Universidad Nacional Autónoma de México
Facultad de Ingeniería
Compiladores
Grupo 03

Implementación de un Analizador Léxico para el lenguaje definido en clase

Profesora: M.C. Laura Sandoval Montaño
Alumno: Zuriel Zárate García
Semestre 2023-1

Ciudad de México, México 2 de noviembre de 2022

1. Objetivos

- Elaborar un analizador léxico en lex/flex que reconozca los componentes léxicos pertenecientes a las clases descritas.

2. Introducción

Desde nuestros primeros semestres en la carrera de Ingeniería en Computación, nos mencionaron vagamente el concepto de "Compilador" y se nos dijo que era el encargado de traducir nuestras instrucciones, escritas en un lenguaje "humano" (lenguaje de programación), en un lenguaje que la máquina pudiese entender para poder ejecutar dichas instrucciones, es decir, puros unos y ceros. Ahora, sabemos que el proceso de compilación consta de varias fases: Análisis Léxico, Análisis Sintáctico y Análisis Semántico, Generación de código (máquina) y optimización de código.

Por lo anterior, en el presente trabajo, presentamos el diseño y desarrollo de un analizador léxico, correspondiente a la primera fase de análisis de un compilador, para un lenguaje de programación definido en clase.

2.1. Planteamiento del problema

Un autómata finito es una "máquina" capaz de distinguir cadenas con las que se puede hacer algo útil y rechazar cadenas que no son relevantes. Esto es un analizador léxico, es decir, una máquina capaz de distinguir las cadenas pertenecientes a un lenguaje. En este caso, dividiremos el reconocimiento en 9 clases con el criterio que se muestra a continuación:

Clase	Descripción
0	Palabras reservadas (ver tabla 2).
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 3).
8	Operador de asignación =

Cuadro 1: Componentes léxicos válidos para el analizador léxico a desarrollar

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

Cuadro 2: Catálogo de palabras reservadas

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

Cuadro 3: Catálogo de operadores relacionales

Una vez definido el lenguaje, las clases y la estructura de los catálogos, se vuelve un desafío programarlo. Afortunadamente, tenemos un generador de analizadores léxicos para C y C++: LEX/FLEX. Utilizando este generador, podemos definir el analizador utilizando expresiones regulares y aplicar reglas sobre las cadenas reconocidas. A continuación, se muestran las expresiones regulares utilizadas para cada clase:

Palabras reservadas

CASE “alternative”
LONG “big”
IF “evaluate”
ELSE “instead”
DOUBLE “large”
WHILE “loop”

DO “make”
INT “number”
DEFAULT “other”
FLOAT “real”
FOR “repeat”
SWITCH “select”
SHORT “small”
CONTINUE “step”
BREAK “stop”
CHAR “symbol”
RETURN “throw”

RESERVADAS {CASE}|{LONG}|{IF}|{ELSE}|{DOUBLE}|{WHILE}|{DO}|{INT}|
{DEFAULT}|{FLOAT}|{FOR}|{SWITCH}|{SHORT}|{CONTINUE}|{BREAK}|{CHAR}|{RETURN}

Identificadores

PESOS "$"
LETRA [a-zA-Z]
IDENTIFICADOR {PESOS}({LETRA})+

Constantes numéricas enteras

ENTERODEC [1-9]([0-9])*|(0)+
ENTEROCT [oO]([0-7])+

Constantes numéricas reales

PUNTO "."
REAL -?([1-9]([0-9])*){0,8}{PUNTO}[0-9]{1,4}

Constantes de tipo cadena

CADENA (\“[^”’]{2,100}\“)|(\’[^\n’]{1,1}\’)|(\“[’]\ ”)

Símbolos especiales

CORCHETEIZQ "["
CORCHETEDER "]"
PARIZQ "("
PARDER ")"
LLAVEIZQ "{"
LLAVEDER "}"
COMA ","
DOSPUNTOS ":"
PUNTOYCOMA ";"
ESPECIALES {CORCHETEIZQ}|{CORCHETEDER}|{PARIZQ}|{PARDER}|{LLAVEIZQ}|
{LLAVEDER}|{COMA}|{DOSPUNTOS}|{PUNTOYCOMA}

Operadores aritméticos

MAS “+”
MENOS “-”
POR “*”
DIAG “/”
ANTIDIAG “\\”
MOD “ %”
CIRCUNFLEJO “∧”

OPARI {MAS}|{MENOS}|{POR}|{DIAG}|{ANTIDIAG}|{MOD}|{CIRCUNFLEJO}

Operadores relacionales

MENQUE “<”
MAYQUE “>”
MENIG “<=”
MAYIG “>=”
IGUALIGUAL “==”
DIFERENTE “!=”

OPREL {MENQUE}|{MAYQUE}|{MENIG}|{MAYIG}|{IGUALIGUAL}|{DIFERENTE}

Operadores de asignación

IGUAL “=”

ASIGNACION {IGUAL}

3. Desarrollo

3.1. Análisis

En este caso, todas las etapas de desarrollo las llevé a cabo solo. Las tareas realizadas hasta ahora fueron de gran ayuda para comenzar la etapa de codificación e implementación. Algo importante que debo mencionar es que utilicé plantillas que hice en segundo semestre, durante la materia de Estructuras de Datos y Algorítmos I, para la estructura de datos utilizada, así como las funciones para realizar operaciones con dicha estructura. Podrá notarlo en el código del programa.

3.2. Diseño e implementación

3.2.1. Estructuras utilizadas

Con base en la plantilla que utilicé, traté de simplificar mucho el uso de estructuras. Para empezar, creé una estructura llamada **Componente**, que tiene los campos: Clase, Valor, Cadena y Tipo. A continuación, una breve descripción de estos campos.

- Clase: Entero que almacena, como tal, la clase a la que corresponde el componente.

- Valor: Esto puede ser la posición, para los Componentes de catálogos o tablas de símbolos o, tal cual, el valor numérico de la cadena (utilizado en constantes numéricas enteras, caracteres especiales y operadores aritméticos).
- Cadena: Almacena la cadena reconocida.
- Tipo: Para los identificadores, se especifica el tipo de dato con la posición de la palabra reservada (empleada en su declaración) en el catálogo de palabras reservadas. Esto sucede una vez que se han llenado los catálogos.

La estructura anterior es útil para almacenar tokens, palabras reservadas, operadores relacionales, identificadores, constantes reales o constantes cadenas en tablas diferentes sin ninguna complicación, además de que nos permite implementar la búsqueda de una forma relativamente sencilla.

Posteriormente, creamos una estructura llamada **Nodo**, que almacenará un componente y una referencia al nodo siguiente.

Finalmente, utilicé la estructura de una lista simplemente ligada para definir las tablas. Esta estructura tiene una referencia al nodo "cabeza" y otra referencia al nodo "cola" (primer nodo en la lista y último nodo en la lista). Recordemos que cada Nodo tiene una referencia al Nodo que le sigue. Esta estructura fue nombrada como **Tabla**.

3.2.2. Algorítmo de búsqueda

El algorítmo de búsqueda implementado es de búsqueda lineal. Por cuestiones de tiempo y simplicidad, decidí utilizarlo. A continuación se muestra una imagen del código utilizado, que también servirá para mostrar un poco el uso de las estructuras definidas.

```
Nodo *buscar(Tabla tabla, char* cad){
    Nodo *q = tabla.H;
    while(q != NULL){
        if(strcmp(q->info->cadena,cad) == 0){ //Si la cadena coincide
            return q;
        }
        q = q->sig;
    }
    printf("\n");
    return NULL;
}
```

Figura 1: Algorítmo de búsqueda implementado.

3.2.3. Inserción en las tablas

Primero que nada, se declararon todas las tablas que pueden existir en el análisis de forma global, de modo que en la sección de reglas puedan ser accedidas. Estas son creadas al inicio del programa con la función **CrearTabla()**.

Para todas las tablas, se utilizaron contadores inicializados en -1 y declarados de forma global. Cuando se encuentra un identificador, por ejemplo, se busca (utilizando yytext con el algorítmo antes mencionado). Si ya está en la tabla de símbolos, el contador no aumenta y no se coloca nada en la tabla, sólo se crea el Componente y se agrega a la lista de tokens (también declarada al inicio de forma global).

Si, por ejemplo, se encuentra una nueva cadena, se aumenta su contador y este contador se vuelve la posición de la cadena nueva en la tabla, es decir, su valor. Lo mismo ocurre con las constantes de tipo real.

Un caso interesante es la especificación del tipo de dato cuando se declara una variable. Dado que el tipo de dato se pone antes, se encuentra una palabra reservada. Cada que se encuentra una palabra reservada, se busca en su catálogo y se extrae su posición. Dicha posición se pasa a una variable global llamada **tipo**. Si lo que sigue es un identificador, se puede iniciar el tipo de dicho identificador con la última actualización de la variable mencionada.

En el caso de las constantes enteras, sólo se crea el token con su valor correspondiente (como cadena). Aquí es donde veo la ventaja de la forma de estructurar los datos que utilicé, ya que se tiene toda la información posible a la mano.

Los símbolos especiales, operadores aritméticos y el operador sólo generan el token (creando el componente y agregándolo a la lista de tokens).

Finalmente, cuando se trata de operadores relacionales y palabras reservadas, simplemente se busca el token pasando a "yytext" como parámetro de la función de búsqueda y la correspondiente tabla en donde se quiere buscar. Se extrae el valor (su posición en la tabla) y con eso se crea un nuevo componente que va a la lista de tokens.

3.2.4. README.md

Las instrucciones de instalación y ejecución del programa se muestran en el archivo README.md. Para más información, favor de consultarlo. Puede descargar el proyecto completo aquí: https://github.com/delionsgate/ZZG_ALEX.

4. Conclusiones

Durante este trabajo, pude reafirmar varios conceptos revisados en clase, así como el uso de expresiones regulares. Pude realizar la implementación del análisis léxico utilizando el generador de analizadores léxicos LEX/FLEX (que simplificó las cosas de una manera enorme), con un poco de contratiempos, pero con mucha satisfacción. Debo admitir que se me complicó un poco todo debido al tiempo que tenía sin programar en C. De repente ver mensajes de errores de segmentación y errores extraños en tiempo de ejecución me sacaron mucho de mi zona de confort. Sin embargo, al final pude sobrellevarlo y trabajar con gusto.

Personalmente, estoy muy satisfecho con este trabajo, ya que realmente me esforcé demasiado y puse empeño en su elaboración. Espero que esto pueda notarse en los resultados. Por lo anterior, me parece que los objetivos de este trabajo se cumplieron satisfactoriamente.

5. Referencias

Chi, R. I. G. (2019). Teoría de lenguajes y autómatas [Consultado el 31 de octubre de 2022. Recuperado de: https://issuu.com/ingrosychi/docs/analizador_lexico_con_afd]. *ISSUU*.

de la Cruz, M., & Ortega, A. (2007). Construcción de un analizador léxico para ALFA con LEX [Consultado el 31 de octubre de 2022. Recuperado de: http://arantxa.ii.uam.es/~mdlcruz/docencia/compiladores/2007_2008/lexico_07_08.pdf]. *Prácticas de Procesadores de Lenguaje*.

Pérez, E. S. (2022). Compiladores [Consultado el 1 de noviembre de 2022. Recuperado de: https://hopelchen.tecnm.mx/principal/sylabus/fpdb/recursos/r94256.PDF]. *UAA - Sistemas Electrónicos*.