Universidad Nacional
Autónoma de México

Facultad de Ingeniería

Compiladores
Clave 434

Analizador léxico

Alumno:
Ángel Alvarado Campos

Grupo 3
Semestre 2023-1

Profesora: M.C. Laura Sandoval Montaño

Descripción del problema

El proyecto final de la asignatura consiste en desarrollar un compilador para un lenguaje determinado por el grupo, para lo cual es necesario afrontar el problema por medio de diferentes etapas que se encargan de cubrir diferentes aspectos y características del compilador final. En esta primera etapa, se solicitó abordar la etapa de *análisis léxico*, la cual se encarga de determinar a los componentes léxicos del lenguaje y enviar los *tokens* para la siguiente etapa de *análisis sintáctico*. Para implementar esta etapa se usó la herramienta de definición de analizadores léxicos Flex.
El lenguaje descrito por el grupo para el semestre correspondiente se formó con 9 clases de componentes léxicos, los cuales se describen con más detalle a continuación junto con sus correspondientes expresiones regulares implementadas con la herramienta Flex.

Clase 0: Palabras reservadas

Las palabras reservadas, definidas para este lenguaje, fueron las que se muestran en la siguiente tabla.

Valor	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

La expresión regular seleccionada para esta clase fue la siguiente.

```
palRes alternative|big|evaluate|instead|large|loop|make|number|other|real|repeat|select|small|step|stop|symbol|throw
```

Clase 1: Identificadores

En este lenguaje, los identificadores son nombrados de acuerdo con las siguientes reglas:

- Todo identificador debe iniciar con el carácter $.
- Posterior al carácter $, se contará con *al menos* una letra mayúscula o minúscula, así que no se permiten caracteres ajenos a estos en el identificador.

De lo anterior, la expresión regular seleccionada fue la siguiente.

```
identificador \$([A-Za-z]+)
```

Clase 2: Constantes numéricas enteras

Las constantes numéricas enteras se definen de la siguiente manera.

- Si se trata en base 10, se tendrán secuencias de 0 a 9, sin ceros a la izquierda a excepción del cero en sí.
- Si se trata en base 8 (octales enteros), deberán iniciar con “o” u “O”, seguido de secuencias de 0 a 7.

La expresión regular propuesta fue la siguiente.

```
ctsNumEnt ([1-9][0-9]*)|(0)|([oO][0-7]+)
```

Clase 3: Constantes numéricas reales

Las constantes numéricas reales se definen de la siguiente manera. Siempre debe llevar parte decimal y es opcional la parte entera.
La expresión regular propuesta fue la siguiente.

```
ctsNumReal ([1-9]([0-9]*))?\.([0-9]+)
```

Clase 4: Constantes cadenas

Las constantes de cadenas se definieron como sigue para el siguiente lenguaje.

- Toda cadena de longitud mayor a 1 debe estar encerrada entre comillas dobles (“”), y no puede tener como carácter componente a “ o ‘.
- Toda cadena de longitud igual a 1, debe estar encerrada entre comillas simples (‘’).
  - La cadena de longitud 1 con el carácter “ debe estar encerrada entre comillas simples, es decir: ‘“’.
  - La cadena de longitud 1 con el carácter ‘ debe estar encerrada entre comillas dobles, es decir: “‘”.

La expresión regular propuesta fue la siguiente.

```
chars [^\'\"]
ctsCadenas (\"\'\")|(\'\"\')|(\'{chars}\')|(\"{chars}({chars}+)\")
```

Clase 5: Símbolos especiales

Los símbolos especiales propuestos para este lenguaje son: [ ] ( ) { } , ; :
Entonces, la expresión regular propuesta para esta clase fue la siguiente.

```
simEsp \[|\]|\(|\)|\{|\}|,|;|:
```

Clase 6: Operadores aritméticos

Los operadores aritméticas propuestos para este lenguaje son: + - * / % \ ^
Entonces, la expresión regular propuesta para esta clase fue la siguiente.

```
opArit (\+)|(\-)|(\*)|(\/)|(\%)|(\\)|(\^)
```

Clase 7: Operadores relacionales

Los operadores relacionales propuestos para este lenguaje se muestran en la siguiente tabla.

Valor	Op. relacional
0	<
1	>
2	<=
3	>=
4	==
5	!=

Entonces, la expresión regular propuesta fue la siguiente.

```
opRela (<)|(>)|(<=)|(>=)|(==)|(!=)
```

Clase 8: Operador de asignación

La última clase está compuesta unicamente por el operador de asignación, el cual para el lenguaje propuesto es: =.
Entonces, la expresión regular propuesta fue la siguiente.

```
opAsig (=)
```

Propuesta de solución y fases del desarrollo del sistema

Análisis (planificación del proyecto)

Para el desarrollo de esta primera componente del compilador, la solución se planificó en tres etapas principales y secuenciales que se describen a continuación.

1. **Expresiones regulares:** En primer lugar y como base de todo el sistema de analizador léxico para el lenguaje establecido, se buscó identificar a las expresiones regulares suficientes que garantizaran la identificación de errores en archivos fuente por parte del analizador. Anteriormente se mostraron las expresiones regulares planteadas para cada clase, sin embargo, fue necesario definir expresiones regulares aparte. capaces de reconocer ciertos tipos de errores relacionados a componentes léxicos en concreto.
2. **Manejo de la información:** En segundo lugar, suponiendo un manejo correcto por parte de las expresiones regulares definidas, tanto de componentes en sí como de errores, fue necesario proponer una manera adecuada para implementar las estructuras que operan con la información de las clases correspondientes a los componentes léxicos del lenguaje.

Dichas estructuras refieren a las tablas de literales (de constantes numéricas reales y constantes cadenas), la tabla de símbolos (para identificadores), y los catálogos de palabras reservadas y operadores relacionales. En la sección de diseño se describe con mayor detalle la implementación realizada para el manejo de estas estructuras de información.

3. **Salidas al usuario:** Una vez establecido un manejo de la información correcto en función de los requerimientos, fue posible dedicar esfuerzos a las salidas del usuario, es decir, a los formatos en los que la información generada (tablas de literales y símbolos, y *tokens*) es proporcionada al usuario final.

Diseño e implementación

Respecto a las implementaciones, las etapas de manejo de información y salidas al usuario fueron las que permitieron mayor flexibilidad, pues para lograr esto se tuvo que trabajar con lenguaje C, mientras que, por otro lado, las expresiones regulares fueron planteadas con cierta sintaxis de Lex/Flex.

Con la finalidad de facilitar y generalizar el manejo de ciertas estructuras, las tablas de literales y símbolos fueron implementadas por medio de listas ligadas unidireccionales, es decir, listas en las que se pueden realizar recorridos en una única dirección. En particular, se propusieron dos casos de listas ligadas: uno para representar a tablas de literales y otro para representar a tablas de símbolos. La necesidad de dos casos basados en una misma estructura de datos se debió a los requerimientos descritos para esta componente del proyecto, en donde se menciona que las tablas de literales, tanto para constantes numéricas como para constantes cadenas, pueden contener elementos duplicados, mientras que las tablas de símbolos no pueden contener elementos duplicados. Para implementar una estructura de lista ligada que no permita el alojamiento de datos repetidos respecto a cierto campo se debe implementar una función de búsqueda sobre la estructura. En este caso, la función de búsqueda implementó un algoritmo de búsqueda secuencial; esto considerando que la búsqueda se realizaría sobre un conjunto de datos (respecto a un campo) no ordenado.

Al tratarse de listas ligadas unidireccionales, las operaciones de inserción de nodos fueron realizadas por medio de recorridos a lo largo de la lista mientras el nodo siguiente respecto a un nodo de referencia no fuera nulo, tal que al llegar al nodo que no cumpla con la condición, se inserte el nuevo nodo como el

siguiente del nodo de referencia actual. Se trata del algoritmo ordinario de inserción de nodos en listas ligadas unidireccionales.

El resto de estructuras de catálogos (para palabras reservadas y operadores relacionales) no fueron implementadas con estructuras de datos, sino que fueron emuladas por medio de funciones con condicionales, pues no se consideró necesario tener almacenada a dicha información de la misma manera que las tablas de literales y símbolos, considerando que en las primeras ya se conocía el rango de operación desde un inicio.

Particularidades, tablas de literales: Para este proyecto existieron dos tablas de literales, una de constantes cadenas y otra de constantes numéricas reales, formadas por dos campos: posición y dato.

```
typedef struct nodo_lit {
    int pos;    // Posicion
    char* dato; // Dato
    struct nodo_lit* next;
} nodo_lit;
```

Cuando Flex reconoce a un componente léxico, lo opera como una cadena de caracteres, así que aprovechando esta característica y considerando que, en esencia, las dos tablas de literales tienen la misma mecánica, se usó una estructura de nodo cuyos campos de *dato* y *posición* están dados por una cadena de caracteres y por un entero, respectivamente.
De esta manera, tanto las constantes numéricas reales como las constantes cadenas encontradas en el archivo fuente de interés son insertadas en los nodos como cadenas de caracteres (char*) de C.

Particulares, tabla de símbolos: La tabla de símbolos está formada por tres campos: posición, nombre del identificador y tipo.

```
typedef struct nodo_tabsim {
    int pos;    // Posición
    char* nombre_id;    // Nombre del identificador
    int tipo;   // Tipo de dato: Inicial es -1
    struct nodo_tabsim* next;
} nodo_tabsim;
```

Para implementar la lista ligada que representa a la tabla de símbolos se usó una estructura de nodo de 3 campos: *posición*, *nombre del identificador*, y *tipo*, con tipos de datos entero, cadena de caracteres (char*) y entero, respectivamente.
Para realizar la inserción de una identificador reconocido por el analizador léxico en la tabla de símbolos primero se aplica la función de búsqueda sobre la lista ligada de símbolos respecto al identificador en cuestión, tal que si el identificador ya existe en la lista, no se realiza la inserción, mientras que si no existe el identificador en algún nodo de la lista, se realiza la inserción en la última posición posible.

Tokens: A continuación, se describe la manera en que los *tokens* son generados para cada clase definida en este lenguaje. Es importante mencionar que los *tokens* tienen una estructura de tupla de dos componentes de la forma *(Clase, Valor).*

- Palabras reservadas (clase 0): Se generan *tokens* de la forma (0,V), donde V es la posición de la palabra reservada en el catálogo de palabras reservadas. El valor V es proporcionado como valor de retorno de una función de condicionales.

- Identificadores (clase 1): Se generan *tokens* de la forma (1,V), donde V es la posición del nombre de identificador en la tabla de símbolos. El valor de V es proporcionado como valor de retorno de la función de inserción en la tabla de símbolos.
- Constantes numéricas enteras (clase 2): Se generan *tokens* de la forma (2,V), donde V es el valor de la constante numérica entera en sí.
- Constantes numéricas reales (clase 3): Se generan *tokens* de la forma (3,V), donde V es la posición de la constante real en la tabla de literales de constantes reales. El valor de V es proporcionado como valor de retorno de la función de inserción en la tabla de literales de constantes reales.
- Constantes cadenas (clase 4): Se generan *tokens* de la forma (4,V), donde V es la posición de cadena constante en la tabla de literales de constantes cadenas. El valor de V es proporcionado como valor de retorno de la función de inserción en la tabla de literales de constantes cadenas.
- Símbolos especiales (clase 5): Se generan *tokens* de la forma (5,V), donde V es el símbolo especial en sí.
- Operadores aritméticos (clase 6): Se generan *tokens* de la forma (6,V) donde V es el operador aritmético en sí.
- Operadores relacionales (clase 7): Se generan *tokens* de la forma (7,V), donde V es la posición del operador relacional en el catálogo de operadores relacionales. El valor de V es proporcionado como valor de retorno de una función de condicionales.
- Operador de asignación (clase 8): Se generan *tokens* de la forma (8,V), donde V es siempre el operador de asignación en sí.

Indicaciones de ejecución

Para ejecutar el programa correctamente, en primer lugar, se parte del archivo con extensión **.l** que contiene a la implementación en Flex del analizador léxico. Para ello, en la terminal de Linux, se usa el siguiente comando, considerando que el nombre del archivo fue “analizador_L.l”.

```
flex analizador_L.l
```

Entonces, se generará en el directorio en que se realice la compilación un archivo llamado “lex.yy.c”, el cual es un archivo en lenguaje C. Para compilar dicho programa se usa el siguiente comando.

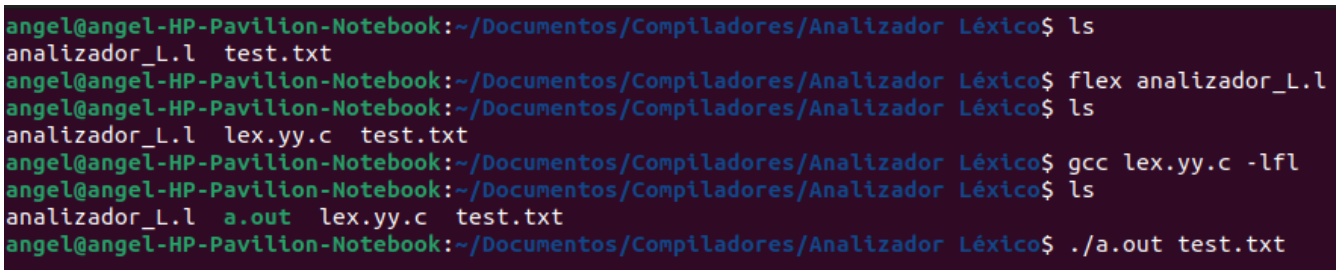
```
lex.yy.c -lfl
```

Una vez compilado el programa en C, se genera un archivo de salida (.out) llamado “a.out”, el cual puede ser ejecutado como cualquier programa escrito en C. Sin embargo, para obtener salidas funcionales es necesario tener en el directorio un archivo de texto que contenga componentes léxicos a evaluar. Dado un nombre genérico, el programa debe ser ejecutado con el siguiente comando desde la terminal.

```
./a.out <nombre de archivo de texto>
```

Al ejecutar el programa, en la terminal se observarán 5 secciones diferentes, correspondientes a la información de la tabla de símbolos, las tablas de literales, los *tokens* generados, y los errores encontrados en el archivo fuente.
A su vez, posterior a la ejecución del programa en sí, se generarán 5 archivos de texto adicionales en el directorio de operación, con los nombres “Tokens.txt”, “TablaLiteralesReales.txt”, “TablaLiteralesConstantes.txt”, “TablaSimbolos.txt” y “Errores.txt”. Estos archivos contienen a la

misma información desplegada en la terminal al concluir la ejecución, con la intención de permitir una posterior consulta a detalle de la información generada.



Ejemplo de ejecución

Conclusiones

En general, en esta primera parte del proyecto final de la asignatura de Compiladores se pudo implementar la fase de analizador léxico de un compilador por medio de la herramienta Flex para definir analizadores léxicos en C. El proyecto en sí me pareció bastante complicado, sobre todo respecto a la definición de las expresiones regulares para reconocer errores; en un inicio me pareció complicado definir las expresiones regulares para las clases descritas, pero fue mucho más complejo definir las expresiones para los errores debido a que se debían considerar situaciones que no eran explícitas dada la definición de cada clase.

Un aspecto que me agradó bastante de este proyecto fue que se tuvieron que retomar conceptos de asignaturas de semestres anteriores, según el plan de estudios, como Estructuras de Datos y Algoritmos y Lenguajes Formales.

En definitiva no fue un desarrollo trivial, pues me tomó bastante tiempo hacer los análisis correspondientes a las expresiones con tal de realizar una implementación robusta. Además, ciertos tecnicismos de C como el manejo de apuntadores para la definición de ciertas estructuras de datos representaron algunas dificultades, aunque no mayores a las expresiones regulares por sí mismas. No obstante, a pesar de las dificultades, considero que en esta primera entrega se ha desarrollado una implementación robusta y adecuada de acuerdo con lo solicitado, así que estoy bastante conforme con el producto de esta primera entrega.