Documentación Analizador Léxico

Para este primer programa que nos encargó la profesora M.C Laura Sandoval Montaño del grupo 02 de la materia de Compiladores en el semestre 2022-1, tuvimos que elaborar el código que cumple la función de analizador léxico dentro de un compilador.

El lenguaje de programación que busca procesar el analizador para corroborar su validez léxica fue elaborado durante la clase con la contribución de todos los miembros del grupo. Se decidió tener 10 clases diferentes de componentes léxicos, los cuales conforman en su totalidad el lenguaje. Cada clase tiene sus reglas específicas para definirse y puede ser representada mediante una expresión regular. A continuación, se muestran las clases y sus expresiones regulares correspondientes:

Clase 0: Palabras reservadas.

Valor	***Palabra reservada***
0	cadena
1	caracter
2	else
3	entero
4	for
5	if
6	real
7	return
8	void
9	while

Expresión regular:

(cadena)|(caracter)|(else)|(entero)|(for)|(if)|(real)|(return)|(void)|(while)

Clase 1: Operadores aritméticos: + - * / $

Expresión regular:

[\+\-*\/\$]

Clase 2: Operadores de asignación.

Valor	***Operador de asignación***	**significado**
0	~	igual
1	+~	más igual
2	-~	menos igual
3	*~	por igual
4	/~	entre igual
5	$~	módulo igual

Expresión regular:

```
[\+\-\*\/\$]?~
```

Clase 3: Símbolos especiales: ( ) { } [ ] & , :

Expresión regular:

```
[\(\)\{\}\[\]&,:]
```

Clase 4: Operadores relacionales.

Valor	***Operador relacional***	**significado**
0	^^	mayor que
1	^”	menor que
2	==	igual que
3	^^=	mayor o igual
4	^”=	menor o igual
5	<>	diferente

Expresión regular:

```
(^[^\”]=?)|(==)|(<>)
```

Clase 5: Identificadores: Inician con una letra, le siguen hasta cinco letras o dígitos y terminan con guion bajo.

Expresión regular:

```
[A-Za-z]([A-Za-z]|[0-9]){0,5}_
```

Clase 6: Constantes numéricas enteras: signados o no signados, máximo 6 dígitos y mínimo 1.

Expresión regular:

```
[\+\-]?[0-9]{1,6}
```

Clase 7: Constantes numéricas reales: no signados.

Expresión regular:

```
([0-9]*\.[0-9]+)|([0-9]+\.[0-9]*)
```

Clase 8: Constantes cadenas: inician y terminan con comillas dobles, en medio se tienen de 0 a 38 caracteres, cualquier carácter excepto las comillas dobles.

Expresión regular:

```
\"[^\"]{0,38}\"
```

Clase 9: Constante carácter: cualquier carácter encerrado entre comillas sencillas.

Expresión regular:

```
‘.’
```

Propuesta de solución:

Para empezar a implementar una solución al problema de crear un analizador léxico, con lo primero que decidí empezar fue con la creación de las expresiones regulares que me definieran todas las clases.

Una vez que conté con las expresiones de las clases, tuve que planificar en qué orden deseaba que mi analizador fuera encontrando las coincidencias. Esto fue muy importante para lograr que identificara correctamente las clases y no me cortara alguna para dividirla entre dos clases u otros errores. Finalmente el orden que me funcionó y con el que quedé satisfecho fue el siguiente:

1. Palabras reservadas (clase 0)
2. Identificadores (clase 5)
3. Operadores de Asignación (clase 2)
4. Operadores Aritméticos (clase 1)
5. Operadores Relaciones (clase 4)
6. Cadenas (clase 8)
7. Carácter (clase 9)
8. Constante real (clase 7)
9. Constante entera (clase 6)
10. Símbolos especiales (clase 3)
11. Espacios (utilidad extra)
12. Saltos de línea (conteo de línea para errores)
13. Error (errores que no coinciden con lo demás)

Ya que se contaba con esto, lo siguiente fue decidir cómo quería manejar mis salidas del programa y las acciones al realizar al encontrar un componente léxico válido o inválido. Para esto de manera general decidí que se escribiría en el archivo de Tokens, el valor correspondiente del siguiente formato: **(N, M)**, donde N es clase y M es la posición en el catálogo, en la tabla o su mismo valor para las clases 1, 3 y 9.

Para las clases de Constantes, es decir la de cadena, carácter, entero y flotante; además de escribir en los tokens se escribiría en el archivo de Tabla de Literales correspondiente y se aumentaría la cuenta de una variable global de cada clase para determinar su posición dentro de la tabla. Para la inserción en las tablas correspondientes, tanto de Tokens como de literales, lo que se hacía era escribir en un archivo cada token o literal en una sola línea. Para el caso de los literales bajo el siguiente formato: **posición valor.** De este modo iba almacenando mis tablas directamente en archivos.

El caso más complejo de implementar fue el de los identificadores y la tabla de símbolos. Esto debido a que se debía checar si ya se había insertado el identificador previamente. Para lograr una implementación que me permitiera hacer

ésta búsqueda decidí utilizar una **lista ligada de nodos.** Los **nodos** eran una **estructura** que consistía de 4 campos: un apuntador al siguiente nodo (NULL si era el último), un entero indicando su posición, una cadena indicando su nombre de identificador (con el que se realiza la búsqueda) y un entero de tipo (inicializado en -1). La lista era otra estructura que constaba con dos apuntadores a nodos, uno al nodo inicial de la lista y otro al nodo final de la lista.

Para realizar la búsqueda de coincidencias tomaba el apuntador al inicio de la lista y comparaba el campo de nombre con el de mi búsqueda actual, si no coincidían usaba el campo de apuntador al siguiente para llegar al siguiente nodo y volver a comparar. Así sucesivamente hasta que el apuntador siguiente de un nodo fuera NULL, significando que llegué al final de la lista. Si no había coincidencias entonces tenía que insertar el nuevo nodo a la lista. Para esto usaba memoria dinámica para crearlo, lo inicializaba con su nombre de identificador, su posición (gracias a una variable global) y -1 como su tipo. Su apuntador a siguiente era NULL. Luego usaba el apuntador al final de la lista para llegar al último nodo y hacía que el apuntador a siguiente nodo dirigiera a mi nuevo nodo que no tuvo coincidencias en la búsqueda. Y finalmente actualizaba el apuntador al final de mi lista al nuevo nodo. De esta forma realicé la búsqueda e inserción en mi lista que representaba la tabla de símbolos. Mi búsqueda es lineal, con una complejidad de tiempo O(n) y la inserción es constante. Cada vez que se agregaba un nuevo nodo a la lista, escribía en el archivo de Tabla de Símbolos su información correspondiente bajo el siguiente formato: **posición nombre -1**.

Finalmente, para darle un buen formato a todo el programa, decidí agregar algunos mensajes guía en la consola al momento de ejecutarlo, al igual que darle título escrito dentro de los archivos. Esto no será ningún problema en un futuro, sólo necesitamos tenerlo presente para las siguientes etapas. Además decidí que para que se pudiera ver fácilmente la información de salida del programa, todo el contenido almacenado en los archivos también se imprimiría en la consola, con espacios y títulos correspondientes para que se entienda fácilmente.

Todo esto suena más sencillos, de lo que fue. Si bien tenía las cosas y detalles bastante claros, el diseño e implementación llevó su tiempo y esfuerzo. Tuve que checar el uso de archivos y apuntadores para poder implementar las estructuras necesarias y guardar la información de manera correcta.

¿Cómo correr el programa?

Para correr el programa lo que debemos hacer es los siguiente:

1. Abrir la consola o terminal de su sistema Linux de preferencia. (Debe contar con flex/lex yacc instalado, al igual que gcc)

2. Moverse al directorio donde tenga el código fuente (cd “directorio”) y preferentemente el archivo que se analizará.

3. Ejecutar el siguiente comando de compilación flex:

```
flex lexico.l
```

4. Luego ejecutar el siguiente comando de compilación de C para el archivo previamente compilado por flex que se llamará lex.yy.c:

```
gcc -lfl lex.yy.c
```

5. Finalmente ejecutar el comando de ejecución del archivo con el código objeto generado y pasando como argumento el nombre del archivo a analizar:

```
./a.out nombreArchivoParaAnalizar.txt
```

6. Con esto habrá logrado ejecutar exitosamente el programa. Se habrán mostrado los resultados en pantalla y se le habrán generado un total de 7 archivos nuevos a su carpeta, 5 de los cuales son los resultados:
    - lex.yy.c
    - a.out
    - Tokens.txt
    - TablaSimbolos.txt
    - TablaLiteralesCadenas.txt
    - TablaLiteralesNumeros.txt
    - Errores.txt

Conclusiones

La elaboración de esta primera etapa de un compilador, el analizado léxico, ha sido un proyecto bastante interesante que me ha ayudado a comprender mejor el funcionamiento de los analizadores. Creo que con sólo haber elaborado esta primera etapa y saber cómo se va revisando un archivo y se generan las tablas y tokens correspondiente me ha dado mucha más perspectiva no sólo de esta etapa sino del proceso completo.

A lo largo de su elaboración no tuve demasiadas dificultades para desarrollar el código. El mayor problema que tuve fue el que se me haya olvidado cerrar un archivo en el que escribía y me pase un buen rato encontrando que ese era el error. Esto debido a que me decía que estaba liberando un espacio en memoria dos veces y como para mi lista ligada que almacenaba los identificadores use asignación de memoria dinámica creía que por ahí estaba el error. Sin embargo, al final si lo pude encontrar y corregir.

Viendo el programa terminado y funcionando, estoy muy satisfecho con él. Creo que cumple con los requerimientos establecidos en las instrucciones y lo hace de una manera ordenada y medianamente eficiente. Si bien había ciertos aspectos que creo eran ambiguos en las instrucciones (como la longitud de los números reales, o como almacenar en los tokens los caracteres de la clase 9), pude resolver estos problemas bajo mi mejor criterio. Siento que tengo una buena base para el resto del compilador y espero poder juntarlo todo bien.