

Diseño de Compiladores 1

Trabajo Practico 1 y 2



Autores:

Roman Luna

romanluna1234@gmail.com

Savo Gabriel Koprivica

skoprivica@alumnos.exa.unicen.edu.ar

Grupo:

20

Cátedra:

Diseño de compiladores 1

Profesores:

Jose A. Fernandez Leon

Introducción.....	3
Temas Particulares.....	3
Decisiones de Diseño.....	4
Analizador Léxico.....	5
Clases.....	6
Tabla de símbolos.....	6
Símbolo.....	6
LectorArchivo.....	7
Constantes.....	7
Main.....	7
AnalizadorLexico.....	7
Matriz de Acciones Semánticas:.....	8
Matriz de transición de estados:.....	9
Autómata De transición de	
Estados(link: https://lucid.app/lucidchart/5c4ef08f-0410-41d1-bbaf-ababff8bef52/edit?invitationId=inv_f167d908-d5f9-4819-941f-a7754df7bc3b&page=0_0#).....	10
Consideración y decisiones tomadas con respecto a los errores.....	11
Salida del Analizador Léxico.....	11
Analizador Sintactico.....	13
Decisiones e implementación:.....	13
Lista de no terminales de la Gramática:.....	13
Manejo de errores del analizador sintáctico.....	14
Salida el analizador Sintáctico.....	14
Errores y consideraciones en estas etapas.....	19
Ejecución del archivo Jar.....	20
Conclusiones.....	23

Introducción

El presente informe exhibe el desarrollo de la creación de un compilador basado en el lenguaje Java. En este informe se verá la creación del analizador Léxico (El cual se encarga de detectar los tokens del código, así como también los errores léxicos que pueda contener) y el analizador Sintáctico (Encargado de la parte sintáctica del código, detectando las estructuras sintácticas y errores). Las decisiones tomadas y cómo resolvimos diversos problemas o cuestiones que nos fueron surgiendo.

Temas Particulares

6. Enteros largos (32 bits): Constantes enteras con valores entre -2^{31} y $2^{31} - 1$. Estas constantes llevarán el sufijo “**l**”.

Enteros sin signo (16 bits): Constantes con valores entre 0 y $2^{16} - 1$. Estas constantes llevarán el sufijo “**ui**”. Se deben incorporar a la lista de palabras reservadas las palabras **LONG** y **UINT**.

7. Punto Flotante de 32 bits: Números reales con signo y parte exponencial. La parte exponencial puede estar ausente. Si está presente, el exponente comienza con la letra “e” (mayúscula o minúscula) y el signo del exponente es obligatorio.

Puede estar ausente la parte entera o la parte decimal, pero no ambas. El “.” es obligatorio. Ejemplos válidos:

1. .6 -1.2 3.e-5 2.E+34 2.5E-1 15. 0. 1.2e+10

Considerar el rango $1.17549435E-38 < x < 3.40282347E+38$ U

$-3.40282347E+38 < x < -1.17549435E-38$ U 0.0

10. En los lugares donde un identificador puede utilizarse como operando (expresiones aritméticas o comparaciones), considerar el uso del operador ‘--’ luego del identificador. Por ejemplo: $a = b-- * 7_i$,

$z = a---b--$,

IF ($a-- > 2_i$) ...

13 .WHILE (<condicion>) **DO** <Bloque_de_sentencias_ejecutables>

<condicion> tendrá la misma definición que la condición de las sentencias de selección.

<Bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves.

18. Herencia por Composición - Uso con nombre

19. Se dara en el Trabajo Practico N°3

21. Forward declaration

26. Se dara en el Trabajo Practico N°3

33. Comentarios multilinea: Comentarios que comiencen con “/*” y terminen con “*/” (estos comentarios pueden ocupar más de una línea).

mundo! %

Para la implementación del compilador, decidimos usar el lenguaje java. Creamos una tabla de símbolos que se encarga de almacenar todos los símbolos de identificadores, constantes y cadenas distintas que haya en el código que se está compilando. Los símbolos en esta etapa solo tienen el lema y el identificador correspondiente. A su vez también tenemos 2 estructuras más encargadas de almacenar el resto de símbolos posibles, la tabla de palabras reservadas y la tabla de caracteres ASCII. La decisión de qué tabla utilizar dependiendo el token es de las acciones semánticas. Dependiendo del estado actual y el carácter actual es la acción semántica que se ejecuta. Las acciones semánticas son las encargadas de ir armando los tokens de manera correcta, así como también detectar los errores y almacenar los símbolos detectados. En lo que corresponde a la parte sintáctica, implementamos la gramática siguiendo las indicaciones propuestas por la cátedra.

Las estructuras antes mencionadas son creadas a partir de una clase que se encarga de leer los archivos txt y formarlas. Todo lo antes mencionado es detallado a continuación en el informe.

Analizador Léxico

Tenemos los siguientes paquetes:

- **Acciones Semánticas:** En este paquete se encuentran todas las acciones semánticas utilizadas para poder construir los token de manera correcta. Para implementarla decidimos utilizar una interfaz, la cual implementamos en cada AS individual.

Las acciones semánticas son:

- AS0: Esta Acción Semántica se encarga de leer los espacios en blanco, tabulaciones y saltos de línea, sin agregarlos al token actual. En caso de ser un salto de línea también aumenta uno la variable `linea_actual` del analizador léxico.
- AS1: Esta Acción Semántica se encarga solamente de leer el próximo carácter del código y concatenarlo con el token actual. Si el carácter es un salto de línea le suma 1 a la variable `linea_actual` del Analizador Léxico. Esta acción retorna 0 lo que significa que que el token todavía no está listo y que tiene que seguir leyendo.
- AS2: Esta Acción Semántica se encarga de leer los caracteres especiales: '+', '/', ';', ',', '{', '}', '(', ')'. Los busca en el mapa de caracteres ASCII y retorna el identificador del mismo.
- AS3: Esta Acción Semántica se encarga de verificar que el token encontrado corresponda a un identificador válido. Se encarga de verificar que su longitud no supere los 20 caracteres, en cuyo caso lo trunca eliminando los caracteres excedentes del final y agrega el Warning a la lista de errores del Analizador Léxico, para luego informarle sobre el mismo al usuario y en que línea ocurrió. Luego de verificar su longitud, lo agrega a la tabla de símbolos en caso de no estar y retorna el ID correspondiente.
- AS4: Esta Acción Semántica se encarga de verificar si el token obtenido pertenece a una palabra reservada(del tipo ==, >=, etc) o si pertenece a un carácter ASCII(=, >, <, etc). Luego obtiene el símbolo y retorna el identificador del mismo.
- AS5: Esta Acción Semántica se encarga de verificar que tipo de constante entera se encontró, si un entero largo o un entero sin signo. Luego verifica que el rango no supere el máximo valor permitido(A este se le suma uno más por el rango de los negativos). Luego se verifica el rango en la Gramática, dependiendo de si es un número positivo o negativo. Almacena el valor en la tabla de símbolos y entrega el token.
- AS6: Esta Acción Semántica se encarga de verificar si la constante flotante excede o no el rango permitido, en caso de no hacerlo lo agrega a la tabla de símbolos y retorna el identificador.

- AS7: Esta Acción Semántica se encarga de leer el siguiente carácter, concatenado con el token actual ya que forma parte del mismo y luego retornando el identificador del mismo.
- AS8: Esta Acción Semántica Se encarga de buscar si el token encontrado pertenece a una palabra reservada válida. Si es así, retorna el identificador de la misma.
- AS9: Esta acción Semántica se encarga de leer los comentarios. Una vez finalizado el token de comentario, esta acción se encarga de eliminarlo y retirar 0,(que significa que siga leyendo). Ya que los comentarios no deben ser almacenados como tokens en la tabla de símbolos.
- AS10: Esta acción Semántica se encarga de almacenar las cadenas en la tabla de símbolo y retorna el id del mismo.
- ASE: Acción semántica encargada de retornar error. En caso de que ocurra un error, por ejemplo, detecte un carácter no válido. Esta acción retorna -1 y detiene la ejecución de la compilación.
- **archivosTxt:** En este paquete se encuentran los archivos txt con los cuales generamos las diferentes tablas/matrices/mapas que vamos a necesitar para el funcionamiento de nuestro analizador léxico:
 - Tabla de palabras reservadas.
 - Tabla de caracteres ASCII.
 - Matriz de acciones semánticas.
 - Matriz de transición de estados.
 - Gramática(Utilizada para generar el parser).
- **Compilador:** Este paquete contiene todas las clases que se utilizan para el funcionamiento del Compilador. Las cuales detallaremos posteriormente en el informe
- **Testeos:** Contiene todos los códigos de prueba utilizados para verificar que el compilador funciona de manera correcta.

Clases

Tabla de símbolos

Esta clase es la encargada de almacenar los diferentes símbolos que van surgiendo en el código que estemos compilando. Los símbolos se guardan en un Hashtable<string, símbolo> en el cual la llave string corresponde al lexema. Antes de agregar un nuevo símbolo corrobora que no exista previamente en la tabla. Estos símbolos pueden corresponder a identificadores, constantes o cadenas.

Símbolo

En esta etapa del trabajo un símbolo solo tiene un Lexema y un id asociado. En entregas posteriores se le agregara un tipo y un ámbito

LectorArchivo

Esta clase la decidimos implementar para mantener un mejor orden en el compilador. Dicha clase se encarga de crear las tablas o matrices correspondientes a partir de los archivos txt mencionados anteriormente. Como por ejemplo la Matriz de transición estados, matriz de acciones semánticas, caracteres ascii, etc.

Constantes

Como Lector Archivo, esta clase la decidimos crear para mejorar la legibilidad y el orden del código. En esta se encuentran constantes que utilizamos en varias partes del programa y a su vez las tablas/matrices que vamos a utilizar, como la Matriz de acciones semánticas o las palabras reservadas, etc.

Main

El main es el encargado de realizar la ejecución del parser, el cual utiliza el léxico. Primero seteamos el código a compilar en el Analizador Léxico y luego se ejecuta la función run() del parser. Mientras el parser se ejecuta, se van consumiendo los tokens encontrados y se va imprimiendo por pantalla las sentencias/bloques que se van detectando con el parser. Los errores sintácticos y léxicos se van almacenando para luego ser mostrados al final de la ejecución del compilador. Una vez finalizada la ejecución de la función parser.run(). El main imprime todos los errores sintáctico encontrados así como también la lista de tokens detectados como los errores léxicos.

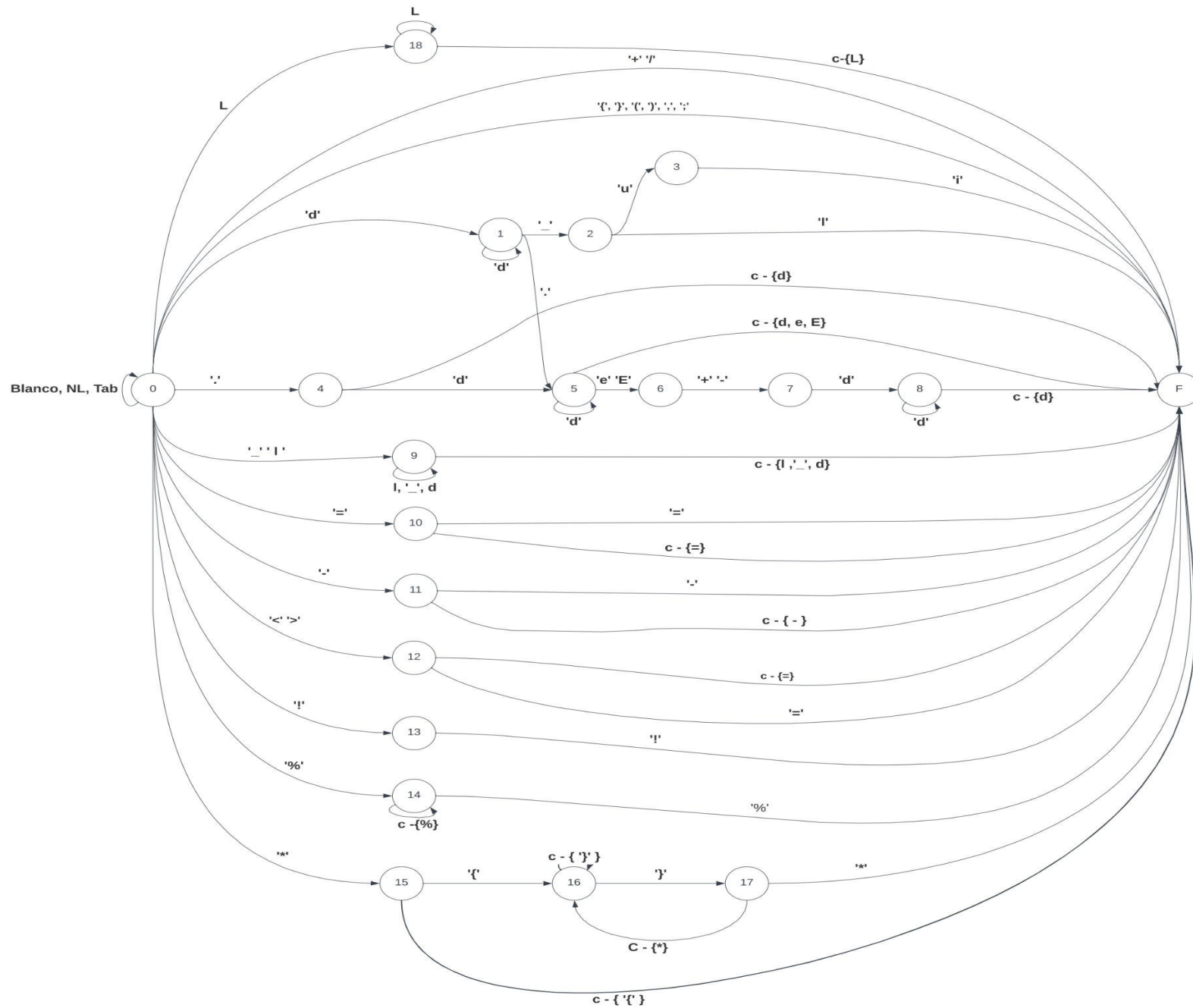
AnalizadorLexico

Esta es la clase principal del analizador léxico, la cual se encarga de generar los tokens y enviárselos al parser. La función principal dentro de la clase es proximoEstado. La misma al ser invocada se ejecuta, avanzando por diferentes estados hasta generar un token completo. Si el token no incumple ninguna regla ni contiene ningún error, el mismo es almacenado en la tabla de símbolos y su id es retornada para poder ser analizada por el analizador sintáctico. El analizador léxico es el encargado de utilizar las acciones semánticas dependiendo de lo indicado en la tabla de acciones semánticas y en la tabla de transición de estados.

Matriz de Acciones Semánticas:

	BL	TAB	NL	I	‘ ’	D	‘e’	‘E’	‘+’	‘-’	‘*’	‘/’	‘=’	‘!’	‘<’	‘>’	‘.’	‘:’	‘,’	PR	‘{’	‘}’	‘%’	‘(’	‘)’	‘!’	‘u’	‘i’	OT RO
0	AS0	AS0	AS0	AS1	AS1	AS1	AS1	AS1	AS2	AS1	AS1	AS2	AS1	AS1	AS1	AS1	AS1	AS2	AS2	AS1	AS2	AS2	AS1	AS2	AS2	AS1	AS1	AS1	ASE
1	ASE	ASE	ASE	ASE	AS1	AS1	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS1	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
2	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS5	AS1	ASE	ASE
3	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
4	AS4	AS4	AS4	AS4	AS4	AS1	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	ASE	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	ASE
5	AS6	AS6	AS6	AS6	AS6	AS1	AS1	AS1	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	ASE
6	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS1	AS1	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
7	ASE	ASE	ASE	ASE	ASE	AS1	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
8	AS6	AS6	AS6	ASE	ASE	AS1	ASE	ASE	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	ASE
9	AS3	AS3	AS3	AS1	AS1	AS1	AS1	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS1	AS1	AS1	ASE
10	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	ASE	ASE	AS4	AS4	AS7	ASE	ASE	ASE	AS4	ASE	ASE	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	ASE
11	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS7	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	ASE
12	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS7	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	ASE
13	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS7	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
14	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	ASE
15	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS1	AS4	AS4	AS4	AS4	AS4	AS4	ASE
16	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	ASE
17	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS9	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	ASE
18	AS8	AS8	AS8	AS8	AS1	AS8	AS8	AS1	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS8	AS1	AS8	AS8	AS8	AS8	AS8	AS8	AS8	ASE

Autómata De transición de Estados



Consideración y decisiones tomadas con respecto a los errores

Los errores Léxicos se manejan en esta etapa, exceptuando el rango de las constantes negativas, las cuales son tratadas en el análisis sintáctico ya que es imposible detectar si una constante es positiva o negativa en esta parte.

Los errores tratados son:

- Identificadores con una longitud mayor a la permitida: Los identificadores tienen un máximo de longitud de 20 caracteres. En caso de superarse esta cantidad, el token se trunca y se eliminan los últimos caracteres sobrantes. Esto es tomado como un warning ya que el código puede seguir compilando y es posible su ejecución pero es probable que no realice la funcionalidad que estaba planeada en un principio. Ya que al recortar el nombre, este puede quedar idéntico a otro Identificador y eso significa que estamos trabajando sobre una variable que no corresponde. También es posible que el código ejecute de manera correcta aunque se recorte el nombre, por esa razón se lo considera una advertencia y se le permite terminar de compilarla para luego ejecutarse y no terminar la compilación por error.
- Errores de rango en las constantes: Las constantes ya sean enteros largos, enteros sin signo o flotantes, tienen un rango tanto superior como inferior. Este mismo no puede ser superado. Si alguna constante tiene un rango mayor o menor, se considera un error y se añade a la lista de errores léxicos. Ya que no es posible reemplazar el valor como en los identificadores porque cambiar el valor de una constante si puede afectar de manera significativa el funcionamiento del programa, sea cual sea el mismo.
- Si se encuentra con FLOATidentificador, no se cuenta como error. El Analizador lo toma como confusión del programador y separa los tokens.

Cualquier error léxico es almacenado en la Lista de errores Léxicos que se encuentra en la clase Analizador Léxico. Al detectarse cualquiera de estos errores, el compilador debe seguir su curso, por lo tanto la sentencia en la que es detectado el error léxico se descarta, es decir que el AnalizadorLexico avanza caracteres hasta encontrarse con una ',' la cual utilizamos como valor de restablecimiento. Todo lo anterior se descarta y se sigue compilando a partir de ese punto. Esto puede ocasionar que en la etapa siguiente, es decir el analizadorSintactico, no se detecten correctamente las estructuras posteriores a la cual tuvo el error porque al descartar caracteres hasta encontrar una coma es probable que se haya descartado una parte de código que no contenía error. Es por ello que si se detecta un error Léxico, el analizador sintáctico pierda algunas sentencias posteriores al error y no las detecta correctamente.

Salida del Analizador Léxico

Para mostrar la salida del analizador léxico, tanto los tokens obtenidos como los errores, decidimos hacerlo al final de la ejecución del compilador. Mostramos la lista de tokens encontrados así como también los errores.

Ejemplo de salida del Léxico para el siguiente código:

```
{  
  FLOAT _numero4;_hola,  
  FLOAT _numero5,  
  
  LONG _numero6,  
  
  _numero6 = _numero4,  
  _numero6 = _numero4 + _hola,  
  
  VOID _fun1 (FLOAT _numero4){  
    LONG _dentro,  
    FLOAT _alto,  
    _alto = _dentro,  
    RETURN,  
  },  
}
```

La salida resultante es:

TOKENS DETECTADOS POR EL LEXICO:

```
[ FLOAT , 269 ]  
[ _numero6 , 257 ]  
[ ( , 40 ]  
[ _numero5 , 257 ]  
[ ) , 41 ]  
[ _numero4 , 257 ]  
[ + , 43 ]  
[ , , 44 ]  
[ _alto , 257 ]  
[ _hola , 257 ]  
[ RETURN , 272 ]  
[ _dentro , 257 ]  
[ { , 123 ]  
[ ; , 59 ]  
[ _fun1 , 257 ]  
[ VOID , 266 ]  
[ = , 61 ]  
[ } , 125 ]  
[ LONG , 267 ]
```

Errores Lexicos:

No hubo ningun error lexico

Analizador Sintactico

El objetivo de esta parte del trabajo era construir un Parser (Analizador Sintáctico) que utilice al Analizador Léxico creado en el Trabajo Práctico N° 1, y que reconozca un lenguaje con las siguientes características:

Programa constituido por un conjunto de sentencias, que pueden ser declarativas o ejecutables. Las sentencias declarativas pueden aparecer en cualquier lugar del código fuente, exceptuando los bloques de las sentencias de control.

Los elementos declarados sólo serán visibles a partir de su declaración (esto será chequeado en etapas posteriores).

El programa estará delimitado por llaves '{' y '}'. Cada sentencia debe terminar con coma ','.

Decisiones e implementación:

Para realizar el parser tuvimos que implementar la gramática siguiendo las pautas que nos brinda el TP2. Tratamos de llevar un orden en la gramática para facilitar su entendimiento y a su vez utilizar nombres claros para cada regla. También agregamos reglas con errores, ejemplo, una regla sin una coma al final. Si se detecta un error en el código, se almacena un mensaje de error que es mostrado al final de la ejecución del parser. Al momento de realizar la gramática no nos surgieron grandes inconvenientes y todos fueron solucionados probando y modificando la gramática hasta obtener un funcionamiento óptimo.

Lista de no terminales de la Gramática:

- bloque_sentencia_programa: El bloque que va a contener las sentencias ya sean declarativas o ejecutables
- sentencia_programa: Puede ser una sentencia ejecutable o declarativa
- sentencias_declarativas: Son las sentencias de declaración, puede ser por ejemplo, una variable, función, etc.
- sentencias_ejecutables: Son las sentencias que pueden ejecutarse y realizar cambios en el programa, por ejemplo, asignaciones, mensajes de salida, etc.
- declaracion_variables: Se declara una variable, el nombre y de que tipo de variable se trata
- declaracion_funciones: Declara la función. El formato es VOID luego nombre de funcion y por ultimo el bloque de la funcion
- tipo: Se utiliza para ver de qué tipo es la variable, si entero largo, entero sin signo o flotante
- list_de_variables: Utilizada para poder declarar más de una variable del mismo tipo sin la necesidad de estar aclarando el tipo a cada variable por separado, ejemplo, FLOAT _a; _b; _c,
- cuerpo_de_funcion: En esta regla se pueden definir dentro todas las sentencias que va a poseer la función.

- `sentencia_declarativa_especifica`: Son las sentencias declarativas que pueden ser utilizadas dentro de una función, ya que no son las mismas que se permiten en el cuerpo del programa.
- `sentencias_retorno`: es la que se utiliza al final de una función
- `declaracion_clases`: Sirve para declarar las clases, la forma que tiene que tener la declaración de la clase.
- `cuerpo_clase`: el cuerpo de clase es las sentencias que se permiten dentro de la declaración de una clase.
- `list_objts_clase`: Permite crear como en el caso de las variables una declaración del objeto de la clase sin la necesidad de colocarle el tipo a cada una por separado.
- `sentencia_ejecucion_funcion`: Son las sentencias ejecutables que se permiten dentro de una función.
- `sentencia_asignacion`: Sentencia que le asigna el valor a una variable, ya sea una constante o el retorno de una función.
- `valor_asignacion`: Puede ser una expresión aritmética
- `invocacion_funcion`: es posible invocar una función en cualquier parte del código.
- `sentencia_IF`: Es la sentencia ejecutable del bloque if, la forma que tiene que tener.
- `condicion_if_while`: Son las condiciones que se pueden colocar en un if o un while, ejemplo, mayor, menor, menor o igual, etc.
- `bloque_sentencias_ejecutables`
- `sentencias_salida`: Es el print que muestra por pantalla, en nuestro caso solo puede ser una cadena por lo especificado en el enunciado
- `sentencias_control`: es el bloque WHILE
- `sentencias_while_do`: Determina cómo debe declararse el bloque while, la forma que tiene que tener.
- `expr_aritmetico`: puede ser suma, resta o un término
- `término`: puede ser una multiplicación, división, factor o invocación_funcion
- `factor`: constante o un identificador
- `const`: puede ser positiva o negativa.

Manejo de errores del analizador sintáctico

Los errores considerados en esta etapa, como por ejemplo, la falta de un paréntesis, o la falta de una coma, etc. Son almacenados en una lista de errores dentro del parser.

Estos errores tienen la siguiente forma: Línea en la que ocurrió(para obtener esto utilizamos la variable Línea Actual del analizador léxico que lleva registro de por que línea de código nos encontramos) + una descripción del error encontrado. Estos errores son mostrados al final de la ejecución del parser

Salida el analizador Sintáctico

La salida del analizador sintáctico se realiza mientras se está ejecutando el parser. Cada vez que se detecta una regla nueva se imprime por pantalla la línea en la que se encontró, así como también un mensaje descriptivo de lo que se detectó. En caso de ser un error, como mencionamos anteriormente, este se almacena en una lista para ser mostrado una vez finalizado el parser.

Ejemplo de salidas del analizador sintáctico:

```
{  
  FLOAT _numero4;_hola,  
  FLOAT _numero5,  
  
  LONG _numero6,  
  
  _numero6 = _numero4,  
  _numero6 = _numero4 + _hola,  
  
  VOID _fun1 (FLOAT _numero4){  
    LONG _dentro,  
    FLOAT _alto,  
    _alto = _dentro,  
    RETURN,  
  },  
}
```

Para este código la salida es:

```
Linea: 2. Se reconocio un tipo FLOAT  
Linea: 2. Se reconocio una declaracion de variables  
Linea: 2. Se reconocio sentencia declarativa  
Linea: 3. Se reconocio un tipo FLOAT  
Linea: 3. Se reconocio una declaracion de variables  
Linea: 3. Se reconocio sentencia declarativa  
Linea: 5. Se reconocio un tipo LONG  
Linea: 5. Se reconocio una declaracion de variables  
Linea: 5. Se reconocio sentencia declarativa  
Linea: 7. Se reconocio un Identificador  
Linea: 7. Se reconocio una asignacion  
Linea: 7. Se reconocio sentencia Ejecutables  
Linea: 7. Se reconocio sentencia ejecutable  
Linea: 8. Se reconocio un Identificador  
Linea: 8. Se reconocio un Identificador  
Linea: 8. Se detecto una suma  
Linea: 8. Se reconocio una asignacion  
Linea: 8. Se reconocio sentencia Ejecutables  
Linea: 8. Se reconocio sentencia ejecutable  
Linea: 10. Se reconocio un tipo FLOAT  
Linea: 11. Se reconocio un tipo LONG  
Linea: 11. Se reconocio una declaracion de variables  
Linea: 12. Se reconocio un tipo FLOAT  
Linea: 12. Se reconocio una declaracion de variables  
Linea: 13. Se reconocio un Identificador  
Linea: 13. Se reconocio una asignacion  
Linea: 13. Se reconocio sentencia Ejecutables  
Linea: 14. Se reconocio RETURN  
Linea: 15. Se reconocio una funcion con parametro  
Linea: 15. Se reconocio sentencia declarativa  
Linea 16, Se reconocio el programa
```

ARRANCO EL PROGRAMA

Otro ejemplo es, esta vez con errores:

```
{
```

```
    IF(10_I > 8_I){
        _numero2 = _numero1,
        _hola = _chau,
    }ELSE{
        _perro = 10_I,
    } END_IF,
```

```
    IF(10_I > 8_I){
        _numero2 = _numero1,
        _hola = _chau,
    }{
        _perro = 10_I,
    } END_IF,
```

```
    (10_I > 8_I){
        _numero2 = _numero1,
        _hola = _chau,
    }ELSE{
        _perro = 10_I,
    } END_IF,
```

```
    IF(10_I > 8_I){
        _numero2 = _numero1,
        _hola = _chau,
    }{
        _perro = 10_I,
    },
```

```
    (10_I > 8_I){
        _numero2 = _numero1,
        _hola = _chau,
    }{
        _perro = 10_I,
    } END_IF,
```

```
    (10_I > 8_I){
        _numero2 = _numero1,
        _hola = _chau,
```



```

    }
        _perro = 10_I,
    },

    IF(){
        _numero2 = _numero1,
        _hola = _chau,
    }ELSE{
        _perro = 10_I,
    } END_IF,

    IF(10_I > 8_I){
        _numero2 = _numero1,
        _hola = _chau,
    }ELSE{ } END_IF,

    _hola = 10_I,

}

```

La salida es:

```

Linea: 3. Se reconocio una constante
Linea: 3. Se reconocio una constante
Linea: 3. Se reconocio una comparacion por mayor
Linea: 4. Se reconocio un Identificador
Linea: 4. Se reconocio una asignacion
Linea: 4. Se reconocio sentencia Ejecutables
Linea: 5. Se reconocio un Identificador
Linea: 5. Se reconocio una asignacion
Linea: 5. Se reconocio sentencia Ejecutables
Linea: 7. Se reconocio una constante
Linea: 7. Se reconocio una asignacion
Linea: 7. Se reconocio sentencia Ejecutables
Linea: 8. Se reconocio un IF_ELSE
Linea: 8. Se reconocio sentencia ejecutable
Linea: 10. Se reconocio una constante
Linea: 10. Se reconocio una constante
Linea: 10. Se reconocio una comparacion por mayor
Linea: 11. Se reconocio un Identificador
Linea: 11. Se reconocio una asignacion
Linea: 11. Se reconocio sentencia Ejecutables
Linea: 12. Se reconocio un Identificador
Linea: 12. Se reconocio una asignacion
Linea: 12. Se reconocio sentencia Ejecutables
Linea: 14. Se reconocio una constante
Linea: 14. Se reconocio una asignacion
Linea: 14. Se reconocio sentencia Ejecutables
Linea: 15. Se reconocio sentencia ejecutable
Linea: 17. Se reconocio una constante
Linea: 17. Se reconocio una constante
Linea: 17. Se reconocio una comparacion por mayor

```

Linea: 18. Se reconocio un Identificador
Linea: 18. Se reconocio una asignacion
Linea: 18. Se reconocio sentencia Ejecutables
Linea: 19. Se reconocio un Identificador
Linea: 19. Se reconocio una asignacion
Linea: 19. Se reconocio sentencia Ejecutables
Linea: 21. Se reconocio una constante
Linea: 21. Se reconocio una asignacion
Linea: 21. Se reconocio sentencia Ejecutables
Linea: 22. Se reconocio sentencia ejecutable
Linea: 25. Se reconocio una constante
Linea: 25. Se reconocio una constante
Linea: 25. Se reconocio una comparacion por mayor
Linea: 26. Se reconocio un Identificador
Linea: 26. Se reconocio una asignacion
Linea: 26. Se reconocio sentencia Ejecutables
Linea: 27. Se reconocio un Identificador
Linea: 27. Se reconocio una asignacion
Linea: 27. Se reconocio sentencia Ejecutables
Linea: 29. Se reconocio una constante
Linea: 29. Se reconocio una asignacion
Linea: 29. Se reconocio sentencia Ejecutables
Linea: 30. Se reconocio sentencia ejecutable
Linea: 32. Se reconocio una constante
Linea: 32. Se reconocio una constante
Linea: 32. Se reconocio una comparacion por mayor
Linea: 33. Se reconocio un Identificador
Linea: 33. Se reconocio una asignacion
Linea: 33. Se reconocio sentencia Ejecutables
Linea: 34. Se reconocio un Identificador
Linea: 34. Se reconocio una asignacion
Linea: 34. Se reconocio sentencia Ejecutables
Linea: 36. Se reconocio una constante
Linea: 36. Se reconocio una asignacion
Linea: 36. Se reconocio sentencia Ejecutables
Linea: 37. Se reconocio sentencia ejecutable
Linea: 39. Se reconocio una constante
Linea: 39. Se reconocio una constante
Linea: 39. Se reconocio una comparacion por mayor
Linea: 40. Se reconocio un Identificador
Linea: 40. Se reconocio una asignacion
Linea: 40. Se reconocio sentencia Ejecutables
Linea: 41. Se reconocio un Identificador
Linea: 41. Se reconocio una asignacion
Linea: 41. Se reconocio sentencia Ejecutables
Linea: 43. Se reconocio una constante
Linea: 43. Se reconocio una asignacion
Linea: 43. Se reconocio sentencia Ejecutables
Linea: 44. Se reconocio sentencia ejecutable
Linea: 47. Se reconocio un Identificador
Linea: 47. Se reconocio una asignacion
Linea: 47. Se reconocio sentencia Ejecutables
Linea: 48. Se reconocio un Identificador
Linea: 48. Se reconocio una asignacion
Linea: 48. Se reconocio sentencia Ejecutables
Linea: 50. Se reconocio una constante
Linea: 50. Se reconocio una asignacion
Linea: 50. Se reconocio sentencia Ejecutables

```
Linea: 51. Se reconocio sentencia ejecutable
Linea: 54. Se reconocio una constante
Linea: 54. Se reconocio una constante
Linea: 54. Se reconocio una comparacion por mayor
Linea: 55. Se reconocio un Identificador
Linea: 55. Se reconocio una asignacion
Linea: 55. Se reconocio sentencia Ejecutables
Linea: 56. Se reconocio un Identificador
Linea: 56. Se reconocio una asignacion
Linea: 56. Se reconocio sentencia Ejecutables
Linea: 57. Se reconocio sentencia ejecutable
Linea: 59. Se reconocio una constante
Linea: 59. Se reconocio una asignacion
Linea: 59. Se reconocio sentencia Ejecutables
Linea: 59. Se reconocio sentencia ejecutable
Linea 61, Se reconocio el programa
ARRANCO EL PROGRAMA
Linea: 15. Error sintactico . Falta palabra ELSE
Linea: 22. Error sintactico . Falta palabra IF
Linea: 30. Error sintactico . Falta palabras ELSE Y END_IF
Linea: 37. Error sintactico . Falta palabras IF y ELSE
Linea: 44. Error sintactico . Falta palabras IF, ELSE Y END_IF
Linea: 51. Error sintactico . Falta condicion
Linea: 57. Error sintactico . Falta bloque sentencias ejecutables
```

Errores y consideraciones en estas etapas

- Dentro de las funciones solo es posible realizar sentencias declarativas ya que el enunciado no menciona nada de sentencias ejecutables.
- Los identificadores no pueden contener letras mayusculas
- Es necesario que los archivos txt sean UNIX(LF).

Ejecución del archivo Jar

El archivo .jar fue creado para 2 versiones distintas de jdk, para la 17.0.8 y para la 11. Ambos archivos están subidos al drive para que sea utilizado el correspondiente.

VERSIÓN DEL JDK:

JAVA_RUNTIME_VERSION="17.0.8+9-LTS-211"

JAVA_VERSION="17.0.8"

JAVA_VERSION_DATE="2023-07-18"

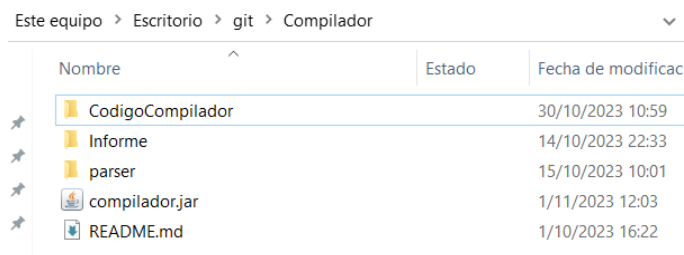
VERSIÓN DEL ECLIPSE:

Version: 2023-09 (4.29.0)

Para poder ejecutar el jar es necesario seguir los siguientes pasos:

1°) abrir una consola de comandos de windows(cmd)

2°) pararte en la carpeta en la que se encuentra el archivo compilador.jar utilizando el comando cd "direccion de la carpeta" (en este caso es en la carpeta ../compilador)



Este equipo > Escritorio > git > Compilador			
Nombre	Estado	Fecha de modificac	
CodigoCompilador		30/10/2023 10:59	
Informe		14/10/2023 22:33	
parser		15/10/2023 10:01	
compilador.jar		1/11/2023 12:03	
README.md		1/10/2023 16:22	

3°) Una vez parado en la carpeta ejecutamos el siguiente comando:

"java -jar compilador.jar"

4°) En caso de que no ocurra ningún error tendrías que aparecer un mensaje en la consola de la siguiente forma:

```
La carpeta a partir de la cual el compilador toma el valor ingresado es \CodigoCompilador\src\Testeos\
Ejemplo de dato a ingresar: [CPP_general.txt] o Sintactico\CPP_general.txt
```

```
Ingresa la direccion del archivo a compilar:
```

La manera de ingresar la dirección de un código para ser compilado es, por ejemplo, si el archivo txt se encuentra en

"C:\Users\rolus\OneDrive\Escritorio\git\Compilador\CodigoCompilador\src\Testeos\lexico.LX_pruebaConErrores.txt", El dato que debo ingresar al compilador es: lexico\LX_pruebaConErrores.txt.

Para probar cada código es necesario volver a ejecutar los pasos anteriores.

Salida general del compilador

Al compilar un código(Ejemplo: LX_pruebaLarga.txt) la salida que no proporciona el compilador es:

Análisis sintáctico:

```
Linea: 3. Se reconocio sentencia declarativa
Linea: 4. Se reconocio un tipo FLOAT
Linea: 4. Se reconocio una declaracion de variables
Linea: 4. Se reconocio sentencia declarativa
Linea: 7. Se reconocio un Identificador
Linea: 7. Se reconocio un Identificador
Linea: 7. Se detecto una suma
Linea: 7. Se reconocio un Identificador
Linea: 7. Se detecto una suma
Linea: 7. Se reconocio una asignacion
Linea: 7. Se reconocio sentencia Ejecutables
Linea: 8. Se reconocio un Identificador
Linea: 8. Se reconocio una constante
Linea: 8. Se reconocio una comparacion por distinto?
Linea: 9. Se reconocio un Identificador
Linea: 9. Se reconocio una constante
Linea: 9. Se detecto una resta
Linea: 9. Se reconocio una asignacion
Linea: 9. Se reconocio sentencia Ejecutables
Linea: 11. Se reconocio un Identificador
Linea: 11. Se reconocio un Identificador
Linea: 11. Se detecto una suma
Linea: 11. Se reconocio una asignacion
Linea: 11. Se reconocio sentencia Ejecutables
Linea: 12. Se reconocio un IF_ELSE
Linea: 13. Se reconocio RETURN
Linea: 14. Se reconocio una funcion sin parametro
Linea: 14. Se reconocio sentencia declarativa
Linea: 16. Se reconocio una invocacion a una funcion sin parametro
Linea: 16. Se reconocio un Identificador
Linea: 16. Se detecto una resta
Linea: 16. Se reconocio una asignacion
Linea: 16. Se reconocio sentencia Ejecutables
Linea: 16. Se reconocio sentencia ejecutable
Linea: 18. Se reconocio una cadena
Linea: 18. Se reconocio sentencia ejecutable
Linea: 22. Se reconocio un Identificador
Linea: 22. Se reconocio una constante
Linea: 22. Se reconocio una comparacion por menor o igual
Linea: 23. Se reconocio un Identificador
Linea: 23. Se reconocio una constante
Linea: 23. Se detecto una resta
Linea: 23. Se reconocio una asignacion
Linea: 23. Se reconocio sentencia Ejecutables
Linea: 24. Se reconocio una sentencia While
Linea: 24. Se reconocio sentencia ejecutable
Linea 28, Se reconocio el programa
ARRANCO EL PROGRAMA
```

Errores Sintacticos:

```
Errores Sintacticos:  
NO HUBO ERRORES SINTACTICOS
```

Tokens Detectados por el Léxico:

```
TOKENS DETECTADOS POR EL LEXICO:  
[ PRINT , 264 ]  
[ <= , 275 ]  
[ FLOAT , 269 ]  
[ _numero1 , 257 ]  
[ END_IF , 263 ]  
[ f1 , 257 ]  
[ DO , 271 ]  
[ RETURN , 272 ]  
[ 1000_1 , 258 ]  
[ 100_1 , 258 ]  
[ var5 , 257 ]  
[ ELSE , 262 ]  
[ 10_1 , 258 ]  
[ var2 , 257 ]  
[ var1 , 257 ]  
[ IF , 260 ]  
[ 1_1 , 258 ]  
[ LONG , 267 ]  
[ !! , 277 ]  
[ ( , 40 ]  
[ ) , 41 ]  
[ + , 43 ]  
[ , , 44 ]  
[ - , 45 ]  
[ _var3 , 257 ]  
[ WHILE , 270 ]  
[ { , 123 ]  
[ ; , 59 ]  
[ VOID , 266 ]  
[ UINT , 268 ]  
[ _var4 , 257 ]  
[ = , 61 ]  
[ } , 125 ]  
[ %el valor de _var3 es% , 259 ]
```

Contenido de la Tabla de símbolos:

```
Tabla de Simbolos. HashTable<String, Simbolo>:  
Muestra '[Clave]' -----> Simbolo: Lexema , ID  
Clave del hashtable: [%el valor de _var3 es%] -----> SIMBOLO: Lexema: [%el valor de _var3 es%] ID: [259]  
Clave del hashtable: [_var4] -----> SIMBOLO: Lexema: [_var4] ID: [257]  
Clave del hashtable: [_var3] -----> SIMBOLO: Lexema: [_var3] ID: [257]  
Clave del hashtable: [var5] -----> SIMBOLO: Lexema: [var5] ID: [257]  
Clave del hashtable: [f1] -----> SIMBOLO: Lexema: [f1] ID: [257]  
Clave del hashtable: [var2] -----> SIMBOLO: Lexema: [var2] ID: [257]  
Clave del hashtable: [var1] -----> SIMBOLO: Lexema: [var1] ID: [257]  
Clave del hashtable: [_numero1] -----> SIMBOLO: Lexema: [_numero1] ID: [257]  
Clave del hashtable: [1_1] -----> SIMBOLO: Lexema: [1_1] ID: [258]  
Clave del hashtable: [1000_1] -----> SIMBOLO: Lexema: [1000_1] ID: [258]  
Clave del hashtable: [10_1] -----> SIMBOLO: Lexema: [10_1] ID: [258]  
Clave del hashtable: [100_1] -----> SIMBOLO: Lexema: [100_1] ID: [258]
```

Errores Léxicos:

```
Errores Lexicos:  
No hubo ningun error lexico
```

Mapa de palabras reservadas:

```
Mapa de palabras reservadas (STRING , ID):  
[ PRINT , 264 ]  
[ == , 276 ]  
[ !! , 277 ]  
[ -- , 278 ]  
[ <= , 275 ]  
[ FLOAT , 269 ]  
[ END_IF , 263 ]  
[ TOF , 273 ]  
[ CLASS , 265 ]  
[ DO , 271 ]  
[ RETURN , 272 ]  
[ CTE , 258 ]  
[ ELSE , 262 ]  
[ CADENA , 259 ]  
[ THEN , 261 ]  
[ WHILE , 270 ]  
[ ID , 257 ]  
[ VOID , 266 ]  
[ IF , 260 ]  
[ UINT , 268 ]  
[ LONG , 267 ]  
[ >= , 274 ]
```

Código a compilar mostrando el numero de linea:

```
Codigo Analizado:
LINEA 1. {
LINEA 2.     LONG var1;var2,
LINEA 3.     UINT _var3,
LINEA 4.     FLOAT _var4;var5,
LINEA 5.
LINEA 6.     VOID f1(){
LINEA 7.         var1 = _numero1+var5+var2,
LINEA 8.         IF(var1 !! 100_1){
LINEA 9.             var1 = var1 - 10_1,
LINEA 10.        }ELSE{
LINEA 11.            var1 = var1 + _numero1,
LINEA 12.        }END_IF,
LINEA 13.        RETURN,
LINEA 14.    },
LINEA 15.
LINEA 16.    _var3 = f1() - _var3,
LINEA 17.
LINEA 18.    PRINT %el valor de _var3 es%,
LINEA 19.
LINEA 20.    *{esto tiene que llegar hasta aca}*
LINEA 21.
LINEA 22.    WHILE(_var3 <= 1000_1)DO{
LINEA 23.        _var3 = _var3 - 1_1,
LINEA 24.    },
LINEA 25.
LINEA 26.
LINEA 27.
LINEA 28. }
```

Conclusiones

Durante el desarrollo de estas dos etapas pudimos comprender el funcionamiento del léxico y el sintáctico de un compilador. Como un compilador analiza un código y a partir de este genera tokens e identifica sentencias que luego van a ser usadas en etapas posteriores. Como son detectados los errores léxicos y sintácticos y una idea general de cómo funciona un compilador hasta este punto. También nos encontramos con dificultades las cuales pudimos resolver pensando las cosas de diferente manera. A como a partir de un código el compilador se encarga de analizar cada palabra y verificar que todo sea correcto.