Analizador Lexico

David Morales, Paúl Cadena,

***Abstract*—El presente artículo presentaremos la información detallada acerca de cómo funciona el análisis léxico en el entorno de compiladores, mediante el uso de gramáticas para poder comprender la separación del lenguaje léxico del semántico, entender los diagramas de transición para lo cual se detallaran en el siguiente articulo.**

***Keywords*—Analizador léxico, semántico, parser, scanner, compilador LaTeX, paper, template.**

I. Introducción

Un analizador léxico es un programa, el cual cumple diferentes funciones específicamente, reconocer un lenguaje, con sus caracteres de entrada, donde se encuentra la cadena a analizar, reconocer subcadenas que correspondan a símbolos del lenguaje y retornar los tokens correspondientes y sus atributos. Pero escribir analizadores léxicos eficientes "a mano" puede resultar una tarea tediosa y complicada, y para evitarla se han creado herramientas de software – los generadores de analizadores léxicos – que generan automáticamente un analizador léxico a partir de una especificación provista por el usuario, pero también se puede realizar mediante programación desde un determinado lenguaje.

En el presente trabajo se abordará temas sobre la construcción de un programa para realizar el analisis léxico, para ello empezaremos por definir el lenguaje, sus componentes del lenguaje y sus reglas de sintaxis, además del programa, sus funciones. Este analizador léxico lee caracteres del archivo de entrada, donde se encuentra la cadena a analizar, reconoce lexemas en un cuadro llamado secuencia ,retornando en otro cuadro los tokens , lexemas , la posicion en que fila y columna esta ubicado de la tabla de codigos . y para finalizar mostraremos si existio algun error en el proceso de analisis.

Este trabajo tiene el objetivo de dar a conocer la lógica y funcionamiento de un analizador léxico , es decir leer el ujo de caracteres de entrada y transformarlo en una secuencia de componentes lexicos que utilizara el analizador sintactico , además de permitir a nosotros, como alumnos, entender mejor este tema, y de este modo adquirir la idea de la implementación de un programa que ayude en su solución; y dejarlo listo para la siguiente fase de la compilación denominada: Análisis Sintáctico.

II. Conceptos Básicos

Para que el analizador lexico consiga el objetivo de dividir la entrada en partes, tiene que poder decidir por cada una de esas partes si es un componente separado y, en su caso, de que tipo. De forma natural, surge el concepto de categoria lexica, que es un tipo de simbolo elemental del lenguaje de programacion. Por ejemplo: identicadores, palabras clave, números enteros, etc.

Los componentes léxicos (en inglés, tokens) son los elementos de las categorías léxicas. Por ejemplo, en C, i es un componente léxico de la categoría identicador, 232 es un componente léxico de la categoría entero, etc. El analizador lléxico irá leyendo de la entrada y dividiéndola en:

- **Componentes Léxicos.-** En general, no basta con saber la categoría a la que pertenece un componente, en muchos casos es necesaria cierta información adicional. Por ejemplo, sería necesario conocer el valor de un entero o el nombre del identicador. Utilizamos los atributos de los componentes para guardar esta información.
  Un último concepto que nos será útil es el de lexema: la secuencia concreta de caracteres que corresponde a un componente léxico.
  Por ejemplo, en la sentencia altura=2; hay cuatro componentes lexicos, cada uno de ellos de una categoría léxica distinta.
- **Categorías Léxicas más usuales.-** Algunas familias de categorías léxicas típicas de los lenguajes de programación son:
  - **Palabras clave.-** Palabras con un signicado concreto en el lenguaje. Ejemplos de palabras clave en C son while, if, return. . . Cada palabra clave suele corresponder a una categoría léxica. Habitualmente, las palabras clave son reservadas. Si no lo son, el analizador léxico necesitara información del sintáctico para resolver la ambiguedad.
  - Operadores Símbolos que especifican operaciones aritméticas, lógicas, de cadena, etc. Ejemplos de operadores en C son +, *, /,
  - **Constantes numéricas.-** Literales que especifican valores numéricos enteros, en coma flotante, etc. Ejemplos de constantes numéricas en C son 928, 83.3E+2. . .
  - **Constantes de carácter o de cadena.-** Literales que especifican caracteres o cadenas de caracteres. Un ejemplo de literal de cadena en C es "una cadena"; ejemplos de literal de caracter son 'x', "...
  - **Símbolos especiales.-** Separadores, delimitadores, terminadores, etc. Ejemplos de estos símbolos en C son , , ;. . . Suelen pertenecer cada uno a una categoría léxica separada.
  - **Identicadores .-** Nombres de variables, nombres de función, nombres de tipos denidos por el usuario, etc. Ejemplos de identicadores en C son

M. Shell is with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332 USA e-mail: (see http://www.michaelshell.org/contact.html).

J. Doe and J. Doe are with Anonymous University.

Manuscript received April 19, 2005; revised January 11, 2007.

i, x10, valor$_l e\acute{i}do$.

A. GRAMÁTICAS REGULARES Y EXPRESIONES REGULARES

Una forma más compacta de representar las gramáticas regulares es con el uso de expresiones regulares. Las expresiones regulares hacen uso de tres operadores (asuma que dos expresiones e1 y e2 generan los lenguajes L1 y L2 respectivamente):

Concatenar. Definida como e1 e2=x y — x L1 y e y L2

Alternar. Denotada como — ó +, es la unión de lenguajes denotados por dos expresiones por lo que e1—e2 = x — x L1 ó x L2

Cerrar. Representada por los corchetes , denota la repetición de la expresión cero o más veces, por lo que e1 = x — x L1* donde L1*= Ui=0 L1i

B. ERRORES LEXICOGRAFICOS

Algunos tipos de errores pueden reconocerse a nivel léxico, pero el analizador léxico tiene una vista muy reducida y localizada del programa fuente.

La recuperación de errores se da en una de las formas de modo de pánico donde se eliminan caracteres sucesivos hasta lograr la sincronización con alguna palabra llave conocida y válida en el lenguaje.

- 1) Algunas otras acciones de recuperación de errores son:
- 2) Borrar los caracteres extraños.
- 3) Insertar los caracteres faltantes.
- 4) Reemplazar un carácter incorrecto por otro correcto.
- 5) Trasponer dos caracteres en posición equivocada

III. DEFINICIÓN DEL LENGUAJE: COMPONENTES LÉXICOS Y REGLAS DE SINTAXIS (DIAGRAMA DE ESTADOS DE LOS COMPONENTES LÉXICOS)

A. Componentes Léxicos

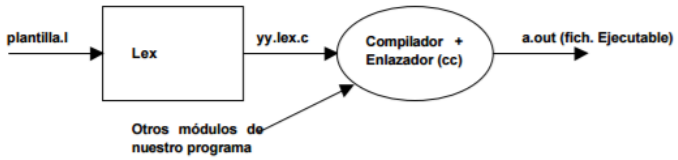
El lenguaje del programa ANALIZADOR LÉXICO tiene los siguientes tipos de componentes léxicos o tokens:

- Identificadores, que sólo son nombres de variables y están compuestos por una única letra minúscula de rango: a. . . z.
- **Constantes numéricas** de un sólo dígito, de rango: 0. . . 9.
- **Operadores:** +, -, *, / y
- **Símbolo de asignación:** = (igual).
- **Paréntesis:** “(”y “)”.
- **Separador de sentencias:** “;” (punto y coma).
- **Indicadores de principio y fin de bloque:** “” y “”.
- **Palabras reservadas**,int, float, bool, char, string, if, then, else, while, do, input, output, return.

IV. FLEX

Para la implementación del analizador léxico se hizo uso de la librería flex de C en un entorno Linux. Lex es una herramienta de los sistemas UNIX/Linux que nos va a permitir generar código C que luego podremos compilar y enlazar con nuestro programa. La principal característica de Lex es que nos va a permitir asociar acciones descritas en C, a la localización de las Expresiones Regulares que le hayamos definido. Para ello Lex se apoya en una plantilla que recibe como parámetro, y que deberemos diseñar con cuidado. Internamente Lex va a actuar como un autómata que localizará las expresiones regulares que le describamos, y una vez reconocida la cadena representada por dicha expresión regular, ejecutará el código asociado a esa regla. Externamente podemos ver a Lex como una caja negra con la siguiente estructura:

Fig. 1. Proceso de flex



La plantilla en la que Lex se va a apoyar para generar el código C, y donde nosotros deberemos describir toda la funcionalidad requerida, va a ser un fichero de texto plano con una estructura bien definida, donde iremos describiendo las expresiones regulares y las acciones asociadas a ella. La estructura de la plantilla es la siguiente:

Se compone de tres secciones con estructuras distintas y claramente delimitadas por una línea en la que lo único que aparece es el carácter doble porcentaje. Las secciones de ‘Declaraciones’ y la de ‘Procedimientos de Usuario’ son opcionales, mientras que la de ‘Reglas’ es obligatoria (aunque se encuentre vacía), con lo que tenemos que la plantilla más pequeña que podemos definir es: doble porcentaje.

A. La Sección de Declaraciones

En la sección de Declaraciones podemos encontrarnos con 3 tipos de declaraciones bien diferenciados:

- Un bloque donde le indicaremos al pre-procesador que lo que estamos definiendo queremos que aparezca ‘tal cual’ en el fichero C generado. Es un bloque de copia delimitado por las secuencias ‘porcentaje‘ y ‘porcentaje’ donde podemos indicar la inclusión de los ficheros de cabecera necesarios, o la declaración de variables globales, o declaraciones procedimientos descritos en la sección de Procedimientos de Usuario.
- - Un bloque de definición de ‘alias’, donde ‘pondremos nombre’ a algunas de las expresiones regulares utilizadas. En este bloque, aparecerá AL COMIENZO DE LA LÍNEA el nombre con el que bautizaremos a esa expresión regular y SEPARADO POR UN TABULADOR (al menos), indicaremos la definición de la expresión

regular. Para utilizar dichos nombres en vez de las expresiones regulares debemos escribirlos entre llaves.

- Un bloque de definición de las condiciones de arranque del autómata

Estos bloques pueden aparecer en cualquier orden, y pueden aparecer varios de ellos a lo largo de la sección de declaraciones. Recordemos que esta sección puede aparecer vacía.

Fig. 2. Sección de declaraciones(Proyecto)

```
delim [ ,\n,\t]
ws {delim}+
caracter ['']+({letra}|{digito}|{ws})*+['']
ignorar_comentarios "//"({letra}|" "|{caracter}|{digito})*|"/*"({letra}|" "|{caracter}|{digito})*
ignorar " "|\t|\n
digito [0-9]
letra [a-zA-Z]
booleano "true"|"false"
decimal {digito}+(\.{digito}+)?(E[+-]?{digito}+)?
string ["]+({letra}|" "|{caracter}|{digito})*+["]
```

B. La Sección de Reglas

AL COMIENZO DE LA LÍNEA se indica la expresión regular, seguida inmediatamente por uno o varios TABULADORES, hasta llegar al conjunto de acciones en C que deben ir encerrados en un bloque de llaves. A la hora de escribir las expresiones regulares podemos hacer uso de los acrónimos dados en la sección de Declaraciones, escribiéndolos entre llaves, y mezclándolos con la sintaxis general de las expresiones regulares. Si las acciones descritas queremos que aparezcan en varias líneas debido a su tamaño, debemos comenzar cada una de esas líneas con al menos un carácter de tabulación. Si queremos incorporar algún comentario en C en una o varias líneas debemos comenzar cada una de esas líneas con al menos un carácter de tabulación. Como normas para la identificación de expresiones regulares, Lex sigue las siguientes: - Siempre intenta encajar una expresión regular con la cadena más larga posible, - En caso de conflicto entre expresiones regulares (pueden aplicarse dos o más para una misma cadena de entrada), Lex, se guía por estricto orden de declaración de las reglas.

C. Secci[on de Procedimientos de Usuario

En la sección de Procedimientos de Usuario escribiremos en C sin ninguna restricción aquellos procedimientos que hayamos necesitado en la sección de Reglas. Todo lo que aparezca en esta sección será incorporado 'tal cual' al final del fichero yy.lex.l. No debemos olvidar como concepto de C, que si la implementación de los procedimientos se realiza 'después' de su invocación (en el caso de Lex, lo más probable es que se hayan invocado en la sección de reglas), debemos haberlos declarado previamente. Para ello no debemos olvidar declararlos en la sección de Declaraciones. Como función típica a ser descrita en una plantilla Lex, aparece el método principal (main). Si no se describe ningún método main Lex incorpora uno por defecto donde lo único que se hace es fijar el fichero de entrada como la entrada estándar e invocar a la herramienta para que comience su procesamiento

Fig. 3. Sección de Reglas(Proyecto)

```
%%
{ignorar_comentarios}               {;}
{ignorar}                   {;}
#include                    {printf("INCLUDE \n");}
float                       {printf("FLOAT \n");}
bool                        {printf("BOOL \n");}
char                        {printf("CHAR \n");}
string                      {printf("STRING \n");}
if                          {printf("IF \n");}
then                        {printf("THEN \n");}
else                        {printf("ELSE\n");}
while                       {printf("WHILE\n");}
do                          {printf("DO\n");}
input                       {printf("INPUT\n");}
output                      {printf("OUTPUT\n");}
{decimal}                   {printf("DECIMAL\n");}
return                      {printf("RETURN\n");}
":"                         {printf(":\n");}
","                         {printf(",\n");}
";"                         {printf(";\n");}
"("                         {printf("(\n");}
")"                         {printf(")\n");}
"["                         {printf("[\n");}
"]"                         {printf("]\n");}
"{"                         {printf("{\n");}
"}"                         {printf("}\n");}
"+"                         {printf("SUMA\n");}
"-"                         {printf("RESTA\n");}
"*"                         {printf("MULTIPLICACION\n");}
"/"                         {printf("DIVISION\n");}
">"                         {printf("MAYOR\n");}
"<"                         {printf("MENOR\n");}
"="                         {printf("ASIGNADOR\n");}
"!"                         {printf("NEGACION\n");}
"&"                         {printf("AND BINARIO\n ");}
"$"                         {printf("ASIGNADOR\n");}
"=="                        {printf("IGUAL \n");}
">="                        {printf("MAYOR IGUAL \n");}
"<="                        {printf("MENOR IGUAL\n");}
"!="                        {printf("DIFERNETE\n");}
"&&"                        {printf("AND LOGICO\n");}
"||"                        {printf("OR \n");}
main                        {printf("MAIN\n") ;}
[-]?{digito}+(E[+-]?{digito}+)?  {printf("ENTERO\n") ;}
{booleano}                  {printf("BOOLEANO\n");}
{caracter}                  {printf("CARACTER\n");}
{caracter}({caracter}|(digito))* {printf("ID\n");}
.                           {printf("ERROR en linea: %d \n",yylineno);}
{string}                    {printf("string\n");}
%%
```

V. Conclusiones

1) El analizador léxico divide la entrada en componentes léxicos.
2) Se pueden emplear las ideas de los analizadores léxicos para facilitar el tratamiento de cheros de texto.
3) Los componentes se agrupan en categorías léxicas.
4) Para reconocer los lenguajes asociados a las expresiones regulares empleamos autómatas de Estados finitos.
5) Se pueden construir los AFD directamente a partir de la expresión regular.
6) El tratamiento de errores en el nivel léxico es muy simple.

References

[1] . Viloria. *Una Herramienta para el Análisis Léxico: Lex* Available: https://www.infor.uva.es/ mluisa/talf/docs/aula/A3-A6.pdf

[2] . Gálvez, M. Mora "Compiladores",2005, España

Fig. 4. Sección deProcedimientos de Usuario(Proyecto)

```
%%

yyerror(char * msg){
printf("%s\n",msg);
}

int main(void){

yyin=fopen("entrada.txt","rt");

if(yyin==NULL){
printf("no se pudo abrir el archivo");
}else{

yylex();
}

}
```