Práctica 1 - Procesadores de Lenguajes

Grupo 7
Carmen Toribio Pérez, 22M009
Sergio Gil Atienza, 22M046
María Moronta Carrión, 22M111

1 Introducción

La primera entrega de esta práctica consiste en la implementación de un Analizador Léxico y una Tabla de Símbolos. Para ello, hemos seguido los siguientes pasos:

1. **Identificar los Tokens** del lenguaje fuente, teniendo en cuenta las especificaciones de nuestro grupo.
2. Construir la **Gramática Regular** que los genera.
3. Diseñar el **Autómata Finito Determinista** equivalente a la gramática. Hemos realizado una representación de este a través de un **diagrama de estados**.
4. Añadir las **Acciones Semánticas**, asociadas a cada una de las transiciones del diagrama de estados.
5. Estudiar los posibles **casos de error**, para poder manejar correctamente la detección y el reporte de errores léxicos.

Por otro lado, hemos comenzado con la implementación de la **Tabla de Símbolos**, concretamente con el diseño de su estructura y su organización.

Finalmente, presentamos una serie de casos de prueba que demuestran el correcto funcionamiento del Procesador, así como su capacidad para manejar errores léxicos.

Como integrantes del **grupo 7**, hemos tenido que cumplir con las siguientes especificaciones:

- Sentencias: Sentencia repetitiva (for)
- Operadores especiales: Asignación con suma (+=)
- Técnicas de Análisis Sintáctico: Descendente Recursivo
- Comentarios: Comentario de bloque (/* */)
- Cadenas: Con comillas simples (' ')

Hemos decidido usar C++ como lenguaje de programación porque la mayor parte de la infraestructura de compiladores está escrita en C o en C++, incluyendo el proyecto LLVM (Low Level Virtual Machine). También hemos tenido en cuenta su flexibilidad y potencia, junto a la amplia variedad de utilidades que tiene su librería estándar. En comparación con lenguajes como Java o JavaScript, C++ ofrece mayor eficiencia y control sobre los recursos del sistema, algo crucial en proyectos como este (que requieren optimización de bajo nivel). Además, consideramos este proyecto como una gran oportunidad para practicar y aprender un lenguaje completamente distinto a los que estamos acostumbrados.

2 Tokens

El primer paso a la hora de construir un Analizador Léxico es la identificación de los tokens. Para hacer la lista nos hemos basado en la actividad práctica de la plataforma Draco. Hemos decidido utilizar el mismo formato en tablas con tal de facilitar su legibilidad.
Tokens obligatorios:

Elemento	Código de Token	Atributo
boolean	bool	-
for	for	-
function	fn	-
if	if	-
input	in	-
int	int	-
output	out	-
return	ret	-
string	str	-
var	var	-
void	void	-
constante entera	cint	Número
Cadena (')	cstr	Cadena ("c*")
Identificador	id	Número (posición en la TS)
+=	cumass	-
=	ass	-
,	com	-
;	scol	-
(	po	-
)	pc	-
{	cbo	-
}	cbc	-

Tokens de operadores aritméticos, lógicos y relcionales:

Grupo de Opciones	Código de Token	Atributo
Grupo Operadores Aritméticos: Suma (+)	sum	-
Grupo Operadores Aritméticos: Resta (-)	sub	-
Grupo Operadores Lógicos: Y lógico (&&)	and	-
Grupo Operadores Lógicos: O lógico (||)	or	-
Grupo Operadores Relacionales: Menor (<)	ls	-
Grupo Operadores Relacionales: Mayor (>)	gr	-

Tokens opcionales:

Grupo de Opciones	Código de Token	Atributo
Menos Unario (-)	sub	-
Más Unario (+)	sum	-
false	cap	-
true	nocap	-
EOF	eof	-

Por tanto, los siguientes tipos de expresiones no serán identificados como tokens: los delimitadores (como los espacios en blanco o las tabulaciones), los comentarios de bloque (/* */) o los saltos de línea (\n).

3 Gramática Regular

En esta sección, describimos la Gramática Regular (gramática de tipo 3 según la jerarquía de Chomsky) que hemos diseñado para identificar y generar los tokens del lenguaje fuente.

Símbolos no terminales

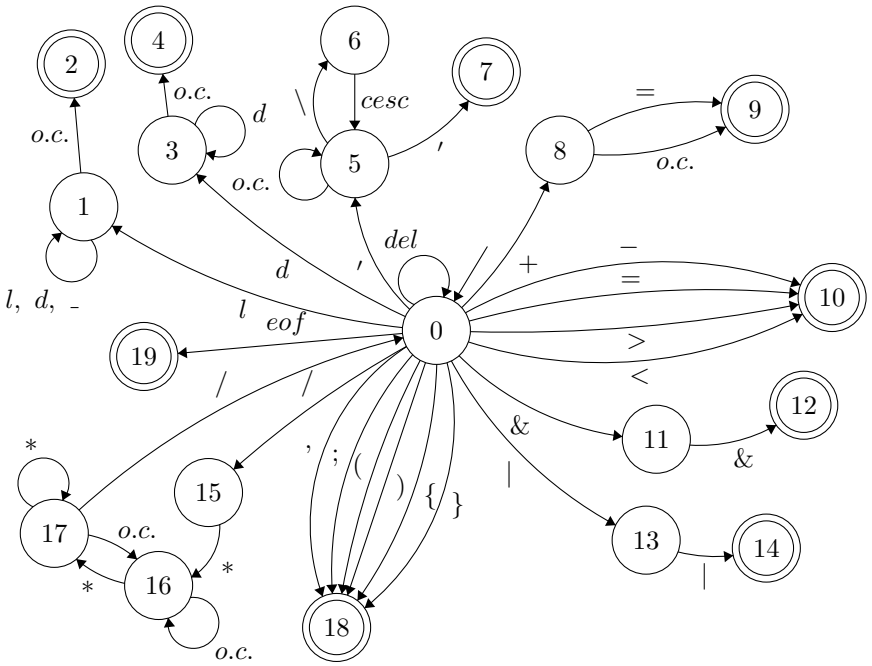
d := 0...9
l := a..z, A...Z
c_1 := espacio o cualquier carácter imprimible menos \
$cesc$:= carácter escapable (', 0, n, a, t, v, f, r, \)
c_2 := cualquier carácter menos * y eof
c_3 := cualquier carácter menos *, / y eof

Gramática Regular

S → del S | lA | dB | 'C | +D| - | = | > | < | &E | |F | /G | } | { |) | (| ; | , | eof
A → lA | dA | _A | λ
B → dB | λ
C → c_1C | \C' | '
C' → $cesc$C
D → = | λ
E → &
F → |
G → *H
H → c_2H | *I
I → /S | c_3H | *I

4 Autómata Finito Determinista

Una vez definida la gramática regular, el siguiente paso es construir el Autómata Finito Determinista (AFD) correspondiente. A continuación, presentamos su diseño, incluyendo las transiciones entre estados.



5 Acciones semánticas

A lo largo de esta sección, describimos las acciones semánticas que hemos añadido a las transiciones del Autómata Finito Determinista. Estas acciones permiten que se lleven a cabo acciones como la lectura, la identificación de los tokens o, en su lugar, una correcta gestión de los errores léxicos. A continuación, detallamos las operaciones utilizadas y, por legibilidad en el AFD, las transiciones en las que se realiza cada una:

Leer:

```
car := leer();          // Todas las transiciones salvo 1:2, 3:4, 6:5, 8:9 con un carácter
                        // distinto de '=', y 0:19
car := leerCesc();      // Transición 6:5. Leído un caracter de escape, devuelve el carácter
                        // correspondiente (por ejemplo un 'n' devuelve un eol)
```

Concatenar:

```
lex := car;             // Transición 0:1
lex := ∅;               // Transición 0:5
lex := lex ⊕ car;       // Transiciones 1:1, 5:5 y 6:5
```

Contador:

```
// Utilizado para contar el número de caracteres de la cadena
cont := 0;              // Transición 0:5. Inicializa el contador a 0
cont := cont + 1;       // Transiciones 5:5 y 6:5
```

Calcular valor entero:

```
// La función val(car) devuelve el valor entero del dígito correspondiente al carácter car
num := val(car);        // Transición 0:3
num := num * 10 + val(car);   // Transición 3:3
```

Generar token:

Cadenas y enteros

```
G1  if (cont > 64)                          // Transición 5:7
        then error (COD_ERROR_STRLEN, lex)
    else Gen_token(cstr, lex)
G2  Gen_token(cint, num)                    // Transición 3:4
```

Delimitadores y operadores de control

```
G3  Gen_token(cumass, -)                    // Transición 8:9 con el carácter '='
G4  Gen_token(ass, -)                       // Transición 0:10 con el carácter '='
G5  Gen_token(com, -)                       // Transición 0:18 con el carácter ','
G6  Gen_token(scol, -)                      // Transición 0:18 con el carácter ';'
G7  Gen_token(po, -)                        // Transición 0:18 con el carácter '('
G8  Gen_token(pc, -)                        // Transición 0:18 con el carácter ')'
G9  Gen_token(cbo, -)                       // Transición 0:18 con el carácter '{'
G10 Gen_token(cbc, -)                       // Transición 0:18 con el carácter '}'
```

Operadores Aritméticos, Lógicos y Relacionales

```
G11 Gen_token(sum, -)                       // Transición 8:9 con un carácter distinto de '='
G12 Gen_token(sub, -)                       // Transición 0:10 con el carácter '-'
G13 Gen_token(and, -)                       // Transición 11:12
G14 Gen_token(or, -)                        // Transición 13:14
G15 Gen_token(ls, -)                        // Transición 0:10 con el carácter '<'
G16 Gen_token(gr, -)                        // Transición 0:10 con el carácter '>'
```

Fin de fichero (EOF)

```
G17 Gen_token(eof, -)                       // Transición 0:19
```

Identificadores y palabras reservadas

```
G18   type := GetTokenCode(lex);*                    // Transición 1:2
      if type = id then
            pos := SearchST(lex);**
            if pos = NULL then
                  pos := AddID(lex);***
            Gen_Token(type, pos);
      else
            Gen_Token(type, -);
```

* La función GetTokenCode(lex) devuelve el código de token de la palabra reservada con la que coincida lex, o el código de ID si no es palabra reservada

** La función SearchST(lex) devuelve la posición del identificador lex en la tabla de símbolos, o NULL si no ha sido insertado aún

*** La función AddID(lex) inserta el identificador lex en la tabla de símbolos y devuelve su posición

6 Gestión de errores

Los casos de error posibles son todas aquellas transiciones no recogidas por nuestro AFD. En estos casos, se generará un error léxico y se informará al usuario de la existencia de un error en el código fuente, junto con el número de línea y columna en la que se ha detectado el error. A continuación, detallamos los errores que hemos identificado y los mensajes de error correspondientes a cada uno:

- **Caracteres no reconocidos.** En caso de que el carácter leído no coincida con ninguna transición del AFD, de manera que no pueda transitar desde el estado 0, se generará un error léxico cuyo mensaje de error es: (línea:columna) ERROR: Carácter inesperado al buscar el siguiente símbolo ("caracter inesperado", U+20AC).
- **Cadena de caracteres demasiado larga.** Si la longitud de la cadena supera los 64 caracteres, se generará un error léxico.
- **Comentario de bloque no cerrado.** Si se detecta un comentario de bloque sin cerrar, se generará un error léxico.
- **Cadena de caracteres no cerrada.** Si se detecta una cadena de caracteres sin cerrar, se generará un error léxico.

7 Tabla de Símbolos

Dado que en esta parte de la práctica solo se debe implementar el analizador léxico, nuestra tabla de símbolos es única y global. Por el momento, simplemente almacena los identificadores que el analizador léxico encuentre, y carecerá de atributos más allá del nombre del propio identificador.

8 Funcionamiento del programa

Breve explicación de lo que hace nuestro programa.

9 Conclusión(?)

Esto te lo dejo a ti, María. Por tener un parrafillo de cierre i guess.