

Especificación y diseño del analizador léxico

Concha Vázquez Miguel
mconcha@ciencias.unam.mx

Flores Martínez Andrés
andresfm97@ciencias.unam.mx

Gladín García Ángel Iván
angelgladin@ciencias.unam.mx

Sánchez Pérez Pedro Juan Salvador
pedro_merolito@ciencias.unam.mx

Vázquez Salcedo Eduardo Eder
eder.vs@ciencias.unam.mx

14 de diciembre de 2018

A continuación se detalla el proceso seguido para la especificación del analizador léxico y su implementación con ayuda de Flex.

Índice

1 Gramática y símbolos terminales	2
2 Expresiones regulares	2
2.1 Palabras reservadas y caso general	2
2.2 Constantes de cadena y carácter	2
2.3 Identificadores	3
2.4 Ignorando espacios	4
2.5 Manejo de errores léxicos	4
2.6 Comentarios	4
2.7 Acciones léxicas	5
2.8 Orden de las expresiones regulares	5

1. Gramática y símbolos terminales

Usamos la misma gramática que nos fue provista, con ligeras modificaciones. Sin embargo, procuramos que el lenguaje generado por ésta fuera el mismo que en el caso original. En todo caso, para la gramática $G(\Gamma, \Sigma, S, \rightarrow)$, el conjunto Σ de los símbolos terminales está compuesto por:

Símbolos terminales de la gramática

▪ int	▪]	▪ while	▪	▪ *
▪ float	▪ (▪ for	▪ >=	▪ /
▪ double	▪)	▪ do	▪ <=	▪ %
▪ char	▪ ,	▪ switch	▪ <	▪ =
▪ func	▪ ;	▪ case:	▪ >	▪ CHARACTER
▪ void	▪ .	▪ default:	▪ !=	▪ CADENA
▪ struct	▪ true	▪ break	▪ ==	▪ ID
▪ {	▪ false	▪ print	▪ !	▪ INT
▪ }	▪ if	▪ return	▪ +	▪ DOUBLE
▪ [▪ else	▪ &&	▪ -	▪ FLOAT

2. Expresiones regulares

2.1. Palabras reservadas y caso general

En la gran mayoría de los casos —como los de las palabras reservadas—, la expresión regular para los símbolos terminales fue muy directa, simplemente especificando literalmente la sucesión de caracteres que conformaban los símbolos terminales y entrecomillándolos cuando fuera necesario para que no fueran confundidos por otros operadores por Flex y en su lugar los tomara literalmente. Notamos por ejemplo que la parte de `:` es compartida por `case:` y `default:`, así que optamos por definir un macro TP en la sección de macros de nuestro archivo de Flex y usarlo luego en la sección de las expresiones regulares como se ve a continuación:




```
TP :
case{TP}
default{TP}
```

Figura 1: Sección de macros y de expresiones regulares, respectivamente

2.2. Constantes de cadena y carácter

Luego, para las constantes de cadena y de carácter generamos las siguientes expresiones regulares:

- CADENA: Como queríamos cualquier secuencia de caracteres ASCII entrecomillados, bastó con identificar el rango de caracteres ASCII, para luego *saltarnos* el de la doble comilla `"`, y especificar que el *token* debería empezar con doble comilla y terminar con otra (escapándolas como se debía en Flex) para poder regresarlo como se esta clase léxica. Esto es, la expresión regular la definimos como:



```
cadena \"[\\x00-\\x21\\x23-\\x7F]*\"
```

Figura 2: Expresión regular para constantes de cadena.

- **CARACTER:** Hicimos lo mismo con las constantes de carácter, evitando que se pudiera utilizar la comilla simple dentro de la especificación como tal del carácter y dictaminando que deberían comenzar y terminar con una comilla simple. La diferencia fue que aquí no utilizamos la cerradura de *estrella de Kleene* pues solamente se permite un carácter:

```
caracter '[\x00-\x26\x28-\x7F]'
```

Figura 3: Expresión regular para constantes de carácter.

2.3. Identificadores

Para el caso de los identificadores, usamos la misma expresión que la que habíamos revisado en clase en un inicio y es la que se utiliza en el lenguaje de programación C. Decidimos limitar la cantidad máxima de caracteres que pueden conformar los identificadores a treinta y dos para que así pudieramos posteriormente reservar memoria en tiempo de compilación y no dinámicamente para el análisis semántico y la generación del código intermedio. La expresión regular fue:

```
ID [_a-zA-Z][_a-zA-Z0-9]{0,31}
```

Figura 4: Expresión regular identificadores.

Ergo, los identificadores deben comenzar con guión bajo o bien con alguna letra del alfabeto (minúscula o mayúscula) y pueden opcionalmente ir seguidos de cero hasta treinta y un caracteres alfanuméricos o guión bajo.

Constantes numéricas

Ahora, para las constantes numéricas, preferimos desde un inicio regresar *tokens* distintos dependiendo del tipo del número. Para este propósito, la expresión regular para los números enteros fue fácilmente definida:

```
int [0-9]+
```

Figura 5: Expresión regular para los números enteros, **INT**.

Para los números decimales (punto flotante y punto flotante de doble precisión), primero definimos subexpresiones regulares en la sección de macros que usamos posteriormente:

```
n1 [0-9]+\.[0-9]*
n2 \.[0-9]+
```

Figura 6: Definición de subexpresiones regulares para los números decimales. La primera indica que debe tener mínimamente un dígito, seguido de un punto decimal y luego opcionalmente más dígitos; la segunda que comienza con un punto decimal y le siguen uno o más dígitos

A partir de éstas fue posible definir los números decimales de tipo float y double. No obstante, antes definimos los *decimales* a secas, mismos que pueden ser una u otra de las subexpresiones definidas antes (se usó una disyunción):

```
decimal ({n1}|{n2})
double {decimal}|{int}
float {double}[Ff]
```

Figura 7: Definición de las expresiones regulares para los números decimales de simple y doble precisión. Nótese cómo los *floats* son básicamente como los *doubles*, pero terminan forzosamente con una f, ya sea mayúscula o minúscula. La semántica de cada tipo se da posteriormente en el análisis semántico.

2.4. Ignorando espacios

Después, colocamos en una clase de caracteres todos los que habrían de ser ignorados, como son:

- Espacios
- Tabulador
- Retorno de carro
- *Form feed*
- *backspace*

Así, esto se ve en la sección de expresiones regulares del archivo de Flex como:

```
[ \t\r\f\b]+ { increment(yytext); }
```

Figura 8: Ignorándolos. Nada más tenemos que incrementar el contador de la columna en la cantidad de los que leímos.

Aquí se pone en evidencia una función que implementamos en la sección de código en C para poder incrementar un contador llevado en una variable global para llevar así un control de la columna en que nos encontramos actualmente al leer el archivo de entrada sobre el que se lleva a cabo el análisis léxico: la función `increment`:

```
void increment(char *cadena) {  
    int size = (int)strlen(cadena);  
    col += size;  
}
```

Figura 9: Especificación de la función que incrementa el contador global de la columna a partir de la longitud de la cadena recibida. al momento de usarla, le pasamos `yytext`. Esto se hace con prácticamente con todos los *tokens* reconocidos.

Estrechamente relacionado con el punto anterior, fue trascendental reiniciar el contador de la columna a uno cada vez leíamos un salto de línea como se aprecia a continuación:

```
\n { col = 1; }
```

Figura 10: Reiniciando el contador de la columna cada vez que se lee un salto de línea.

2.5. Manejo de errores léxicos

Ahora, para manejar los errores léxicos, bastó con *atrapar* todo lo demás con un `.` y hacer que en dado caso se imprimiera con ayuda de una función de error `lexical_error` en un archivo de errores y en consola la línea, columna y carácter que ocasionaba el error:

```
. { increment(yytext);  
    lexical_error(yylineno, col, yytext[0]);  
    exit(1);  
}  
  
void lexical_error(int line, int column, char c) {  
    // No importa el número de parámetros de escritura en donde se escribirán los errores léxicos.  
    FILE *f = fopen("lexical_errors.txt", "w");  
    printf("Ocurrió un error léxico: \nLínea: %d\nColumna: %d\nCarácter que lo ocasiona: %c\n\n", line, column, c);  
    fprintf(f, "Ocurrió un error léxico: \nLínea: %d\nColumna: %d\nCarácter que lo ocasiona: %c\n\n", line, column, c);  
    fclose(f);  
}
```

Figura 11: Atrapando los errores léxicos y definición de la función para la impresión de errores; no se implementa recuperación de errores en ningún punto del proyecto como fue discutido.

2.6. Comentarios

Finalmente, fue necesario implementar la funcionalidad de detectar los comentarios que deberían empezar con `/*` y terminar con `*/` utilizando estados; más aún, logamos el reporte de errores en caso de dejar comentarios sin cerrar en el archivo.

```

/* Estado exclusivo para manejar los comentarios. */
%x comentario

%%

"/*"          { BEGIN(comentario); }
<comentario>[^"*/"]* { increment(yytext); }
<comentario>"*/" { increment(yytext);
                  BEGIN(INITIAL); }
<comentario><<EOF>> { printf("%s\n", "Error léxico: faltó cerrar el comentario.");
                    exit(1); }

```

Figura 12: Manejo de comentarios.

Vemos así como se define un estado exclusivo para el manejo de comentario, al que se entra luego de leer `/*`. Al leer cualquier cosa que no sea `*/` dentro de este estado, nada más incrementamos el contador de la columna y seguimos dentro del mismo estado. Sin embargo, si leemos `*/`, quiere decir que el comentario se ha cerrado y regresamos el estado inicial. Por último, en caso de estar en estado de comentario y leer el fin de archivo, entonces reportamos el error léxico correspondiente.

2.7. Acciones léxicas

Tuvimos que definir las acciones léxicas para dar el tipo y valor a los *tokens* de constantes de carácter y de cadena, así como también copiar el valor en cadena de los números leídos a los *tokens* que serían retornados. Para todos, el resultado fue:

```

{character} { increment(yytext); yyval.car.type = 3; yyval.car.val = yytext[0]; return CHARACTER; }
{cadena}   { increment(yytext); strcpy(yyval.cad.val, yytext); return CADENA; }
{ID}       { increment(yytext); strcpy(yyval.id, yytext); return ID; }
{int}      { increment(yytext);
            yyval.num.type = 0;
            strcpy(yyval.num.val, yytext);
            return INT; }
{double}   { increment(yytext);
            yyval.num.type = 2;
            strcpy(yyval.num.val, yytext);
            return DOUBLE; }
{float}    { increment(yytext);
            yyval.num.type = 1;
            strcpy(yyval.num.val, yytext);
            return FLOAT; }

```

Figura 13: Parte de la sección de expresiones regulares en donde se tuvieron que definir acciones léxicas.

2.8. Orden de las expresiones regulares

Evidentemente, tuvimos que tener sumo cuidado en la especificación de todas las expresiones regulares para poder detectar todos los *tokens* adecuadamente. Por ejemplo, tuvimos atención de colocar `<=` antes de `<` pues de lo contrario simplemente regresaríamos directamente `<` en lugar de percatarse de que viene en conjunto con `=`. La importancia del orden la tuvimos sobre todo con los operadores, pues muchos tienen prefijos comunes. El orden resultante fue:

```

"&&"      { increment(yytext); return AND; }
"||"      { increment(yytext); return OR; }
">="      { increment(yytext); return GEQ; }
"<="      { increment(yytext); return LEQ; }
"<"       { increment(yytext); return LT; }
">"       { increment(yytext); return GT; }
"!="      { increment(yytext); return NEQ; }
"=="      { increment(yytext); return EQ; }
"!"       { increment(yytext); return NOT; }
"+"       { increment(yytext); return PLUS; }
"-"       { increment(yytext); return MINUS; }
"*"       { increment(yytext); return PROD; }
"/"       { increment(yytext); return DIV; }
"%"       { increment(yytext); return MOD; }
"="       { increment(yytext); return ASSIG; }

```

Figura 14: Orden en las expresiones regulares para los operadores.