Ensamblaje de un compilador: Diseño de un analizador léxico

Dario Vargas, Alexis Guerrero

Estudiantes de la Escuela Politécnica Nacional, de la carrera de Ingeniería de Sistemas

***Abstract*— El analizador léxico forma parte fundamental de la estructura de un compilador, y junto con el analizador semántico son un par básico para poder controlar y manejar errores que se puedan producir. El analizador léxico se basa en analizar palabra por palabra y ver si pertenece a una gramática específica. La construcción de este analizador se realizará mediante la herramienta informática Flex la cual implementa varios elementos de la teoría de compiladores tales como expresiones regulares, autómatas finitos deterministas y no deterministas. Los resultados esperados del analizador es poder ver la correspondencia de cada palabra o carácter a su definición de expresión regular y también su categoría sintáctica. La implementación del analizador lexical brindará al programador mayores destrezas al momento de generar código, ya que permite aplicar teoría no solo de las típicas estructuras de programación, sino de estructuras más complejas que necesitan mayor elaboración y entendimiento.**

***Palabras clave*—compilador, Flex, analizador léxico**

I. Introducción

Los distintos programas existentes en todo el mundo informático se han creado gracias a varios lenguajes de programación existentes, pero la creación de estos programas también implica usar otro programa llamado compilador para que pueda funcionar. Estos compiladores permiten que la máquina entienda lo que el programa pueda hacer y que el programador pueda ejecutar de manera correcta su código fuente. La creación de un programa que pueda validar otros programas se escucha un tanto complejo y muy laborioso, lo cierto es que efectivamente si es así, pero el diseñar un compilador enriquece de conocimiento al programador, porque le permite mejorar destrezas y talentos. Para el diseño de compiladores existen herramientas de ayuda que facilitan su creación, por tal razón esta práctica no es un tema del que todos huyan por su peculiar dificultad. Es así que la construcción del compilador se divide en tres partes principales y en esta ocasión se tratará y se pondrá en práctica la elaboración del analizador lexical, una de las bases más fuertes del compilador.

II. Marco teórico

Como se sabe un compilador es un programa de implementación y procedimientos complejos que incluye bastantes estructuras de programación tales como árboles, grafos, etc; o ciertos algoritmos especiales que permiten una funcionalidad más eficiente de dicho programa y obviamente una excelsa detección de errores en el mismo. Con respecto a esta última característica, es una de las más importantes, ya que principalmente define la base fundamental de un compilador.

El análisis de errores en primera instancia se forma por dos partes las cuales son el analizador lexicográfico y el analizador sintáctico, encontrados en el front end del compilador; aunque totalmente está formado por muchas más partes tales cómo optimizadores, selectores de instancias, localizadores de registro, y otras pocas más (estos dos últimos forman parte del back end y son los responsables de la relación entre el código generado a nivel de máquina y la ejecución del set de instrucciones del procesador que se está usando). Ante lo anterior es necesario aclarar que en esta parte solamente se tratará sobre el tema de análisis de errores lexicográficos, que es lo que primariamente constituye este proyecto.

Para empezar, sería bueno entender el significado literal de la palabra léxico, el cual según el diccionario de la RAE lo define como un conjunto de vocablos pertenecientes a un determinado lenguaje o a una determinada actividad. Efectivamente esta definición refleja la tarea esencial de un analizador léxico, la cual básicamente consiste en separar las distintas palabras o lexemas hallados en el código fuente escrito por el usuario y ver si cada una de estas cumple con las reglas preestablecidas por el mismo usuario, es decir la gramática que va tener dicho lenguaje. Es así que el analizador léxico se guiará por este

criterio de pertenencia a dicha gramática para poder devolver error o poder anunciar que las distintas palabras escritas están correctas y pertenecen al lenguaje fuente determinado por el usuario.

Claro está que este analizador léxico no se encarga para nada de revisar si los conjuntos de palabras analizadas pertenecen a lenguaje de máquina, ya que esta parte es tarea de otros componentes del compilador. Este analizador basa la veraz pertinencia de una palabra al lenguaje en las distintas categorías sintácticas (Ej: verbo, correr; verbo es la categoría sintáctica mientras que correr es la palabra analizada). El papel que juegan las categorías sintácticas es muy importante al momento de construir los reconocedores de palabras, ya que la estructura de cada categoría sintáctica se asignará en una determinada expresión regular.

Las expresiones regulares son estructuras generales impuestas por el diseñador del compilador que definen a un tipo de categoría sintáctica, es decir por ejemplo una categoría sintáctica "Número" estará formada por una sucesión de dígitos infinita, también por un punto si fuera el caso de ser un número decimal; pero dicha definición se la realiza de una manera más formal para que sea una expresión regular.

Número = [dígitos]$^{+}$ + ["."dígitos]?

La expresión anterior sería un ejemplo de expresión regular. Como se pudo observar las expresiones regulares están formadas principalmente de conjuntos de caracteres y en otros casos por caracteres específicos. La estructura de estas expresiones no es una estructura definitiva o globalmente estandarizada, es decir no se puede decir que una expresión regular esté mal o bien, sino que este criterio dependerá del lenguaje definido por el diseñador del compilador.

Las expresiones regulares utilizan ciertos símbolos tales como +,?,*, entre otros, en sus formulaciones. Estos símbolos son una especie de operadores que indican exclusión, opcionalidad, o la existencia de cero uno o más ocurrencias de un determinado carácter. La necesidad de colocar este tipo de restricciones es de vital importancia porque dichas restricciones son las que van a regir en el criterio de pertenencia al momento que el analizador léxico vaya leyendo cada palabra; el no definir correctamente una expresión regular de acuerdo al lenguaje fuente que se está usando podría ocasionar ambigüedades y también errores al momento posterior de compilar el programa.

Sustitución simple	.	Sustituye a un carácter simple de la línea de entrada (salvo *\n, \r, \f, \0 -newline, return, line feed, null,* respectivamente-)
Anclajes	^	Representa el principio de la línea de entrada *Ej.* ^A → representa líneas que empiezan con una A.
	$	Representa el final de la línea de entrada *Ej.* A$ → representa líneas que acaban con una A.
Conjunto de caracteres / Rangos	**[***expr***]**	Conjunto de caracteres admitidos. *Ej.* m[ab] → representa cadenas que contengan una m seguida de a o b. *Ej.* [02468] → representa cadenas que contengan algún dígito par.
	[^*expr***]**	Conjunto de caracteres no admitidos. *Ej.* m[^abc]→ representa cadenas que contengan una m seguida de un carácter que no sea ni a, ni b ni c. [^\n] → Todo menos el símbolo de nueva línea
	[...-...]	Rango de caracteres *Ej.* [a-zA-Z] →representa las letras del alfabeto. [.'¿!i;:\.\?] →Signos de puntuación. (Observar los signos "escapados") [a-zA-Z0-9\-] → Letras del alfabeto o números o el signo menos
Alternativa	|	Representa a cualquiera de las expresiones que se encuentran a ambos lados del mismo (como OR) *Ej.* A|B → Representa A, B
Opcionalidad	?	Equivale a la aparición de cero o una vez el ítem o carácter precedente. *Ej.* BA? → Representa B, BA
Repetición	*	Equivale a la repetición de cero o más veces el ítem o carácter precedente. *Ej.* BA* → Representa B, BA, BAA, BAAA, etc.
	+	Repetición de una o más veces el ítem o carácter precedente. *Ej.* BA+ → Representa BA, BAA, BAAA, etc. (No empareja con B)
	{n} \{n\}	Repetición n veces del carácter precedente *Ej.* A{3} → Representa AAA
	{n, } \{n, \}	Repetición al menos n veces del carácter precedente *Ej.* A{3,} → Representa AAA, AAAA, AAAAA, etc.
	{n, m} \{n, m\}	Repetición de al menos n veces pero no más de m del carácter precedente. *Ej.* A{3,4}→ Representa AAA, AAAA
Referencia	\d	Donde *d* representa cualquier dígito decimal simple (entre 1 y 9). Representa la expresión número "d-ésima" encerrada entre los caracteres ( ) empezando desde la izquierda. *Ej.* ([0-9])\1([0-9]) →Representa dos números iguales seguidos de otro

FIGURA 1. CARACTERES DE RESTRICCIONES PARA DEFINIR UNA EXPRESIÓN REGULAR

Estas expresiones regulares están ligadas directamente con los autómatas finitos deterministas y los autómatas finitos no deterministas. Estos conceptos pertenecen directamente a máquinas virtuales que analizan patrones y dependiendo de su entrada y estado de aceptación cambian a un siguiente estado o devuelven un error. La diferencia entre los dos autómatas mencionados, es que el primero reconoce una serie de patrones sin opción a tener ambigüedades, mientras que el segundo si acepta ambigüedades o transiciones delta (espacios vacíos). No se profundizará más en el tema de los autómatas ya que para el diseño del analizador léxico la herramienta usada incorpora autómatas de reconocimiento.

Posteriormente al proceso de reconocimiento de errores léxicos, se da el reconocimiento de errores semánticos, es decir los errores de concordancia en la oración o sentencia establecida. Luego de reconocer errores y optimizar código finalmente se puede enviar el código fuente transformado a lenguaje de máquina para que pueda ser interpretado por el procesador y de esta manera poder obtener los resultados escritos en el código fuente de algún programa escrito. Estos temas mencionados no pertenecen al área del diseño del analizador léxico.

III. DESARROLLO PRÁCTICO

Para poder diseñar el analizador léxico se utilizó la herramienta Flex, el cual es un programa para la generación de otros programas que realizan concordancia en patrones de texto. En relación con la teoría Flex viene a ser un programa que incluye el uso de expresiones regulares y autómatas para poder realizar reconocimiento de patrones. Este programa es muy servicial porque posee algunas ventajas tales como:

- Ser software de tipo libre.
- Tener funciones que faciliten el reconocimiento de patrones.
- Administrar archivos con una estructura de complejidad menor.

La herramienta Flex es para sistemas Linux, aunque también se puede ejecutar en Windows mediante la instalación de la plataforma correcta. En este caso para diseñar el analizador léxico se ha usado el sistema operativo Ubuntu que está basado en GNU/Linux, exactamente la versión 16.04.1 LTS y de Flex la versión 2.6

Para poder realizar el script en Flex en la primera sección se definió las expresiones regular de cada identificador de los datos tipo bool, string, char e int, así mismo se definió los conjuntos para los delimitadores, letras, dígitos y también se definió el conjunto de caracteres especiales. Además, también se declaró una variable **numLinea** para poder controlar en cual línea existirá error de alguna palabra que no tenga su estructura especificada. Se declaró también un puntero de archivo que generará un fiche de texto llamado **secuenciaTokens** a partir del fichero que se analice desde la terminal de Ubuntu.

```
#include<stdio.h>

FILE *fileOuput;
int numLinea=1;

%}

delim [ ,\n,\t]
ws {delim}*
letra [a-zA-Z]
digito [0-9]
charEsp [\,,\;,\:,\(,\),\[,\],\{,\},\+,\-,\*,\/,\<,\>,\=,\!,\&,\$]
id {letra}+({letra}|{digito})*
idInt [-]?{digito}+
idFloat {idInt}([\.]{digito}+)?
idBool (true|false)
idChar (\')+({letra}|{charEsp}|[ ])+(\')
idString (\")+({letra}|{charEsp}|{digito}|[ ]|{id})++(\")
comentParr (\/\*)+({letra}|{charEsp}|{digito}|[\#,\^,\%]|{ws}|{id})*+(\*\/)
comentLin (\/\/)+({letra}|{charEsp}|{digito}|{id}|[\#,\^,\%]|[ ,\t])*+(\n)

%%
```

Figura 2. Se puede observar las distintas declaraciones de expresiones regulares de cada categoría sintáctica y los conjuntos de los delimitadores, letras, dígitos y caracteres especiales. Al inicio se puede observar el comando para generar un archivo de salida donde se encontrará el respectivo resultado.

En la segunda división es necesario poner las reglas, esta parte se considera la más importante del fichero, porque si no se cumplen dichas reglas al momento de compilar, Flex retornará error, en este caso esta devolución de error es necesaria ser controlada.

```
bool fprintf(fileOuput,"%s\t es una palabra reservada Bool\n",yytext);
char fprintf(fileOuput,"%s\t es una palabra reservada Char\n",yytext);
string fprintf(fileOuput,"%s\t es una palabra reservada String\n",yytext);
if fprintf(fileOuput,"%s\t es una palabra reservada If\n",yytext);
then fprintf(fileOuput,"%s\t es una palabra reservada Then\n",yytext);
else fprintf(fileOuput,"%s\t es una palabra reservada Else\n",yytext);
while fprintf(fileOuput,"%s\t es una palabra reservada While\n",yytext);
do fprintf(fileOuput,"%s\t es una palabra reservada Do\n",yytext);
input fprintf(fileOuput,"%s\t es una palabra reservada Input\n",yytext);
output fprintf(fileOuput,"%s\t es una palabra reservada Output\n",yytext);
return fprintf(fileOuput,"%s\t es una palabra reservada Return\n",yytext);
"<" fprintf(fileOuput,"%s\t es el operando Menor que\n",yytext);
"<=" fprintf(fileOuput,"%s\t es el operando Menor e igual que\n",yytext);
">" fprintf(fileOuput,"%s\t es el operando Mayor que\n",yytext);
">=" fprintf(fileOuput,"%s\t es el operando Mayor e igual que\n",yytext);
"==" fprintf(fileOuput,"%s\t es el operando SI es igual\n",yytext);
"!=" fprintf(fileOuput,"%s\t es el operando NO es igual\n",yytext);
"~=" fprintf(fileOuput,"%s\t es el operando NO es igual\n",yytext);
"&&" fprintf(fileOuput,"%s\t es el operando Y\n",yytext);
"||" fprintf(fileOuput,"%s\t es el operando O\n",yytext);

\n {++numLinea;}
[ ]
{comentParr}
{comentLin}
{idInt} fprintf(fileOuput,"%s\t es un numero Entero\n",yytext);
{idFloat} fprintf(fileOuput,"%s\t es un numero Decimal\n",yytext);
{idBool} fprintf(fileOuput,"%s\t es un Boolean\n",yytext);
{id} fprintf(fileOuput,"%s\t es una Variable\n",yytext);
{idChar} fprintf(fileOuput,"%s\t es un Char\n",yytext);
{idString} fprintf(fileOuput,"%s\t es una (cadena) String\n",yytext);
{charEsp} fprintf(fileOuput,"%s\t es un Caracter Especial\n",yytext);
.     {
        ++numLinea;
        fprintf(fileOuput, "Error en la linea: %d    %s\n",numLinea, yytext );
        yyterminate();}

%%

main (int argc, char *argv[]) {
        yyin=fopen(argv[1],"r");
        fileOuput=fopen("secuenciaTokens.txt","w");
        yylex();
        fclose(fileOuput);
}
```

Figura 3. En la imagen se puede observar que el reconocimiento de las palabras clave se lista primero, porque Flex lee lo primero que se lista. Luego se puede observar las reglas pertinentes al reconocimiento de operadores y también la función que hace aumentar a la variable numLinea. Luego de las palabras clave se puede ver que se han listado los identificadores de elemento tales como cuales palabras son strings, char, int, espacios o caracteres especiales. Al final de la imagen se aprecia cómo se produce el archivo de salida **secuenciaTokens** que se produce en el mismo lugar donde se compilo el fichero de ejemplo.

*Untitled Document 1

```
int       es una Variable
main      es una Variable
(         es un Caracter Especial
)         es un Caracter Especial
{         es un Caracter Especial
int       es una Variable
research          es una Variable
=         es un Caracter Especial
65        es un numero Entero
;         es un Caracter Especial
float     es una Variable
rdf45xcv          es una Variable
=         es un Caracter Especial
45.231233123124554        es un numero Decimal
sahdksdhjksa      es una Variable
}         es un Caracter Especial
```

Figura 4. Este es un ejemplo de fichero de salida, donde se puede apreciar que cada carácter se va analizando y distribuyendo a su categoría sintáctica pertinente.

Para poder analizar un archivo de texto mediante Flex, lo primero es crearlo, ya que si no existe algún archivo de texto a analizar habrá errores.

IV. CONCLUSIONES

- La herramienta Flex realiza únicamente análisis léxico, más no análisis semántico; para realizar este tipo de tarea se necesita otra herramienta llamada Bison.
- La fundamentación correcta de un analizador lexical permite construir un compilador de mejor calidad.

- La mejor definición de una expresión regular de una determinada categoría sintáctica permite el mejor control de errores.
- El tiempo de compilación depende bastante del sistema operativo que se esté usando.
- Las palabras que se listan en primer lugar en la mayoría de ficheros de compilación, son las palabras reservadas. Esto se da en cualquier fichero.

V. BIBLIOGRAFÍA

[1] K. Cooper y L. Torczon, Engineering a compiler, Houston,Texas: Morgan Kaufman, 2012.