Diseño de Compiladores I

Trabajo Práctico N° 1 y 2

2017

Grupo N°15

Integrantes

Federico Dominguez fedemillo.93@gmail.com
Brian López Muñoz lopezmbrian@gmail.com
Tomás Juárez tomasjuarez.exa@gmail.com

Docente Asignado

Prof. Dr. José Massa

Introducción

El objetivo de este trabajo consiste en diseñar e implementar un analizador léxico (*lexer*) y generar un analizador sintáctico (*parser*) a partir de una especificación de la gramática aceptada. Para llevar a cabo estos objetivos es necesario elegir un lenguaje de programación y un generador en particular, el cual en nuestro caso es Java y YACC, respectivamente.

A lo largo del informe se especificarán las decisiones relativas al diseño e implementación de los componentes, así como discusiones respecto a las soluciones propuestas.

Analizador Léxico

El analizador léxico es una de las partes fundamentales del compilador, el cual consiste en la materialización de un autómata para aceptar *tokens* desde el código fuente. Este autómata debe ser finito y determinístico y debe expresar las expresiones regulares para decidir si una cadena de texto debe ser reconocida o no dentro del lenguaje. Para ello se realizó un diagrama de transición de estados en donde el conjunto de estados y transiciones entre éstos decanta en una matriz. Esta matriz es, en efecto, la representación del autómata [figura 1] en donde no sólo se representan las transiciones de estados sino también las acciones semánticas involucradas [tabla 1].

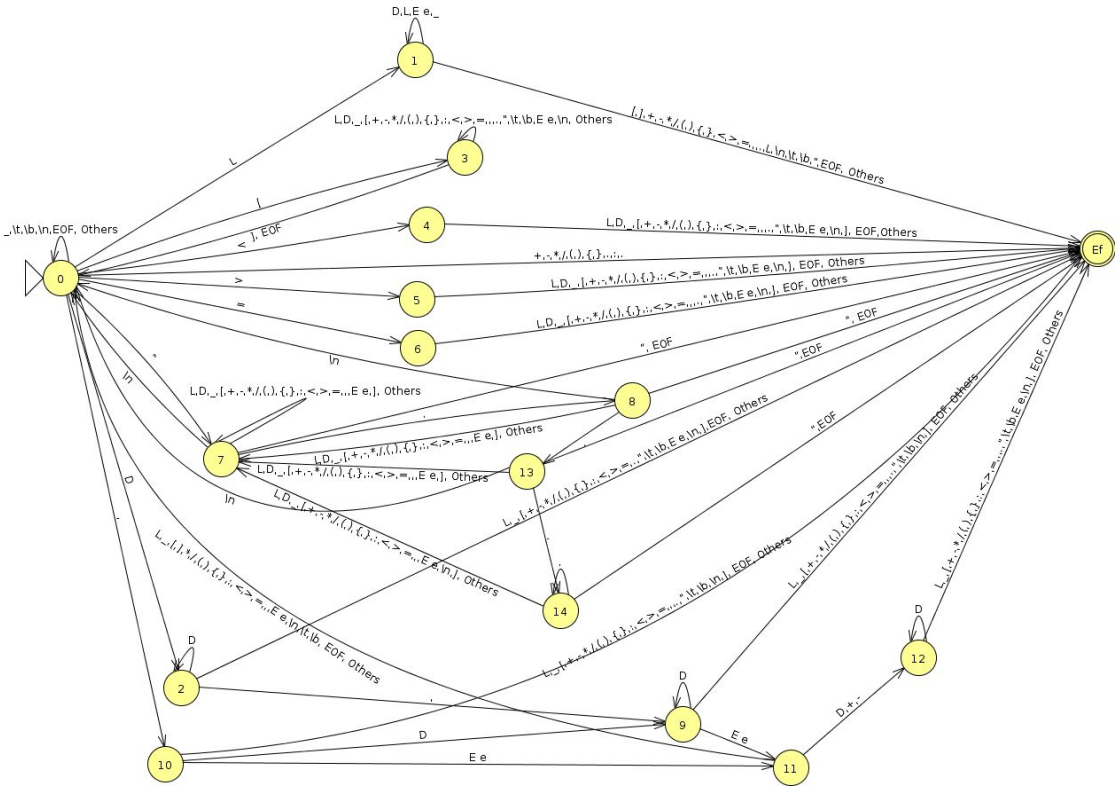

Figura 1: diagrama de transición de estados.

	L-{E,e}	D	_	[	]	+ -	* / ( ) :{}	<	>	=	,	.	"	\t \b	E e	\n	others	EOF
0	1	2	0/AS12	3/AS11	0/AS12	EF/AS4	EF/AS4	4	5	6	10	EF/AS4	7	0	1	0	0/AS12	0
1	1	1	1	EF/AS1	EF/AS1	EF/AS1	EF/AS1	EF/AS1	EF/AS1	EF/AS1	EF/AS1	EF/AS1	EF/AS1	EF/AS1	1	EF/AS1	EF/AS12	EF/AS12
2	EF/AS2	2	EF/AS2	EF/AS2	EF/AS2	EF/AS2	EF/AS2	EF/AS2	EF/AS2	EF/AS2	9	EF/AS2	EF/AS2	EF/AS2	EF/AS2	EF/AS2	EF/AS2	EF/AS2
3	3	3	3	3	0	3	3	3	3	3	3	3	3	3	3	3/AS16	3	0/AS15
4	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS5	EF/AS6	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11
5	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS7	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11
6	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS8	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11
7	7	7	7	7	7	7	7	7	7	7	7	8	EF/AS10	7	7	0/AS12	7	EF/AS14
8	7	7	7	7	7	7	7	7	7	7	7	13	EF/AS10	7	7	0/AS12	7	EF/AS14
9	EF/AS3	9	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	11	EF/AS3	EF/AS3	EF/AS3
10	EF/AS11	9	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	EF/AS11	11	EF/AS11	EF/AS11	EF/AS11
11	EF/AS12	12	EF/AS12	EF/AS12	EF/AS12	12	EF/AS12	EF/AS12	EF/AS12	EF/AS12	EF/AS12	EF/AS12	EF/AS12	EF/AS12	EF/AS12	EF/AS12	EF/AS12	EF/AS12
12	EF/AS3	12	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3	EF/AS3
13	7	7	7	7	7	7	7	7	7	7	7	14	EF/AS10	7	7	0/AS12	7	EF/AS14
14	7	7	7	7	7	7	7	7	7	7	7	14	EF/AS10	7	7	7/AS13	7	EF/AS14
EF	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabla 1: Matriz de transiciones y acciones semánticas.

En la siguiente tabla [tabla 2] se describen las acciones semánticas.

AS	Descripción
AS1	Elimina el último caracter del buffer. En caso de que supere 15 caracteres trunca el lexema y arroja Warning. Genera el token identificador o bien de una palabra reservada del lenguaje.
AS2	Elimina el último caracter del buffer. Valida los límites de números largos sin signo. Si está fuera de rango descarta el buffer, pero si se encuentra en rango genera el token de una constante de tipo ULONG.
AS3	Elimina el último caracter del buffer. Valida los límites de números flotantes. Si está fuera de rango descarta el buffer, pero si se encuentra en rango genera el token de una constante de tipo FLOAT.
AS4	Genera el token con el código ASCII de un operador o símbolo simple de la siguiente lista: '+', '-', ' *', '/', '(', ')',' .', ':','{','}',',','<','>'.
AS5	Genera un token de comparador '<>' (distinto).
AS6	Genera un token de comparador '<=' (menor o igual que).
AS7	Genera un token de comparador '>=' (mayor o igual que).
AS8	Genera un token '==' (igual que).
AS9	Elimina el último caracter del buffer. Genera un token Asignación '='.
AS10	Genera un token de string y elimina las comillas del lexema.
AS11	Elimina el último caracter del buffer y genera el token con el código *ASCII* de un operador o símbolo simple.
AS12	Elimina el primer símbolo del lexema y arroja Warning de símbolo inválido.
AS13	Elimina los últimos cuatro caracteres del buffer que se corresponden con los puntos suspensivos y el salto de línea de los strings multilíneas.
AS14	Arroja warning por falta de cierre de string y genera un token de tipo string.
AS15	Arroja warning por falta de cierre de comentario.
AS16	Suma 1 al número de línea actual en el lexer cuando se encuentra con un salto de línea '\n'.

Tabla 2: descripción de las acciones semánticas utilizadas.

Temas Asignados

En este apartado se listan los temas particulares asignados al grupo para facilitar la corrección al docente asignado. Por otro lado, también fueron implementados los temas comunes a todos los grupos, tal como se espera en la cátedra.

1. **Enteros largos sin signo**: Constantes enteras con valores entre 0 y $2^{32} - 1$. Se debe incorporar a la lista de palabras reservadas la palabra **ULONG**.
2. **Flotantes**: Números reales con signo y parte exponencial. El exponente comienza con la letra E (mayúscula o minúscula) y puede tener signo. La ausencia de signo

implica positivo. La parte exponencial puede estar ausente. El símbolo decimal es la coma “,”. Se debe incorporar a la lista de palabras reservadas la palabra **FLOAT**.

3. Incorporar a la lista de palabras reservadas las palabras **SWITCH** y **CASE.** Incorporar las llaves **{** y **}** como caracteres válidos.
4. Incorporar a la lista de palabras reservadas la palabra **LET**.
5. **Conversiones explícitas**: con lo cual se deberá incorporar a las anteriores : **UL_F**.
6. **Comentarios multilínea**: comentarios que comienzan con “[” y terminan con “]” (estos comentarios pueden ocupar más de una línea).
7. **Cadenas multilínea**: cadenas de caracteres que comienzan con y terminan con “ ”” . Estas cadenas pueden ocupar más de una línea, y en dicho caso, al final de cada línea, excepto la última, deben aparecer 3 puntos suspensivos **“ ... ”**. (En la Tabla de símbolos se guardará la cadena sin los puntos suspensivos, y sin el salto de línea).

Diseño e Implementación

El analizador léxico o *lexer* consta de dos partes fundamentales: la matriz de transición de estados y acciones semánticas, y el *motor* para la lógica de obtención del siguiente token. Sin embargo, la matriz de transición de estados y acciones semánticas requiere a su vez otros componentes de software para poder llevar a cabo sus funciones, los cuales serán descritos a continuación.

Matriz de Transición de Estados y Acciones Semánticas

La clase ***TransitionMatrix*** contiene toda la lógica y las estructuras necesarias para realizar la transición de estados y al mismo tiempo realizar acciones semánticas involucradas. Esto se debe a que resultó más sencillo incluir las acciones semánticas dentro de transiciones en el contexto de la matriz de estados en lugar de mantener dos estructuras diferentes. Es importante destacar que todas las estructuras utilizadas en el analizador léxico son dinámicas, lo cual fue un requisito solicitado explícitamente por la cátedra. Ésta estructura se implementó utilizando una tabla de hash, cuya clave es un entero para representar un estado de partida y su valor es una lista de transiciones. En otras palabras, la estructura se declara como una tabla de hash parametrizada según *Map<Integer, List<Transition>>*, lo cual permite agregar nuevos estados y transiciones dinámicamente.

Las transiciones se llevaron a cabo en la clase ***Transition***. Estas transiciones poseen tres elementos esenciales: un estado destino representado con un entero, la regla de transición que define si es posible pasar del estado anterior al siguiente con el último carácter leído y una acción semántica para aplicar en caso de que sea posible la transición [figura 2].

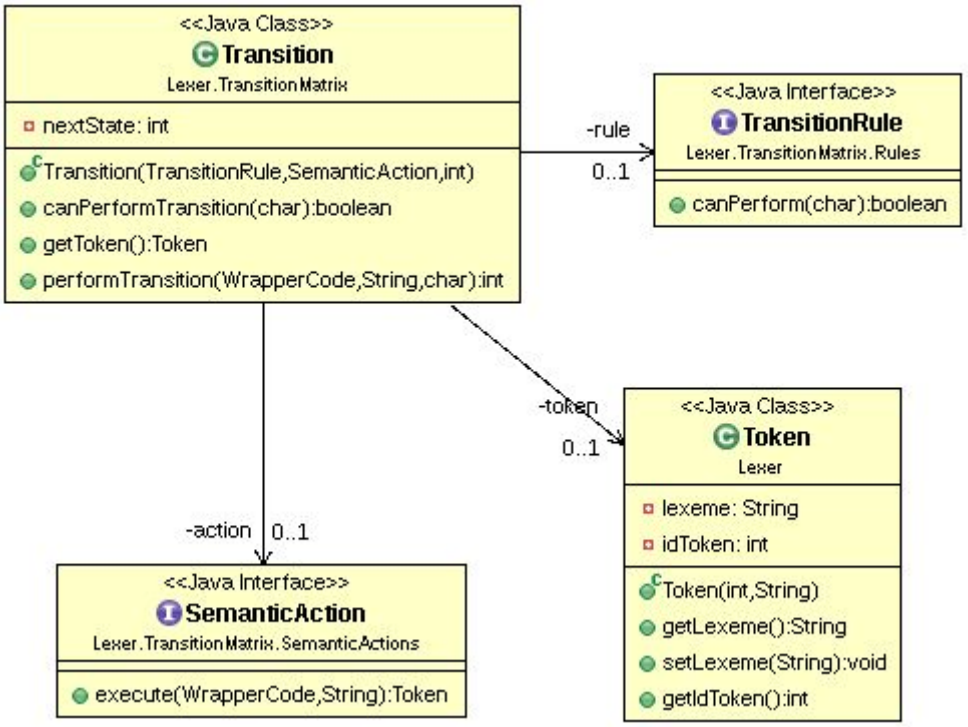

Figura 2: clase Transition y relaciones con otros componentes esenciales.

Las reglas de transiciones devuelven un valor booleano indicando si la transición de un estado a otro es posible con un caracter dado. En este caso resultó provechoso utilizar el patrón de diseño *Strategy* ya que cada regla de transición encapsula una lógica diferente en un método de igual signatura respecto a las demás reglas: todas devuelven un valor booleano y todas reciben un caracter para verificar si es posible la transición, lo cual se abstrajo a una interfaz llamada ***TransitionRule*** [figura 3]. Se identifican cinco casos particulares: una regla alfabética (***CharRule***), una regla de dígitos (***NumericRule***), una regla con algún caracter en particular (***SpecificcharacterTransition***) y una regla incondicional (***TrueRule***) que es utilizada por default en caso de que un estado no pueda pasar a ningún otro en la matriz de transiciones; cuando el carácter es inválido en ese contexto. Cabe soslayar que la regla alfabética reconoce las letras excepto la *'e'* y *'E'*, dado que estas poseen tratamientos específicos para constantes de punto flotante. Asimismo, identificamos el último caso que es en realidad una composición entre dos objetos del tipo ***TransitionRule*** ya que para evaluar un 'o' lógico ya que, por ejemplo, puede ocurrir el caso en el cual un identificador contenga tanto números como letras. Esta clase es llamada ***OrRule*** y fue implementada utilizando el patrón de diseño *Composite*.

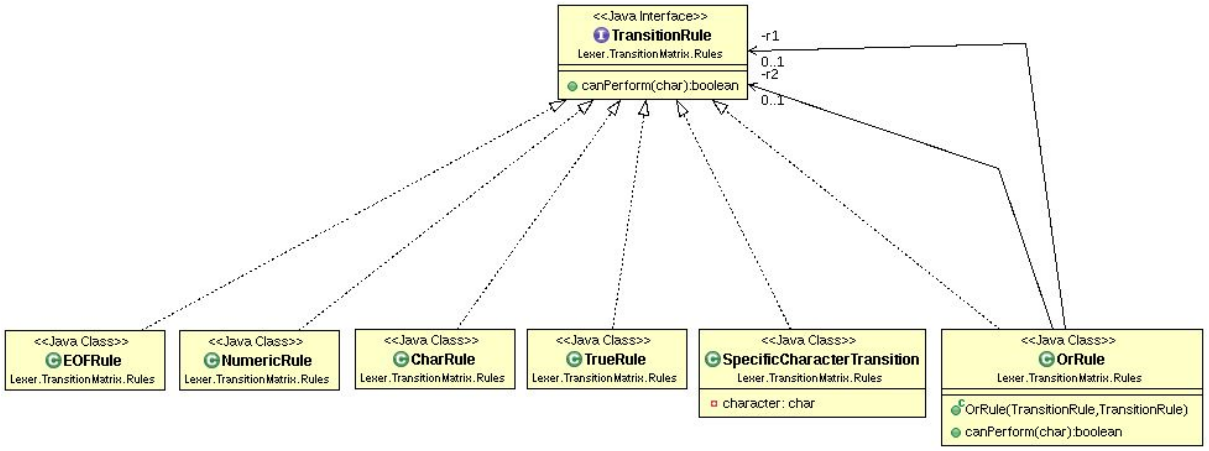

Figura 3: reglas de transición.

Finalmente, las acciones semánticas incluidas dentro de las transiciones se implementaron de forma tal que generen un objeto ***Token*** cuando sea necesario o bien un valor nulo (***null***) cuando no deba generar uno (cuando no esté correspondida a un estado final). Nuevamente es menester la utilización del patrón *Strategy* debido a que cada acción semántica tiene una lógica diferente encapsulada en métodos con la misma signatura. Esta abstracción se encuentra en la clase ***SemanticAction***, en donde el método *execute* devuelve un **Token** (o ***null***) y recibe un código y un buffer contenidos en un objeto para poder modificarlos cuando sea necesario [figura 4]. En nuestro caso, se identifican once casos distintos de acciones semánticas [tabla 3].

Por otro lado, la necesidad de "*wrappear*" el código y el *buffer* surge debido a que los objetos Strings son inmutables en java y no retornan modificados al llamador de la función, es decir, no pueden ser pasados como parámetros por referencia. La clase que envuelve el *buffer* y el código es llamada ***WrapperCode*** y también posee la línea actual y la cantidad de errores encontrados.

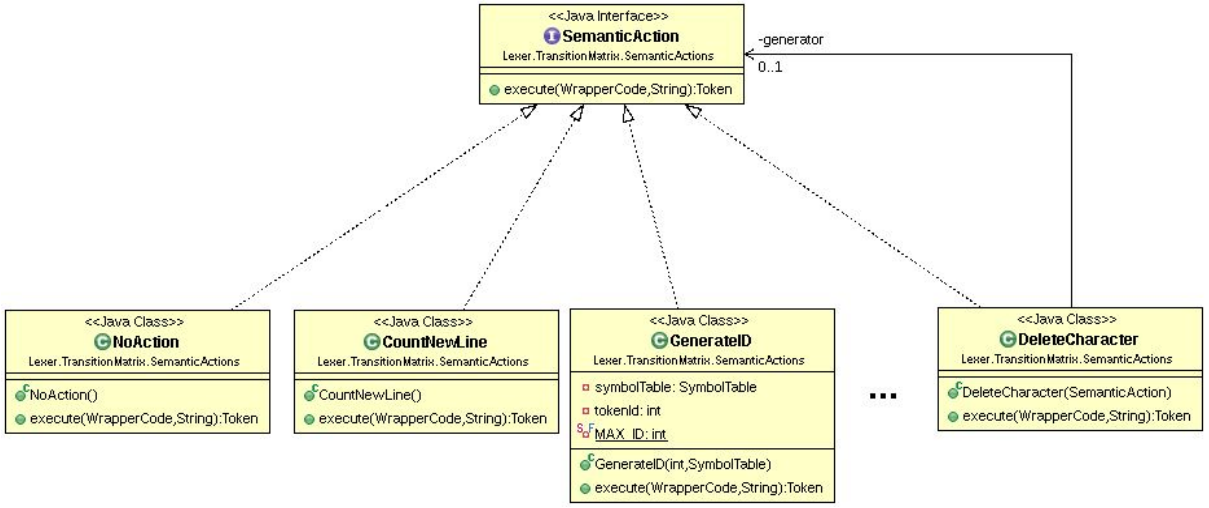

Figura 4: acciones semánticas.

Clase	Descripción

GenerateID	Elimina el último caracter del buffer. En caso de que supere 15 caracteres trunca el lexema y arroja Warning. Genera el token identificador o bien de una palabra reservada del lenguaje según la tabla de símbolos. Por otro lado, inserta el identificador a la tabla de símbolos.
GenerateULONG	Elimina el último caracter del buffer. Valida los límites de números largos sin signo. Si está fuera de rango descarta el buffer, pero si se encuentra en rango genera el token de una constante de tipo *ULONG*.
GenerateFloat	Elimina el último caracter del buffer. Valida los límites de números flotantes. Si está fuera de rango descarta el buffer, pero si se encuentra en rango genera el token de una constante de tipo *FLOAT*.
GenerateOperator	Genera el token con el código *ASCII* de un operador o símbolo simple, es decir, cuando el lexema del token posee un único caracter.
GenerateString	Genera un token de string y elimina las comillas del lexema.
GenerateCompoundSymbol	Genera un token para operadores de más de un símbolo. Es decir, los comparadores, ya que poseen dos caracteres en su lexema. Esto ocurre dado que no podemos obtener el código ASCII de dos caracteres, pero podemos pasar por constructor el ID de token específico para este elemento al crear el objeto.
SkipSymbol	Elimina el primer símbolo del lexema y arroja Warning de símbolo inválido.
SkipNewLineString	Elimina los últimos cuatro caracteres del buffer que se corresponden con los puntos suspensivos y el salto de línea de los strings multilíneas.
NoAction	No hace nada. Es simplemente para no generar *NullPointerExceptions* en el código.
GenerateUnclosedToken	Arroja un *warning* debido a la falta de cierre de cadenas de texto o comentarios. En el caso del string, genera el token asumiendo que el resto del archivo es un string y en el caso del comentario consume el resto del archivo como parte del mismo pero no devuelve ningún token debido.
CountNewLine	Suma 1 en la variable de conteo de líneas.

Tabla 3: descripción de las implementaciones de las clases semánticas.

Motor del Analizador Léxico

El llamado “motor” del analizador léxico une todas las partes; utiliza la matriz de transición de estados que contiene los estados, las reglas de transición, las acciones semánticas y opera con la tabla de símbolos. Esta lógica está desarrollada en un único método llamado *getNextToken()* dentro de la clase *Lexer*. Esta clase, además de poseer este método genera la matriz de transición de estados, instancia la tabla de símbolos e inserta las palabras reservadas.

El método *getNextToken()* itera sobre el código hasta que no quede nada para leer, es decir, hasta llegar al fin de archivo. En cada iteración se toma un caracter del código y se quita del mismo y eventualmente es devuelto por medio de una acción semántica (si corresponde). Por otro lado, existe una variable que da cuenta del estado actual en el autómata; cada vez que se realiza una transición, esta posee el nuevo estado y en cada iteración varía. En el último caso, *getNextToken()* devolverá un valor nulo (***null***) y será allí cuando se dejen de consumir los tokens ya que no hay más a reconocer en el código. A continuación se ilustra la situación mediante un pseudocódigo [pseudocódigo 1].

```
Token getNextToken() {
	int currentState = 0; //el principio
	char currentChar = “”;
	string buffer = "";


	while (code.length > 0 && currentState != ESTADO_FINAL) {
		currentChar = code.removeFirstChar();
		buffer += currentChar;
		currentState = transitions.performTransition(
			currentState,
			buffer,
			currentChar,
			code
		);
		if (currentState == 0)
			buffer = “”;//Descarto.
	}
	return transitions.getToken();
}
```

Pseudocódigo 1: obtención del siguiente token.

Las constantes utilizadas en el lexer se encuentran en la clase ***LexerConstants*** para mantener el código limpio y ordenado.

Tabla de Símbolos

La tabla de símbolos está implementada en la clase ***SymbolTable***, la cual está encargada de mantener y utilizar una estructura dinámica, específicamente una tabla de hash parametrizada cuyo primer parámetro es un *String*, el cual representa el lexema del token contenido en el símbolo y el segundo parámetro es de tipo *List<Symbol>*. La clase ***Symbol*** contiene un token (identificador y lexema), el tipo del identificador/constante y un valor booleano para saber si es una palabra reservada o no. La lista de símbolos sirve para poder poseer dos símbolos con el mismo lexema ya que podría darse el caso en donde existan dos identificadores con el mismo lexema pero se utilicen en contextos diferentes; una función puede llamarse igual que una variable pero son diferentes símbolos. En nuestro

caso particular, esto puede ser de utilidad en el caso de *shadowing*, utilizado en el trabajo práctico 3.

Finalmente, una vista más amplia de las distintas clases utilizadas en el analizador léxico puede verse en la siguiente imagen [Figura 5].

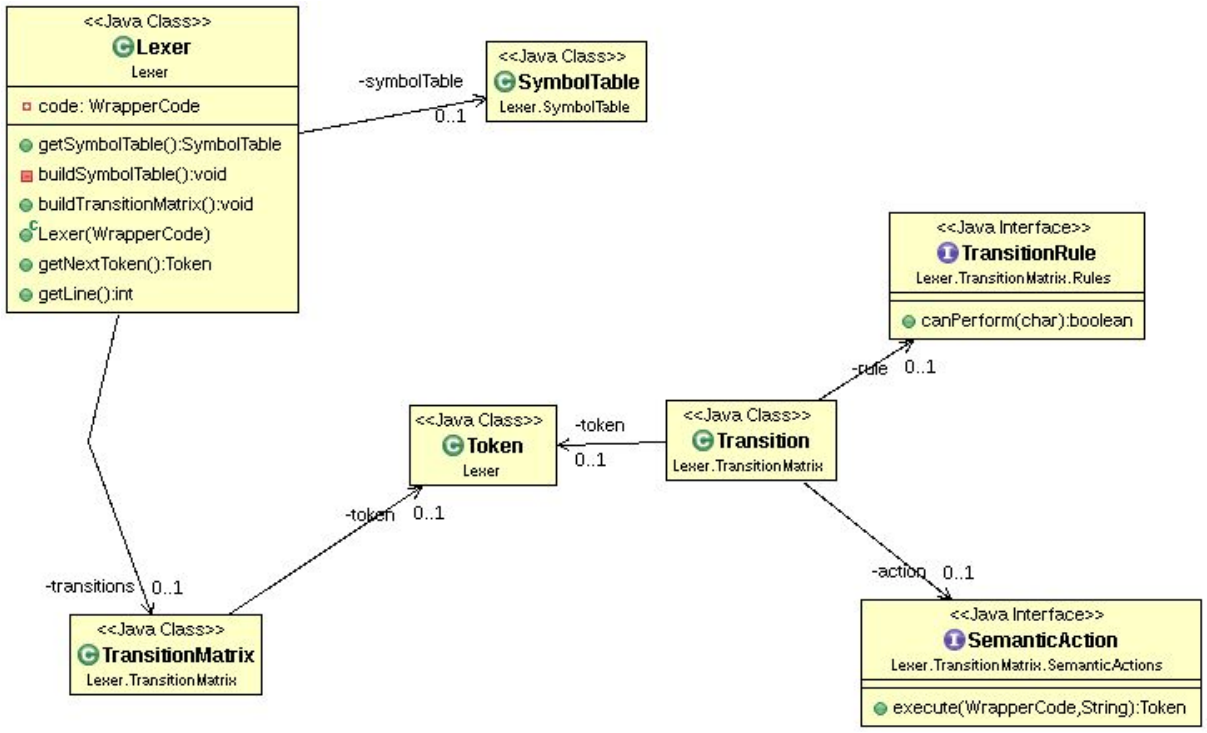

Figura 5: clases importantes del analizador léxico.

Consideraciones

Con el fin de clarificar algunos aspectos importantes del analizador léxico, se listan algunos supuestos considerados por el grupo.

1. En la columna 'others' de la matriz de transición de estados, también presente en el diagrama de transición de estados, se intenta mostrar que *'others'* es cualquier símbolo excepto los que estén explicitados en las columnas de la matriz. Esto se utilizó para acortar la escritura ya que el resultado sería ilegible.
2. Cuando se encuentren comentarios multilínea con más de tres puntos antes del salto de línea, se cuentan los puntos que sobren como parte del string. Por ejemplo, sea el siguiente código:
   "`comentario .....`
   `multilinea`"
   se generaría un string "*comentario ..multilínea*". Además, el salto de línea se quita y no se utiliza un espacio para reemplazarlo, con lo cual si no se introduce un espacio explícito, se concatenarán las líneas sin espacio de por medio.
3. Cuando se encuentre un comentario multilínea que no tenga los tres puntos suspensivos antes del salto de línea se descartará el buffer leído hasta ese entonces

y se volverá al estado inicial. Por ejemplo, sea el siguiente código:
“comentario..
multilinea”
cuando el lexer llegue a leer el salto de línea, no habrá encontrado tres puntos sino dos. En este contexto, el lexer descartará el buffer y arrojará un error por el símbolo no reconocido. Luego seguirá leyendo y encontrará un identificador llamado “multilinea” y una apertura de string ““, pero no encontrará un cierre de string, con lo cual arrojará un warning y reconocerá un string vacío.

4. Los strings vacíos se almacenan en la tabla de símbolos.
5. La tabla de símbolos contiene las palabras reservadas del lenguaje y los símbolos reconocidos en el analizador léxico por una cuestión de simplicidad. Sin embargo, al imprimirse se separa en dos secciones: palabras reservadas del lenguaje y símbolos reconocidos.
6. El caracter *EOF* no se puede obtener al leer el archivo dado que Java provee un adaptador en el tratamiento de los archivos que provee la lectura del mismo por líneas, con lo cual la forma correcta de dejar de iterar sobre el archivo es cuando no haya más líneas. Sin embargo, el archivo se vuelca en un string para que el lexer pueda procesar el código con lo cual:
    a. Por cada línea leída se inserta en el string que contiene el código explícitamente el caracter de salto de línea ‘\n .‘
    b. Al finalizar de leer el archivo se inserta explícitamente el caracter *EOF*.
7. El rango de los números de punto flotante no es exacto. Es decir, si bien el límite inferior es *1,17549435E-38*, el valor *1,17549434E-38* se encontrará fuera de rango aunque no debería estarlo. Este se debe a la representación interna de los números de punto flotante de Java, y debe contemplarse utilizar menos dígitos (ver casos de prueba) ya que no se puede comparar con tanta precisión.
8. Cuando existan comentarios o cadenas de texto sin cerrar se asumirá que el resto del archivo es parte del string/comentario y se arrojará un warning.
9. En muchos errores léxicos se imprime por pantalla el mensaje “buffer descartado”, lo cual implica no reconocer el token que se estaba reconociendo por algún error léxico.

Casos de Prueba

A continuación se muestra la tabla que contiene el mapeo entre los casos de prueba, el archivo relacionado y el resultado obtenido por el mismo [Tabla 4].

Número de archivo	Subíndice	Caso de Prueba	Arroja error	Resultado
1	1	Constante flotante con valor inferior dentro de rango.	No.	Token reconocido.
1	2	Constante flotante con valor superior dentro de rango.	No.	Token reconocido.
1	3	Constante 0,0.	No.	Token reconocido.

1	4	Constante flotante con primer valor inferior fuera de rango.	Sí.	ERROR: El número flotante 1,174E-38 se encuentra fuera del rango válido. Descartando token.
1	5	Constante flotante con primer valor superior fuera de rango.	Sí.	ERROR: El número flotante 3,41E38 se encuentra fuera del rango válido. Descartando token.
2	1	Constante entera larga sin signo con valor inferior dentro de rango.	No.	Token reconocido.
2	2	Constante entera larga sin signo con valor superior dentro de rango.	No.	Token reconocido.
2	3	Constante entera larga sin signo con primer valor inferior fuera de rango.	No.	Reconoce dos tokens: - y 1.
2	4	Constante entera larga sin signo con primer valor superior fuera de rango.	Sí.	ERROR: El número largo sin signo 4294967296 se encuentra fuera del rango válido. Descartando token.
3	1	Mantisa de números flotantes con parte decimal de números.	No.	Token reconocido.
3	2	Mantisa de números flotantes sin parte decimal.	No.	Token reconocido.
3	3	Número flotante sin parte decimal y con exponente.	No.	Token reconocido.
3	4	Número flotante con parte decimal y con exponente.	No.	Token reconocido.
3	5	Número flotante sin parte decimal y sin exponente.	No.	Token reconocido.
3	6	Número flotante con parte decimal y sin exponente.	No.	Token reconocido.
3	7	Número flotante con exponente negativo.	No.	Token reconocido.
3	8	Número flotante con exponente positivo explícito.	No.	Token reconocido.
3	9	Número flotante con exponente positivo implícito.	No.	Token reconocido.
4	1	Identificador con menos de 15 caractéres.	No.	Token reconocido.
4	2	Identificador con más de 15 caracteres.	Sí.	WARNING: el identificador unidentificadorconunnombremuylargo supera el máximo de símbolos permitidos con lo cual se truncará a 15 símbolos.

5	1	Identificadores que contengan _.	No.	Token reconocido.
5	2	Identificadores que contengan dígitos.	No.	Token reconocido.
6	1	Identificador comenzando con _.	No.	Descarta _ y arroja error. Reconoce token identificador.
6	2	Identificador comenzando con un dígito.	No.	Reconoce dos tokens: una constante numérica y otro token identificador.
7	1	Palabras reservadas escritas en minúsculas.	No.	Todos los tokens reconocidos correctamente.
7	2	Palabras reservadas escritas en mayúsculas.	No.	Todos los tokens reconocidos correctamente.
8	1	Comentario multilinea bien definido.	No.	No reconoce ningún token dado que el comentario no se debe consumir.
8	2	Comentario multilinea sin cierre.	Sí.	WARNING: se detectó un comentario sin cierre de corchetes.
9	1	Cadena de una sola línea bien definida.	No.	Token reconocido.
9	2	Cadena de una sola línea sin cierre.	Sí.	WARNING: se detectó un String sin cierre de comillas.
9	3	Cadena multilínea bien definida.	No.	Token reconocido.
9	4	Cadena multilínea sin cierre.	Sí.	WARNING: se detectó un String sin cierre de comillas.
9	5	Cadena multilínea con .. en lugar de ...	Sí.	ERROR: símbolo no reconocido. Descartando "cadena.. Luego sigue reconociendo los demás tokens a partir del fallo.
10	1	Todos los símbolos solicitados por la cátedra.	No.	Tokens reconocidos.
10	2	Todos los símbolos solicitados por la cátedra con símbolos que no deben ser reconocidos.	Sí.	Descarta los símbolos erróneos y reconoce los correctos.

	Funciona sin errores.
	Funciona con warnings.
	Funciona con errores.
	Reconoce varios tokens.

	Descarta símbolo y reconoce un token.

Tabla 4: casos de prueba del analizador léxico.

Analizador Sintáctico

El analizador sintáctico es otra de las partes fundamentales del compilador en donde se define la gramática del lenguaje. Esto es sustancial, dado de aquí en más se trabajará sobre este componente en las próximas etapas.

La gramática del lenguaje se rige por una serie de reglas sobre los *tokens* que se consumen desde el analizador léxico. En este esquema, el analizador sintáctico consume tokens bajo demanda para poder verificar que la secuencia dada pertenece al lenguaje o no. En este contexto, no solo es importante definir una gramática para las estructuras válidas del lenguaje, sino también para las inválidas. Esto sirve para arrojar errores y advertencias que puedan ayudar al programador con cierta precisión, así también como para poder seguir consumiendo tokens una vez que se encuentren errores; el parser debe seguir consumiendo tokens hasta que el código fuente se haya consumido por completo por más que existan errores de sintaxis en el mismo, ergo, se define una gramática de error con estos fines.

En las siguientes secciones se hablará en detalle del diseño e implementación del analizador sintáctico.

Temas Asignados

En este apartado se listan los temas particulares asignados al grupo.

1. **SWITCH CASE**, con lo cual se debió incorporar las llaves (**{** y **}**) como caracteres válidos en el lexer.
2. Se deberán incorporar, como posibles sentencias ejecutables, asignaciones del tipo **LET <asignación> .**
3. **Conversiones Explícitas**: en cualquier lugar donde pueda aparecer una variable o una expresión, podrá aparecer: *UL_F(<expresion>)*, en donde **UL_F** será una palabra reservada, que se deberá incorporar a las anteriores.

Diseño e Implementación

A diferencia del analizador léxico, la lógica del analizador sintáctico es generada por el generador de parsers elegido por el grupo. En nuestro caso, optamos por la utilización de YACC.

YACC es un generador de analizadores sintácticos LALR(1), con lo cual es conveniente que la gramática definida sea recursiva a izquierda. La idea detrás de YACC es que se genere el código del parser mediante la descripción de la gramática mezclado con porciones de código Java. Una vez terminada la gramática, se genera el código Java mediante el comando *yacc -J gramatica.y*, el cual arroja una clase que encapsula toda la lógica necesaria. Existen funciones predefinidas de yacc tales como, *yylex()* e *yyparse()* que

son muy importantes para el funcionamiento y sirven para consumir el próximo token (pidiendo al analizador léxico) y para comenzar la etapa de parsing, respectivamente.

Para poder manejar los errores que surgen a partir de una secuencia de tokens no reconocida se introduce una gramática de error para informar al usuario las líneas que poseen errores. Esta gramática de error es muy larga y tediosa, pero es capaz de detectar reglas malformadas y anunciarlas. Por otro lado, se lleva un conteo de errores con el fin de notificarlo al usuario al finalizar la compilación. Una de las partes más difíciles fue la de anunciar la línea de error ya que si bien el lexer provee la línea de cada token, la recursión de las reglas gramaticales hace que sea complicado establecer la línea de error. Para solucionar este inconveniente se declaran dos variables en la clase *ParserVal*, la cual representa a la estructura de los elementos terminales y no terminales que se obtienen posicionalmente en *YACC*. Estos campos representan la línea en la cual comienza el token y la línea en la cual finaliza (ya que algunos pueden empezar y finalizar en distintas líneas). En cada regla, se obtiene posicionalmente un elemento terminal o no terminal y se pide la linea en la cual comienza o termina para poder establecer con mayor precisión la línea de error. Por ejemplo, el siguiente código da cuenta de ello:

```
IF condition ')' THEN executional_statements ELSE executional_statements END_IF
'.' {
       $$.begin_line = $1.begin_line;
       $$.end_line   = $9.end_line;
       this.currentRuleError++;
       this.yyerror(ParserConstants.ERR_LPAREN_EXPECTED, $1.end_line);
}
```

Se puede ver en esta regla que falta el símbolo ‘(‘ en la regla de selección.Entonces, para anunciar esto se toma la línea de la palabra reservada ‘IF’, es decir, ***$1.end_line***, ya que se asume que a continuación debería existir una apertura de paréntesis. Además, puede verse que la regla devuelve su comienzo de línea y fin de línea. En este caso comienza en la lína en donde se encuentre el token ‘IF’ y termina en el caracter ‘.’, ubicados en la posición $1 y $9, respectivamente. Para devolver información relacionada a la regla actual se utiliza **$$**, el cual es una instancia de la clase *ParserVal* previamente nombrada.

En muchas ocasiones surgieron conflictos, los cuales se solucionaron quebrando la regla en niveles o bien utilizando un caracter de sincronización, tal como ‘.’. La mayoría de los problemas surgen al introducir la gramática de error ya que el parser tiene conflictos para reducir reglas que en un punto se solapan. En este punto, *YACC* los resuelve y los anuncia por consola. En nuestra gramática no hay conflictos de ningún tipo ya que se solucionaron de esta manera.

Otro conflicto a solucionar fue el de los rangos de números negativos. En el caso de los números de punto flotante, el rango es el mismo para números positivos y negativos, con lo cual no hay que realizar tratamiento alguno en la gramática. Sin embargo, el caso de las constantes largas sin signo es diferente ya que estas no pueden estar acompañadas de un signo negativo. Esto no puede realizarse en la etapa de análisis léxico y debe chequearse en la etapa del analizador sintáctico ya que hay una sobrecarga del operador ‘-’. En la regla de constantes numéricas con signo negativo de la gramática se solicita a la tabla de símbolos el tipo de la constante y de ser *ulong* se arroja un error por estar fuera de rango, ya que estas deben ser mayores a cero. Finalmente, se remueve la constante

positiva de la tabla de símbolos ya que en la etapa del análisis sintáctico se insertó en la tabla de símbolos como una constante positiva.

Como las constantes de punto flotante pueden ser positivas o negativas se debe dar de alta en la tabla de símbolos las constantes negativas. Sin embargo, las constantes positivas son dadas de alta en la etapa del analizador léxico, con lo cual al intentar dar de alta una constante positiva puede ocurrir que se inserte la constante negativa, y también se deje la constante positiva o bien que se reemplace la constante positiva por la negativa y en realidad se hayan creado dos constantes de igual magnitud pero con distinto signo en el programa. Esto no es correcto ya que la tabla de símbolos debe dar de alta nada más ni nada menos las constantes que el programador definió, motivo por el cual se implementó un conteo de referencias. Esto es: cada constante positiva, al ser insertada en la tabla de símbolos posee una referencia por defecto. Una vez en la etapa de análisis sintáctico, cuando se encuentre una constante positiva se suma una referencia en la misma y de encontrarse una constante negativa, se resta una referencia en la constante positiva de igual magnitud y se corrobora la existencia de la constante negativa en la tabla de símbolos; si existe se suma un valor en su referencia y si no existe, se da de alta en la tabla de símbolos con una referencia. Esta lógica puede verse en la regla *<factor>* del archivo contenedor de la gramática.

Gramática

A continuación se describe la gramática definida en formato BNF.

```
1. <program> ::=
            /*VACÍO*/
          | <statements>
2. <statements> ::=
            <declarative_statements>
          | <executional_statements>
          | <statements> <declarative_statements>
          | <statements> <executional_statements>
3. <declarative_statements> ::= <variables_declaration_statement>
4. <variables_declaration_statement> ::= <var_list> ':' <type> '.'
5. <var_list> ::=
            ID
          | <var_list> ',' ID
6. <type> ::=
            ULONG
          | FLOAT
7. <executional_statements> ::=
            <if_statement>
          | <if_else_statement>
          | <assignation_statement>
          | <out_statement>
          | <switch_statement>
8. <if_statement> ::=
            IF '(' <condition> ')' THEN <executional_statements> END_IF '.'
          | IF '(' <condition> ')' THEN <explicit_delimited_execution_block>
   END_IF '.'
```

```
9. <if_else_statement> ::=
            IF '(' <condition> ')' THEN <executional_statements> ELSE
   <executional_statements> END_IF '.'
          | IF '(' <condition> ')' THEN <executional_statements> ELSE
   <explicit_delimited_execution_block> END_IF '.'
          | IF '(' <condition> ')' THEN <explicit_delimited_execution_block>
   ELSE <executional_statements> END_IF '.'
          | IF '(' <condition> ')' THEN <explicit_delimited_execution_block>
   ELSE <explicit_delimited_execution_block> END_IF '.'
10. <explicit_delimited_execution_block> ::= BEGIN <execution_block> END '.'
11. <execution_block> ::=
            <execution_block> <executional_statements>
          | <executional_statements>
12. <assignation_statement> ::=
            LET ID '=' <expression> '.'
          | ID '=' <expression> '.'
13. <out_statement> ::= OUT '(' STRING_CONST ')' '.'
14. <switch_statement> ::= SWITCH '(' ID ')' '{' <cases_rule> '}' '.'
15. <cases_rule> ::=
            <cases_rule> <case_rule>
          | <case_rule>
16. <case_rule> ::=
            CASE NUMERIC_CONST ':' <explicit_delimited_execution_block>
          | CASE NUMERIC_CONST ':' <executional_statements>
17. <condition> ::=
            <expression> EQUAL <expression>
          | <expression> '>' <expression>
          | <expression> '<' <expression>
          | <expression> GREATER_EQUAL <expression>
          | <expression> LESS_EQUAL <expression>
          | <expression> NOT_EQUAL <expression>
18. <expression> ::= <expression_rule> /*regla redundante debido a una
   corrección en la gramática.*/
19. <expression_rule> ::=
            <expression> '+' <term>
          | <expression> '-' <term>
          | <term>
20. <term> ::=
            <term> '*' <factor>
          | <term> '/' <factor>
          | <factor>
21. <factor> ::=
            ID
          | NUMERIC_CONST
          | '-' NUMERIC_CONST
          | UL_F '(' <expression_rule> ')'
```

Es importante destacar que toda la gramática es recursiva a izquierda y que si bien existen reglas redundantes, estas sirven pura y exclusivamente para poder mantener el código con mayor facilidad por los integrantes del grupo ya que en los próximos trabajos

prácticos se trabajará sobre esta gramática. Para una mejor comprensión de los distintos elementos no terminales, se listan con una breve descripción en la siguiente tabla [Tabla 5].

Regla	Descripción
<program>	Esta es la regla raíz de la gramática. Acepta programas en blanco (sin nada escrito) o bien programas formados por una o varias sentencias, ya sean declarativas o ejecutables o una combinación de ellas mediante la regla <statements>.
<statements>	Esta regla contiene todas las sentencias posibles del programa. Pueden ser declarativas y/o ejecutables. Se utiliza pura y exclusivamente para realizar la recursividad entre las reglas.
<declarative_statements>	Esta regla degenera en sentencias declarativas.
<variables_declaration_statement>	Esta regla sirve para aceptar sentencias de declaraciones de variables. Puede aceptar listas de identificadores o bien un solo identificador junto con su(s) tipo(s).
<var_list>	Esta regla acepta un sólo identificador o bien varios identificadores separados con ','.
<type>	Esta regla contiene los tipos posibles para las declaraciones de las variables. Éstos pueden ser *FLOAT* o bien *ULONG*.
<executional_statements>	Esta regla acepta sentencias ejecutables en el programa, las cuales pueden ser sentencias *if*, *if-else*, *switch*, sentencias de asignación y la sentencia *OUT*.
<if_statement>	Esta regla acepta sentencias de tipo *if*. La misma puede tener una sola sentencia ejecutable anidada o bien un bloque de sentencias de ejecución. No acepta sentencias declarativas dentro, por pedido de la cátedra.
<if_else_statement>	Esta regla acepta sentencias de tipo if. La misma puede tener una sola sentencia ejecutable anidada o bien un bloque de sentencias de ejecución. No acepta sentencias declarativas dentro, por pedido de la cátedra.
<explicit_delimited_execution_block>	Esta regla envuelve un bloque de ejecución entre *BEGIN* y *END*. Esto se hizo para agregar más claridad en la gramática, aunque es redundante y podría obviarse.
<execution_block>	Esta regla acepta uno o más sentencias de ejecución. No acepta sentencias declarativas ya que los bloques de ejecución sólo se utilizan en las estructuras de control.
<assignation_statement>	Esta regla acepta sentencias de asignación de expresiones a identificadores con y sin *LET*.
<out_statement>	Esta regla acepta una sentencia de impresión de cadenas de texto *OUT* cuyo parámetro es un string.
<switch_statement>	Esta regla acepta una sentencia *switch*.
<cases_rule>	Esta regla acepta uno o más *cases*. La regla sirve para realizar la recursión a izquierda de los distintos casos para

	aceptar más de uno de ser necesario.
`<case_rule>`	Esta regla acepta un único *case* utilizado dentro de la sentencia *switch*.
`<condition>`	Esta regla acepta condiciones simples con comparadores entre expresiones.
`<expression>`	Esta regla es una indirección redundante que se dejó debido a una corrección en el trabajo. Se podría quitar sin problema alguno.
`<expression_rule>`	Esa regla acepta expresiones para establecer composiciones entre identificadores, constantes, y todas las operaciones aritméticas posibles.
`<term>`	Esta regla acepta operaciones aritméticas de multiplicación y división de factores y está separada de estos para establecer precedencia por niveles.
`<factor>`	Esta regla acepta operaciones aritméticas de suma y resta entre constantes e identificadores y también conversiones de expresiones *UL_F(<expression>)*.

Tabla 5: descripción de los elementos no-terminales o reglas gramaticales.

Consideraciones

A continuación se listan las consideraciones utilizadas por el grupo en el analizador sintáctico.

1. Si bien existen muchas indirecciones o reglas redundantes que podrían omitirse, estas sirven al grupo para ver con mayor claridad el código y es conveniente para el mantenimiento en etapas posteriores.
2. La sentencia *UL_F* no puede anidarse consigo misma. Es decir, no se puede introducir *UL_F(UL_F(<expression>))*, ni tampoco convertirse una expresión o variable que ya sea de tipo *float* ya que no tiene sentido, hablando estrictamente de la semántica.
3. Los programas en blanco son aceptados por el parser.
4. En el enunciado se solicita que todas las sentencias terminan con ‘.’, lo cual lo tomamos literalmente. En la gramática todas las sentencias tienen el caracter de sincronización ‘.’ al final, incluso las llaves del *switch*.
5. La sentencia switch debe tener al menos un *case* dentro, de lo contrario, será tomado como un error de sintaxis.
6. En muchas casos es necesario descartar las sentencias anidadas cuando ocurre un error en una sentencia de un nivel superior (sobre la que se anida). Por ejemplo, cuando una sentencia *switch* no abre llaves ‘{‘. Esto se debe a que la regla de la gramática de error debe consumir tokens hasta un caracter de sincronización, el cual en nuestro caso es el ‘.’ ya que si se utiliza el token *error* sin un caracter de sincronización consumiría como error el resto del programa y esto no debe ocurrir. Esto es posible ya que si la estructura de nivel superior tiene un error de sintaxis no debe generarse código alguno para el programa, con lo cual se debe notificar el

error y continuar leyendo el programa, sin importar las sentencias anidadas. En otros casos es posible seguir leyendo las sentencias anidadas, y en algunos resulta más complejo dada la utilización del token *error*. Cabe destacar que de no utilizar este token, la gramática generaría conflictos.

7. Al diagnosticar los errores, se intenta explicitar con la mayor precisión posible el mismo. En algunos casos puede ocurrir que una sentencia tenga múltiples errores o errores que no pudieron ser descritos en la gramática de error. Esto se debe a que no se pudieron contemplar todas las combinaciones de error en todas las reglas por lo tedioso que esto resultaría. Por ello, se utiliza el token error en las sentencias de más alto nivel por si en alguna de ellas ocurrió un error por varios y así poder continuar con el análisis del programa, tal como se muestra en la regla a continuación:

```
statements:
              declarative_statements
       |      executional_statements
       |      statements declarative_statements
       |      statements executional_statements
       |      error '.'
              {
                     this.yyerror("Al menos un error encontrado en una
sentencia cercana.");
};
```

   Cabe destacar que el mensaje “al menos un error encontrado en una sentencia cercana” se refiere a que está cerca de la línea especificada ya que puede ser que no se encuentre justamente en esa línea. Esto se debe a que no se sabe a priori qué error ocurrió ni donde específicamente, pero generalmente acierta la línea.
8. Cuando se reconoce una regla con *LET* el compilador imprime “se reconoció una regla de asignación”, al igual que si no tuviera *LET*. De todas formas las reglas están diferenciadas y más allá de que el mensaje sea igual, en las acciones semánticas particulares se dará un tratamiento específico en los futuros trabajos ya que tienen una semántica diferente.
9. Cuando el parser encuentra una constante *ULONG* negativa arroja error ya que esta constante está fuera de rango. Esto se debe a que son constantes largas sin signo. Esto se lleva a cabo cuando el parser verifica en la tabla de símbolos si la constante es de tipo *ulong* ya que la gramática sólo acepta constantes numéricas indiscriminadamente. Por otro lado, el parser no elimina de la tabla de símbolos la constante positiva que se insertó en el lexer, ya que de igual forma no se generará código para la sentencia.
10. Cuando el parser encuentra una constante *FLOAT* negativa, es decir, cae en la regla con el token ‘-’ y la constante numérica, al verificar en la tabla de símbolos que es de tipo float, cambia la entrada del símbolo en cuestión en la tabla de símbolos por la misma constante en negativo. Es decir, si durante el lexer se almacenó una constante *X* de punto flotante (positiva), cuando se llegue a la regla del parser en donde se sepa que la constante era en realidad negativa se procede a cambiar el valor *X* de la tabla de símbolos por *-X*.

Casos de Prueba

A continuación se listan casos de prueba realizados sobre el analizador sintáctico respecto a todas las estructuras válidas. También se prueban distintos casos de estructuras con errores, las cuales no deben ser aceptadas [tabla 6].

Número de archivo	Subíndice	Caso de Prueba	Arroja error	Resultado
1	1	Sentencia declarativa con una sola variable.	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.
1	2	Sentencia declarativa con más de una variable.	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.
1	3	Sentencias declarativas mal definidas.	Sí.	Errores anunciados correctamente. El parser nunca deja de consumir tokens debido a los errores.
2	1	Sentencia "if".	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.
2	2	Sentencia "if" mal definida.	Sí.	Errores anunciados correctamente. El parser nunca deja de consumir tokens debido a los errores.
3	1	Sentencia "if-else".	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.
3	2	Sentencia "if-else" mal definida.	Sí.	Errores anunciados correctamente. El parser nunca deja de consumir tokens debido a los errores.
4	1	Sentencia de control "switch" con múltiples cases.	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.
4	2	Sentencia de control "switch" mal definida.	Sí.	Errores anunciados correctamente. El parser nunca deja de consumir tokens debido a los errores.
5	1	Asignaciones simples y con expresiones.	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.

5	2	Asignaciones mal definidas.	Sí.	Errores anunciados correctamente. El parser nunca deja de consumir tokens debido a los errores.
6	1	Sentencia LET.	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.
6	2	Sentencia LET mal definida.	Sí.	Errores anunciados correctamente. El parser nunca deja de consumir tokens debido a los errores.
7	1	Sentencia OUT.	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.
7	2	Sentencia OUT mal definida.	Sí.	Errores anunciados correctamente. El parser nunca deja de consumir tokens debido a los errores.
8	1	Sentencia UL_F.	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.
8	2	Sentencia UL_F mal definida.	Sí.	Errores anunciados correctamente. El parser nunca deja de consumir tokens debido a los errores.
9	1	Constante float negativa.	No.	Acepta el token y cambia actualiza la tabla de símbolos por la constante negativa.
9	2	Constante ulong fuera de rango (las flotantes están cubiertas en el lexer).	Sí.	Error de constante fuera de rango anunciado y se quita la constante positiva de la tabla de símbolos.
	Funciona sin errores.			
	Funciona con errores.			

Tabla 6: casos de prueba del analizador sintáctico.

Conclusión

La realización del analizador léxico y sintáctico ayudó a comprender a los integrantes del grupo el funcionamiento de los compiladores utilizados a los largo de la carrera. Si bien la complejidad de la implementación no fue muy dificultosa, resultó muy tediosa debido a la cantidad de elementos a reconocer en el lenguaje. El analizador léxico fue la situación más compleja a cubrir ya que requería la realización manual del mismo y además, dependía casi en su totalidad de un buen diseño en la matriz de estados. En el caso del analizador

sintáctico, las horas de trabajo se redujeron considerablemente gracias a la generación del mismo utilizando YACC, aunque comprendemos que de no haber utilizado un generador, el código hubiese sido más complejo y laborioso que el caso del analizador léxico. Finalmente, nos permitió ver cómo tareas que parecen ser sencillas como la notificación de errores en una línea determinada es complejo de realizar en la práctica.

Esta entrega fue para nosotros la primera vez en donde se pone en práctica la teoría aprendida en materias como Ciencias de la Computación 1 y Lenguajes de Programación 1, lo cual nos permite afianzar los conceptos adquiridos a lo largo de la carrera.