasando como parámetros la dirección de la tabla de literales creada, la referencia a la cadena a insertar y el valor almacenado por *numChar*. El valor devuelto por la función se almacenará en la variable global *valorToken*, al ser la posición de la cadena, será a su vez el valor que se enviará como parámetro a la función ***addFinalListaToken()***, que nos permitirá crear el token correspondiente a la cadena, así mismo se envía el número de clase que será la ocho.

Ampliando el funcionamiento de la función ***addCadenaALista()***, es importante mencionar que al interior se crea un apuntador de tipo *nodoCadena*, que considere el espacio suficiente para almacenar una estructura de este tipo, mediante memoria dinámica se hace el manejo de este aspecto; este nodo será inicializado, asignando a su valor el elemento de tipo *cadena* devuelto por la función ***crearCadena()*** al pasar como parámetros el tamaño de la lista (siendo la posición de la cadena a insertar), el contenido que será la referencia a la cadena a almacenar y el tamaño de dicha cadena. El elemento *next* se inicializa con *null*.

Dentro de la misma función se tiene una estructura *if* con la cual se evalúa si la lista es vacía, en tal caso se asocia la dirección del nodo creado al elemento *head* de la lista, indicando que este será el primer nodo; en caso contrario, se realiza un recorrido sobre los elementos de la lista hasta llegar al último, entonces se asocia la direccion del nodo creado al elemento *next* del último nodo encontrado, de tal manera que se agrega al final de la lista. Finalmente, se incrementa en uno el tamaño de la lista y se devuelve el valor de la posición del nuevo nodo insertado.

Definición de tokens

Una de las principales funciones del analizador léxico es generar un **token.** Dicho **token** tiene dos campos: Clase y Valor.

En el campo **Clase**, tendrá el valor de la Clase a la que pertenece el componente léxico.

Mientras que en el campo *valor*, varía dependiendo del componente léxico, el *valor* para el token de *identificador* es la posición dentro de la tabla de símbolos. El *valor* para los tokens de *palabras reservadas, operadores lógicos, operadores sobre cadenas,* y *operadores de asignación;* será la posición dentro de su correspondiente tabla (catálogo). Para las *constantes de cadenas* el *valor* será la posición dentro de la tabla de literales. Para *operadores aritméticos* y *símbolos especiales,* su *valor* será el mismo carácter o su correspondiente ASCII. Finalmente, para las *constantes enteras* como *valor* será el mismo valor de la constante entera. Si es negativo se pondrá el signo – si es con el signo + se omitirá

En base a lo anterior se optó por utilizar una lista ligada con tipo de datos *token***,** el cual representa la colección de un *short clase* y un *int valor*.

La Lista ligada tiene como colección únicamente un apuntador de tipo *struct token**.

Al momento de detectar un componente léxico que cumple con las expresiones regulares definidas, para todos los componentes, se realizan dos funciones esenciales para generar los tokens: encontrar el valor del token y agregarlo a la tabla de tokens.

Para encontrar el valor del token se utiliza una variable globla *valorToken,* y se mandan a llamar a las funciones ***buscaOperadores(), buscaReservadas(), addCadenaALista(), addIdentificaALista(), obtenerAscii(), convierteInt().*** A continuación, se específica qué devuelve cada una de estas funciones:

- ***buscaOperadores(), buscaReservadas()***

Son utilizados para los componentes que se encuentren en catálogos y devuelven la posición del operador o cadena a buscar, dicho cadena u operador a buscar es el componente léxico encontrado, y se encuentra en *yytext***.** Por otro lado, al tratarse de catálogos, se pasa como parámetro el arreglo de estructuras en donde se encuentra ese catálogo.

- ***obtenerAscii()***

Esta función devuelve el valor en ASCII, del símbolo especial encontrado. Al momento de encontrar un componente léxico correspondiente a esta clase, se manda a llamar esta función y se le pasa como parámetro el carácter encontrado. Dependiendo de cuál sea el carácter encontrado, la función devolverá su valor en ASCII.

- ***obtenerInt()***

Al momento de encontrar un componente léxico correspondiente a una constante entera, está la lee como una cadena, por ello, esta función devuelve el valor en entero, pero como *int*, no como un arreglo de *char*.

No basta con usar la función ***atoi()***, ya que nuestro analizador debe detectar (+123) o (-8369), es decir, números con signos encerrados entre paréntesis. Por lo tanto, esta función realiza un *if* y si detecta que no se está mandando un número con símbolo, es decir, solo "123", sí realizará la función ***atoi()*** de esa cadena.

En caso contrario, va a distinguir si la cadena que se está mandando se tratará de un número positivo o negativo, si es positivo, va a copiar carácter por carácter después del signo "+" hasta antes de llegar al ")", con esto se estará copiando únicamente el valor del número sin su signo. En caso de que sea negativo, copiará desde el signo "-" hasta antes de llegar al ")", con esto se copiará el valor del número negativo con todo y su signo.

Una vez eliminando los paréntesis de la cadena y su signo (+ de ser el caso), esa cadena ahora sí la convierte a *int* con la función ***atoi()***. Y este mismo valor es el que se retorna a *valorToken***.**

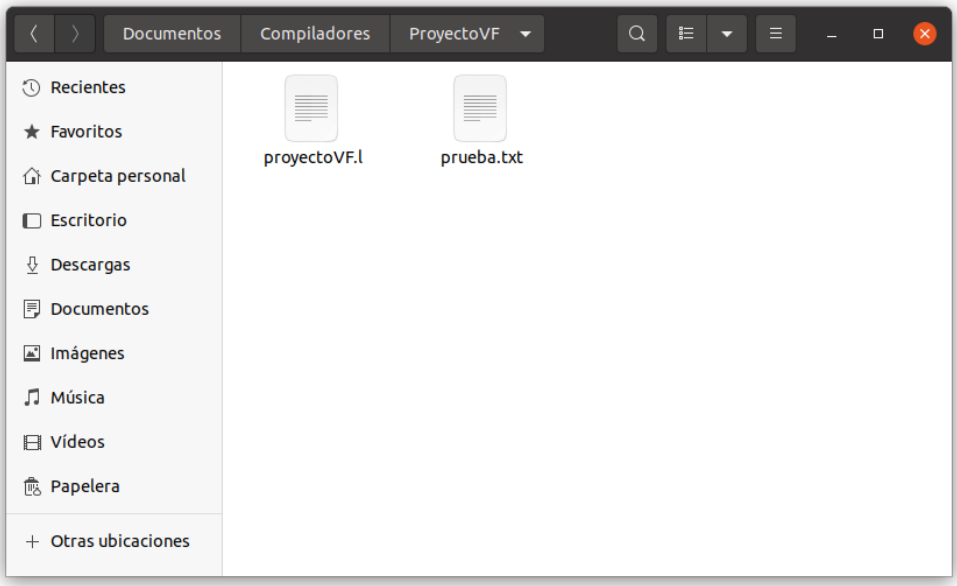
Técnica de inserción

Para añadir se utiliza la función ***addFinalListaToken()***, donde se pasa como parámetro la lista ligada a donde se va a añadir, el valor de la clase y el ***valorToken.***

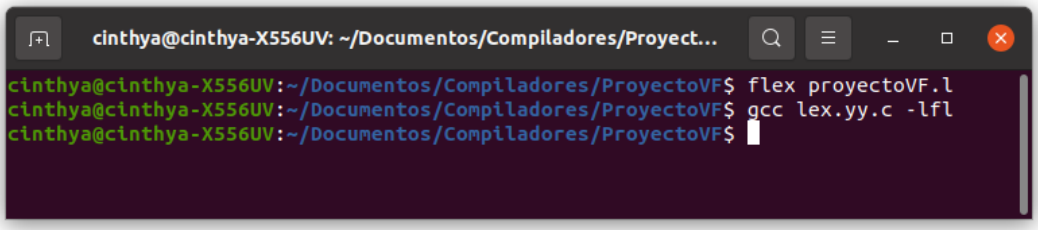
Dependiendo del componente léxico que haya encontrado será el valor de la clase y el ***valorToken,*** como ya se explicó será su respectiva, posición, valor en ASCII o número.

Instrucciones de ejecución

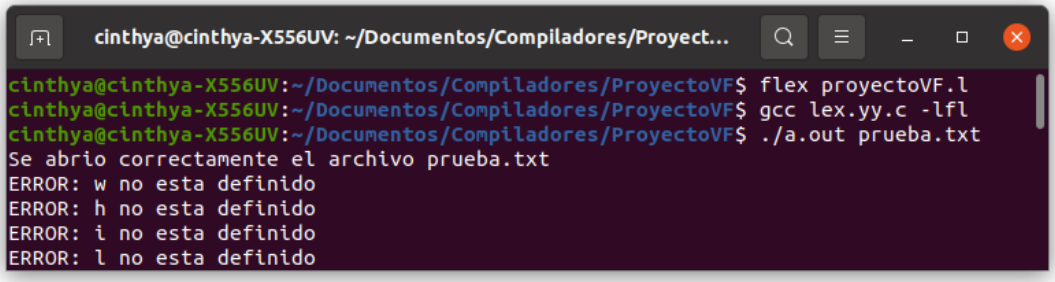
Para ejecutar el programa, se debe ubicar tanto el programa "programaVF.l", correspondiente al analizador léxico, como el programa a analizar con extensión ".txt" dentro del mismo directorio.



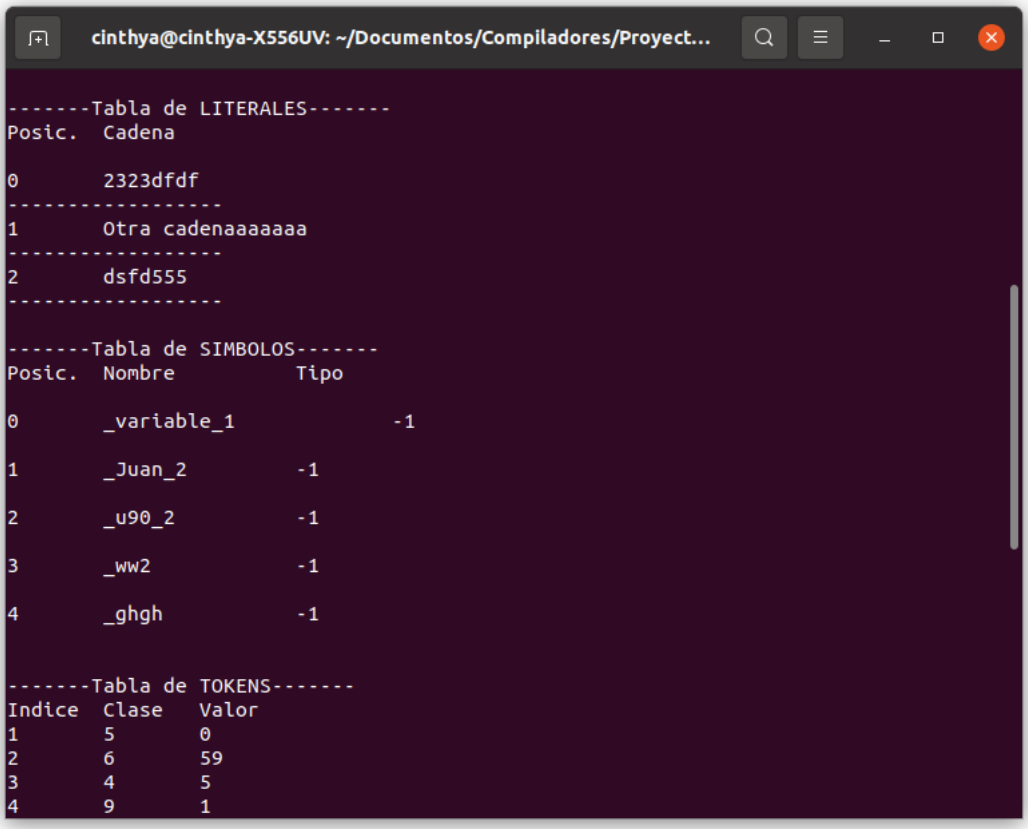
Desde la terminal se accede al directorio en el que se ubica el programa y se ingresa el comando "flex proyectoVF.l" para realizar la compilación del programa ".l". Posteriormente se ingresa el comando "gcc lex.yy.c -lfl" para la compilación del programa en C.



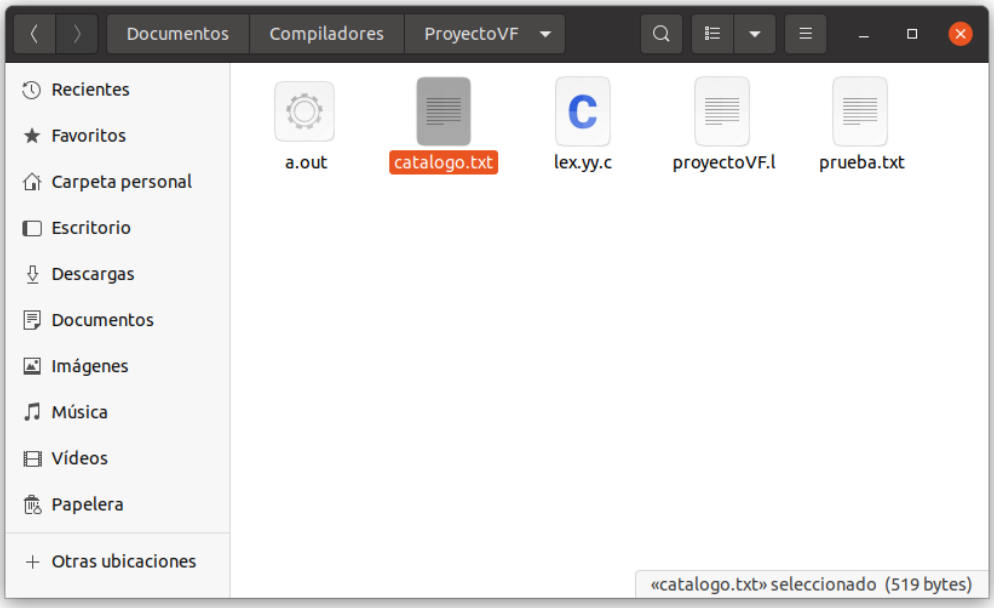
Para su ejecución se deberá ingresar el comando "./a.out <nombre_del_programa>.txt", donde <nombre_del_programa> se sustituye por el nombre del archivo que contenga el programa que se desea analizar.



Se mostrará en pantalla la salida correspondiente a los errores, la tabla de literales, de símbolos y la de tokens.



Adicionalmente, en la carpeta en la que se ubique el programa se creara un archivo llamado "catalogo.txt" que contiene los catálogos creados para los componentes léxicos, con los que trabaja el programa.



Conclusiones

Morales Ortega Carlos

Gracias al presente proyecto se ha aprendido a realizar un analizador léxico en flex/lex, además el hecho de desarrollarlo, implico saber a detalle cómo funciona, sus funciones, e importancia del analizador léxico, ya que sin esto no se hubiera podido desarrollar el proyecto.

Por lo tanto, se reforzó el conocimiento adquirido en clase y lo teórico lo plasmamos de manera teórica, esto resultó bastante interesante, ya que, al construir nuestro propio analizador, recurrimos a nuestros conocimientos acerca de este tema y de materias anteriores.

El uso de las expresiones regulares es fundamental para poder definir a nuestros componentes léxicos y sobre ellos se basa todo, al tener mal estas expresiones, entonces nuestro analizador léxico no haría su función esperada y nos arrojaría errores o simplemente no haría su función de forma esperada o adecuada.

La parte compleja del proyecto radica en realizar las funciones necesarias al momento de detectar un componente léxico, es decir, crear los catálogos, tablas de literales, símbolos y generar los tokens. Aunque, fue la parte que más nos obligó a aprender acerca del lenguaje C y del analizador léxico.

Finalmente, el proyecto obligó a investigar más allá de lo que ya hemos aprendido y además refleja nuestra creatividad para poder solucionar el problema propuesto o bien abordar el problema y entregar una solución.

Vélez Grande Cinthya

Mediante el desarrollo del presente proyecto fue posible aplicar todos los conocimientos adquiridos en teoría con respecto a la implementación de un analizador léxico. Se revisaron aspectos como el diseño de las expresiones regulares para el reconocimiento de clases de componentes léxicos, la implementación de los catálogos mediante tablas estáticas, así como la creación y actualización de las tablas de símbolos y literales mediante tablas dinámicas. Otros aspectos revisados fueron la creación de tokens para cada clase de componente léxico y el manejo de errores.

Considero que es muy interesante ver la forma en que se conjunta todo lo que hemos aprendido en la parte teórica y poder programarlo; pese a que no fue una tarea fácil, logramos analizar y comprender adecuadamente el problema, de manera que nos fue posible abstraer el comportamiento deseado del analizador y crear todos los elementos pertinentes para su desarrollo, desde estructuras de datos, hasta funciones que nos permitieran un manejo adecuado de la información, desarrollando capas de abstracción que nos facilitaran tareas más complejas.

Cabe mencionar que fue un reto coordinar esfuerzos y dividir el problema de tal manera que el trabajo fuera equitativo para ambas partes; sin embargo, considero que esto lo logramos de una forma óptima, reflejando nuestro esfuerzo conjunto en el trabajo desarrollado.

Finalmente, se cumplió satisfactoriamente el objetivo del proyecto, pues se desarrolló un analizador léxico en lex/flex que reconociera correctamente el lenguaje planteado en clase y que realizara las funcionalidades básicas planteadas para un analizador léxico, como la creación y actualización de la tabla de símbolos y de literales; así como la generación de tokens y reconocimiento de errores léxicos.