UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROYECTO 1

ANALIZADOR LÉXICO

NOMBRES:

Piña Rossette Marco Antonio

Reyes Mendoza Miriam Guadalupe

ASIGNATURA:
Compiladores

GRUPO: 3

SEMESTRE: 2023-1

FACULTAD DE INGENIERÍA

ANALIZADOR Léxico

DESCRIPCIÓN DE PROBLEMA

Las clases de los componentes léxicos válidos para el analizador léxico son:

CLASE	DESCRIPCIÓN
0	Palabras reservadas (ver tabla).
1	Identificadores. Iniciar con $ y le sigue al menos una letra minúscula o mayúscula. Ejemplos: $ejemplo, $Varible, $OtraVariable, $XYZ
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).
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.
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”
5	Símbolos especiales [ ] ( ) { } , : ;
6	Operadores aritméticos + - * / % \ ^
7	Operadores relacionales (ver tabla)
8	Operador de asignación =

La tabla nos muestra las diferentes clases que tendremos y junto con ellas, las características que deben cumplir los componentes léxicos que reconocerá el analizador, por lo que es importante definir de forma clara las expresiones regulares que nos permitiran el reconiccimento y distinción de cada clase.

CLASE	PALABRA RESERVADA	EQUIVALE EN C
0	alternative	case
1	big	long
2	evaluate	if
3	instead	else
4	large	double
5	loop	while
6	make	do
7	number	int
8	other	default
9	real	float
10	repeat	for
11	select	switch
12	small	short
13	step	continue
14	stop	break
15	symbol	char
16	throw	return

CLASE	EQUIVALE EN C
0	<
1	>
2	<=
3	>=
4	= =
5	! =

EXPRESIONES REGULARES

PALABRAS RESERVADAS

Para esta clase, la expresion regular es muy sencilla puesto que no sigue alguna regla en espcifico, solo se busca que sea una de la palabras que se encuntran listadas.

$$PALRES = "alternative"|"big"|"evaluate"|"instead"|"large"|"loop"|"make"|"number"|"other"|"real"|"repeat"|"select"|"small"|"step"|"stop"|"symbol"|"throw"$$

De este modo, decimos que una palabra reservada puede ser cualquiera de las listadas, tal cual y como estan declaradas, sin ninguna variación.

IDENTIFICADORES

Se sabe, que los identificadores pueden tener cualquier nombre, sin importar si tiene una estructura entre mayúsculas y minúsculas, es decir, estas pueden ir intercaladas o sin algún formato o regla específica. lo que nos interesa en este caso es que para saber que se trata de un identificador nosotros tenemos que poner el símbolo '$'.

$$LETRA = [a - zA - Z]$$
$$IDENT = \$ < LETRA >^{+}$$

Como vemos, el símbolo se encuentra al inicio de la cadena y de esta forma cumple las reglas establecida para esta clase, teniendo el símbolo '$' para el Reconocimiento de un identificador.

CONSTANTES NÚMERICAS ENTERAS

En este caso, fue más fácil dividir la estructura de los componentes que llevar a nuestra clase, puesto que aquí identificará constantes enteras decimales y octales. En este caso, definiendo al dígito y al decimal, se reconoce al dígito debido a que será utilizado posteriormente y de este modo no estaremos definiendo más que una vez su valor para su posterior utilización.

$$DIGITO = [0 - 9]$$
$$DECIMAL = [1 - 9] < DIGITO >^{*}$$
$$OCTAL = [Oo][0 - 7]^{*}$$
$$ENTERO = < DECIMAL > | < OCTAL >$$

Cómo vemos un número decimal puede estar formado por números del del 1 a infinito, como vemos nosotros estamos poniendo que las constantes decimales se definen primero con un número que va del 1 al 9, seguido o no de cualquier otro número o números que esten entre 0 a 9. Para los números octales, vemos que tenemos que estos iniciarán con la letra O, ya sea mayúscula o minúscula y que podrán o no, continuar con un valor o valores que se encuentre entre 0 a 7. Teniendo estos 2, entonces nosotros definimos a un entero como la combinación entre octales y decimales.

CONSTANTES NÚMERICAS REALES

Es una expresión regular sencilla ya que solamente necesitamos hacer la implementación del punto, en este caso como podemos ver nos interesa que forzosamente se encuentre uno o más dígitos, seguido de un punto y posteriormente seguido de uno o más dígitos.

$$REAL = < DIGITO >^{+} [.] < DIGITO >^{+}$$

Reconocemos números que no carezcan de un dígito antes del punto y después del punto.

CONSTANTES CADENAS

Esta fue una expresión regular que cambiamos varias veces durante el diseño de nuestro analizador léxico, y ya que en un inicio nosotros habíamos definido todos los caracteres que son aceptables, sin embargo nos causaban muchos errores al momento de reconocerlos en lex, por lo que optamos mejor solamente excluir los caracteres que no son permitidos.

$$CARACTER = ' [^\varepsilon " ']^+ '$$
$$CADENA = " [^\varepsilon " ']^+ "$$

Como podemos observar nosotros estamos poniendo que entre comillas, puede ir cualquier carácter que no sean otras comillas o una cadena vacía.

SIMBOLOS ESPECIALES

$$SIMBOLOS\ ESPECIALES = [\ |\ ]\ |(\ |\ )\ |\{\ |\ \}\ |,|:|;$$

Para lo que son símbolos y operadores, sólo basta con definir cuáles son los validos. No es necesario crear una expresión regular tan grande oh tan compleja, sino una que sólo identifique cuáles son los permitidos.

OPERADORES ARITMÉTICOS

$$OPERADORES\ ARITMETICOS = +|*|\ /\ |\%|\ \backslash\ |\hat{}\ |-$$

Definimos, tal cual los símbolos operadores aritméticos permitidos.

OPERADORES RELACIONALES

Del mismo modo que los 2 anteriores, nosotros definimos cuáles son los operadores válidos.

$$OPERADORES\ RELACIONALES = < | > | \leq | \geq | == |! =$$

OPERADORES DE ASIGNACIÓN

Al ser sólo uno, solamente lo reconocemos.

$$OPERADORES\ ASIGNACIÓN = =$$

SOLUCIÓN Y FASES DE DESARROLLO

Tenemos que crear un analizador lexico, para esto, debemos tener claro que es la primera fase del compilador, también conocido como escáner. Convierte el programa de entrada de alto nivel en una secuencia de tokens. Se puede implementar con los autómatas finitos deterministas, cuyo resultado es una secuencia de tokens que se envía al analizador para el análisis de sintaxis.

Las funciones principales de esta fase son:

- Identificar los componenetes léxicos.
- Clasificar los componenetes léxicos en clases como constantes, palabras reservadas, identificadores, entre otros e introdúzcirlas en diferentes tablas.
- Ignorarar los comentarios y espacios en blanco del programa fuente.
- Identificar tokens no válidos.

ANÁLISIS

El anlizador léxico toma el código fuente modificado que está escrito en forma de oraciones. En otras palabras, ayuda a convertir una secuencia de caracteres en una secuencia de tokens. El analizador léxico divide esta sintaxis en una serie de tokens. Elimina cualquier espacio adicional o comentario escrito en el código fuente.

Si el analizador léxico detecta que el token no es válido, genera un error. La función de analizador lexico en el diseño del compilador es leer flujos de caracteres del código fuente, buscar tokens legales y pasar los datos al analizador de sintaxis cuando lo requiera.

La tarea principal del análisis léxico es leer los caracteres de entrada en el código y producir tokens. Una secuencia de caracteres que no es posible escanear en ningún token válido es un error léxico. Datos importantes sobre el error léxico:

- Los errores léxicos no son muy comunes, pero deben ser manejados.
- Los errores ortográficos de identificadores, operadores y palabras clave se consideran errores léxicos.
- Generalmente, un error léxico es causado por la aparición de algún carácter ilegal, principalmente al principio de un token.

PLANIFICACIÓN

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

- Reconocimiento del archivo de entrada y salida → Antonio
- Definición de las expesiones regulares y estructura para cada clase → Miriam
- Reconocieminto del espacio en blanco, comentarios y salto de linea → Miriam
- Definición de tokens → Antonio
- Reconocimeinto y distinción de tokens por clase → Antonio y Miriam
- Impresión de tablas → Antonio
- Archivo de entrada → Antonio

DISEÑO E IMPLEMENTACIÓN

Además 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 creación, de lo que ya sabíamos por los ejemplos vistos en clase, en este caso la generación y el reconocimiento de las diferentes expresiones regulares para cada clase, al igual que la creación y lectura de un archivo de entrada y salida respectivamente. Se definió la estructura que iba a llevar el token, para esta definimos que iba a contener la clase y el valor.

Siguiendo las tablas de símbolos y constantes fueron implementadas usando una lista ligada simple escrita en C y se utiliza como biblioteca externa al programa $SLL.h$. Al utilizar listas ligadas, los métodos de inserción fueron más sencillos, pues se utilizaron los apuntadores a los nodos finales para hacerlo más eficiente.

Una vez definidas las difernetes clases y su estructura, creamos mediante SLL las tablas para cada clase, para posteriormente declarar las funciones que nos permiten la creacion de tokens, al igual que la función que nos permitia imprimir las diferentes tablas.

Un análisis algo complejo fue la deteccion de espacios en blanco y lineas de comentario pues aunque existe en lex una función descrita como $[:\mathrm{blank}:]$, por alguna razón no la reconocía y la marcaba como un error por lo que decidimos mejor ponerlo como una expresión regular.

Por último, creamos la generación de tokens para cada clase, esto fue lo más complicado debido a que tuvimos que hacerlo de forma específica para cada una, ya que cada clase cuenta con características diferentes que debe cumplir para generar un token. Ya dentro del main, solo mandamos esta tabla y como para las palabras reservadas y los operadores relacionales tenemos un valor que debe de ser el mismo que el del token, entonces hacemos un ciclo iterativo que nos permita ver en qué posición está el que se declaró.

Los errores qué se llegan a generar se marcan y se dan a conocer mediante la impresión de quién es lo que está generando error, en este caso cuando no reconoce una palabra reservada o cuando un identificador ya existe. para terminar, debemos saber que nuestro archivo de salida nosotros estamos imprimiendo las tablas de nuestras clases, al igual que la tabla de tokens dónde ponemos su valor y su número de clase.

Se utilizó un método funcional para la implementación de las acciones que debía tomar el analizador en función de lo que reconociera.

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 ***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

La realización de este proyecto fue muy enriquecedora, pues logramos desarrollar nuestras habilidades en Lenguajes Formales y Autómatas, así como también aprendimos un nuevo lenguaje para la realización de compiladores.

Igualemnte pudimos retomar conocimientos anteriores de estructuras de datos para poder implementar las tablas requeridas para el analizador léxico. Finalmente, considero que se cumplió el objetivo de obtener la primera parte del compilador funcional.

REYES MENDOZA MIRIAM GUADALUPE

La creación del analizador léxico, fue algo compleja, al inicio nosotros decidimos empezar con la definición de expresiones regulares, esto fue algo sencillo, sin embargo hubo varias veces en que para la clase cuatro tuvimos que hacer diferentes modificaciones ya que nosotros queríamos poner todos los caracteres que reconocía hasta que encontramos que una mejor solución era exceptúan solamente los que no estaban permitidos.

la parte más compleja en el desarrollo de este proyecto fue el poder crear las diferentes tablas y sobre todo los tokens para cada clase, ya que se necesita Hacer una diferente estructura que nos permita reconocer cuando se genera un error Y se tiene que definir clase a clase cómo va a estar generando el token. Sin duda esta fue la parte más tardada, pero que sin embargo nos fue de mucha utilidad para aprender a manejar esta parte, una vez teniendo eso solamente nos bastó con identificarlo en las tablas.

Sin duda fue un proyecto bastante complejo pero muy óptimo para el aprendizaje y análisis de las etapas de diseño de un analizador leíxco, al igual que su implementación y creación, dejándonos obtener un escáner el cual es la primera etapa para la creación de un compilador completo.