Compilador.

Trabajo Diseño de Compiladores - Etapa 1.

Universidad Nacional del Centro de la Provincia de Buenos Aires.

07/10/2024

Alumnos: Lapenta Nicolas; Girardi Emiliana; Román Nahuel.

Tutor: José Fernández León

INTRODUCCIÓN

Un **compilador** es un programa de computadora que traduce un lenguaje a otro. Toma como entrada un programa escrito en su lenguaje fuente y produce un programa equivalente escrito en su lenguaje objetivo.

Las fases de la compilación son:

- **Análisis léxico:** Se analiza el código fuente para identificar las unidades básicas del lenguaje, llamadas tokens.
- **Análisis sintáctico:** Se organiza la secuencia de tokens en una estructura lógica llamada árbol de sintaxis que respeta la gramática del lenguaje.
- **Análisis semántico:** Verifica que las instrucciones del programa tengan sentido desde el punto de vista lógico.
- **Optimización de código:** Se mejora el código intermedio para que sea más eficiente en términos de tiempo de ejecución o uso de recursos.
- **Generación de código:** Traduce el código intermedio optimizado a un lenguaje de máquina o a un lenguaje ensamblador específico para el procesador de la computadora.
- **Ensamblaje y vinculación:** Convierte el código ensamblador a código binario ejecutable y combina distintos módulos (si los hay) en un único archivo ejecutable.

Para este trabajo se realizó el diseño de las etapas de análisis léxico y sintáctico en lenguaje Java de un compilador del lenguaje propuesto por la cátedra. Dicho lenguaje debe contemplar las siguientes características:

- **Identificadores** cuyos nombres pueden tener hasta 15 caracteres de longitud. El primer carácter sólo puede ser una letra, y el resto pueden ser letras, dígitos y "_".
  La primera letra de un identificador podrá definir el tipo de un identificador:
  Los identificadores que comienzan con u, v o w serán de tipo entero sin signo d.
  Para single, los identificadores que comiencen con s serán de tipo single.

- **Constantes de tipo single.** Números reales con signo y parte exponencial. La parte exponencial puede estar ausente. Si está presente, el exponente comienza con la letra “s” (minúscula) y el signo del exponente es opcional. Su ausencia implica signo positivo para el exponente. La parte entera, la parte decimal, y el ‘.’ son obligatorios.
  Rango a considerar:
  1.17549435s-38 < x < 3.40282347s+38
  -3.40282347s+38 < x < -1.17549435s-38 0.0
- **Constantes de tipo unsigned**, con valores entre 0 y 216 – 1.
- **Constantes de tipo octal.** Comenzando con un 0, serán una secuencia de dígitos que pueden ir de 0 a 7. Rango a considerar: 00 a 0177777.
- **Palabras reservadas:** REPEAT, IF, THEN, ELSE, BEGIN, END, END_IF, OUTF, TYPEDEF, FUN, RET, GOTO, TRIPLE, OCTAL, TIPO_UNSIGNED, TIPO_SINGLE, TIPO_OCTAL, UNTIL.
- **Operadores aritméticos:** “+”, “-” , “*”, “/”.
- **Operador de asignación:** “:=”
- **Comparadores:** “>=”, “<=”, “>”, “<”, “=”, “!=” “(”, “)”, “,”, “.” y “;”
- **Cadenas de multilínea** que pueden ser impresas por pantalla mediante la palabra reservada OUTF.
  Ejemplo: [¡Hola
  mundo!]
- **Comentarios de 1 línea:** Comentarios que comiencen con “##” y terminen con el fin de línea.
- Un **programa** se define con un nombre y luego un conjunto de sentencias declarativas y ejecutables delimitadas por BEGIN y END.
- Una **sentencia declarativa** puede ser una declaración de variable, una definición de una lista de variables, una definición de función o una declaración del tipo triple.
- Una **sentencia ejecutable** puede ser una sentencia if, un ciclo REPEAT UNTIL, una invocación a función, una asignación, una sentencia goto, o una impresión por pantalla OUTF.

- Una **definición de función** se define como:
  <tipo> FUN ID (<parametro>) BEGIN <cuerpo_ de_la_funcion> END
  Donde cuerpo_de_la_funcion es un conjunto de sentencias declarativas que obligatoriamente debe terminar con una sentencia de retorno RET (<expresion>).
- Una **sentencia if** tiene la siguiente estructura:
  IF (condicion) THEN <bloque_sentencias_ejecutables> ELSE <bloque_sentencias_ejecutables> END_IF ;
  (admitiendo también una sentencia sin bloque ELSE).
- Un **ciclo REPEAT UNTIL** tiene la siguiente estructura:
  REPEAT <bloque_de_sentencias_ejecutables> UNTIL (<condicion>);
- Una **condición** es una comparación de una o varias expresiones aritmeticas.
- Una **invocación a función** se define como: ID(<parametro>). Donde un parámetro es una expresión aritmética.
- Una **impresión por pantalla** tiene la siguiente estructura:
  OUTF(<cadena>);
  o
  OUTF(<expresion>);
- Se pueden declarar **tipos triple** de un tipo determinado y además variables de dicho tipo definido por el usuario.
  Por ejemplo: typedef triple <uinteger> tint;
- Una **sentencia GOTO** se define como:

  GOTO etiqueta@.

- Se pueden definir **expresiones aritméticas** cuyos términos sean multiplicaciones o divisiones. A su vez los operandos pueden ser constantes, identificadores o invocaciones a funciones.

A continuación se detalla el proceso de desarrollo de ambas etapas.

ANALIZADOR LÉXICO

El análisis léxico es la primera fase de la compilación cuyo objetivo es leer el programa fuente y prepararlo para el análisis sintáctico.
Un token es una secuencia de caracteres que forman una unidad significativa en el contexto del lenguaje. Esta etapa se encarga de identificar tokens, y notificar errores y warnings.

Se diseñó un analizador léxico que es capaz de identificar los tokens propios del lenguaje en el programa fuente. Para lograr este cometido se planificó un autómata finito para representar de forma correcta la transición de estados. A su vez cada transición cuenta con una acción semántica que hace a la funcionalidad del código.

El autómata está representado por el siguiente gráfico en Miro: https://miro.com/app/board/uXjVKhLgPis=/?share_link_id=359965663467.

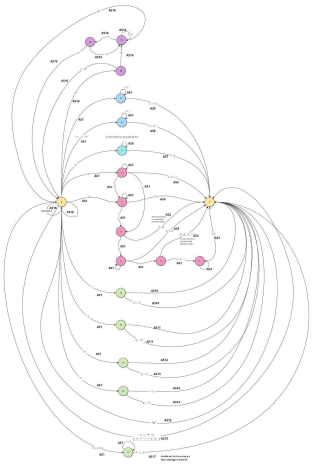


Figura 1. Autómata Finito.

Estados del Autómata

Se plantearon diecisiete estados distintos los cuales se agrupan para identificar distintos tipos de tokens:

- **Octal:** El estado *cero* es el encargado de detectar un octal.
- **Single:** El conjunto de estado *uno, dos, tres, cuatro y cinco*, son los encargados de detectar el tipo single. Dentro de este conjunto el estado dos y cuatro manejan errores de constante mal escrita a través de sus acciones semánticas, agregando el carácter faltante.
- **Multilínea:** El estado *seis* es el encargado de detectar una constante multilínea.
- **Identificador Unsigned:** El estado *siete* es el encargado de detectar un identificador de tipo unsigned. El identificador debe comenzar con "*u*", "*w*", o "*v*", puede contener cualquier letra, números y el símbolo "_".
- **Identificador Single:** El estado *ocho* es el encargado de detectar un identificador de tipo single. El identificador debe comenzar con "*s*", puede contener cualquier letra, números y el símbolo "_".
- **Comentarios:** El conjunto de estados *diez, once y doce*, son los encargados de manejar los comentarios y descartarlos.
- **Asignaciones:** El estado *trece* reconoce las asignaciones y en el caso de que falte el carácter '=', lo agrega a través de su acción semántica y pasa al estado final.
- **Diferencia:** El estado *catorce* reconoce la diferencia y en el caso de que falte el carácter '=', lo agrega a través de su acción semántica y pasa al estado final.
- **Mayor y Mayor o igual:** El estado *quince* es el encargado de reconocer el token mayor o el token mayor o igual, pudiendo diferenciarlos.
- **Menor y Menor o igual:** El estado *dieciséis* es el encargado de reconocer el token menor o el token menor o igual.
- **Identificador:** El estado *diecisiete* reconoce un identificador. Para ser un identificador un token debe comenzar con una letra y posteriormente puede contener cualquier letra, números y el símbolo "_".
- **Caracteres inválidos y Saltos de línea:** al llegar un carácter invalido o un salto de línea desde el estado *cero*, se continuará en dicho estado.

Matriz de Transición de Estados.

La siguiente matriz indica a qué estado debe dirigirse el Léxico ante la llegada de un carácter, teniendo en cuenta el estado en el que se encuentra actualmente. El estado 50 corresponde al estado final.

Estado	d	o	0	[	u	v	w	s	#	>	<	!	:	.	+	-	*	/	=	(	)	,	;	]	l	_	/r	/n	invalid
0	1	1	6	7	8	8	8	9	10	15	16	14	13	50	50	50	50	50	50	50	50	50	50	50	17	50	50	50	0
1	1	50	1	50	50	50	50	50	50	50	50	50	50	2	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
2	3	50	3	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
3	3	50	3	50	50	50	50	4	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
4	5	50	5	50	50	50	50	50	50	50	50	50	50	50	5	5	50	50	50	50	50	50	50	50	50	50	50	50	50
5	5	50	5	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
6	1	6	1	50	50	50	50	50	50	50	50	50	50	2	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	50	7	7	7	7	7
8	8	50	8	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	8	8	50	50	50
9	9	50	9	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	9	9	50	50	50
10	0	0	0	0	0	0	0	0	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	12	11	11
12	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	0	11
13	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
14	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
15	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
16	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
17	17	17	17	50	17	17	17	17	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	17	17	50	50	50

Para una mejor visualización de la matriz: Matriz Transición de Estados.

Tabla de Símbolos.

La *tabla de símbolos* es un atributo del analizador léxico y es una estructura transversal a todo el compilador. Su objetivo principal es almacenar información adicional de cada token de forma dinámica. Cada parte del compilador tiene acceso a la tabla de símbolos y es capaz de modificarla.

Dentro del código la tabla de símbolos es representada a través de la estructura dinámica *hashMap* que provee Java, donde la clave es el *lexema* (de tipo String) y el valor es otra estructura dinámica (*ArrayList*) que brinda el lenguaje y se utiliza para guardar atributos de cada token en formato Integer. Esta estructura se encuentra dentro de una clase llamada TablaSimbolos.

Las palabras reservadas se cargan automáticamente al momento de inicializar la clase y los valores numéricos de cada token se condicen con los especificados por el analizador sintáctico, proveídos por YACC.

Los tipos de identificadores de variables están representados a través de valores enteros y se utilizará esta convención hasta el final del compilador, siendo:

- UNSIGNED: 1
- SINGLE: 2

- OCTAL: 3
- MULTILINEA: 4

Acciones Semánticas.

Una acción semántica es un fragmento de código en el lenguaje del compilador, en este caso Java, que se asocia a las transiciones, con el objetivo de no sobrecargar el autómata de estados.

En esta sección se describen las acciones semánticas implementadas y su función.

Cada acción semántica extiende la clase abstracta *accionSemántica que* contiene un método "*ejecutar*" el cual recibe por parámetro un objeto de tipo *léxico* y un carácter que indica el *último carácter leído*. Este método se encargará de ir construyendo el token y cuando sea necesario agregarlo a la tabla de símbolos junto con alguno de sus atributos.
A su vez, se encarga de devolver el número de token correspondiente cuando debe ser entregado, es decir, en las transiciones al estado final.

```
public abstract Optional<Integer> ejecutar(Character caracterActual, Lexico
                                           lexico)
```

Acción semántica 1

- Agrega el *último carácter leído* al atributo *token* del *léxico*
- Indica que se debe leer el siguiente carácter.

Acción semántica 2

- Detecta una constante de tipo single mal escrita y la corrige.
- Añade la constante leída a la tabla de símbolos (si no está) junto con el número de token correspondiente.
- Hace un set del lexema de la constante a yylval.
- Retorna el token de la constante.

Acción semántica 3

- Detecta una constante de tipo single.
- Añade la constante leída a la tabla de símbolos (si no está) junto con el número de token correspondiente.
- Hace un set del lexema de la constante a yylval.
- Retorna el token de la constante.

Acción semántica 4

- Detecta una constante de tipo entero unsigned.
- Corrobora el rango de la constante y la trunca si es necesario.
- Añade la constante leída a la tabla de símbolos (si no está) junto con el número de token correspondiente.
- Hace un set del lexema de la constante a yylval.
- Retorna el token de la constante.

Acción semántica 5

- Detecta una constante de tipo octal.
- Corrobora el rango de la constante y la trunca si es necesario.
- Añade la constante leída a la tabla de símbolos (si no está) junto con el número de token correspondiente.
- Hace un set del lexema de la constante a yylval.
- Retorna el token de la constante.

Acción semántica 6

- Agrega el carácter leído al token, ignorando los saltos de línea, ya que se está reconociendo una cadena multilínea.
- Indica que se debe leer el siguiente carácter.
- No retorna token.

Acción semántica 7

- Hace un set del lexema de la cadena a yylval, ignorando el corchete que cierra ‘]’.
- Retorna el token MULTILÍNEA.

Acción semántica 8

- Reconoce un identificador de tipo unsigned, agregándolo a la tabla de símbolos si no lo está, junto con el número de token y el descriptor numérico de tipo unsigned.
- Hace un set del lexema del identificador a yylval.
- Retorna el token ID.

Acción semántica 9

- Reconoce un identificador de tipo single, agregándolo a la tabla de símbolos si no lo está, junto con el número de token y el descriptor numérico de tipo unsigned.
- Hace un set del lexema del identificador a yylval.
- Retorna el token ID.

Acción semántica 10

- Reconoce una asignación que puede ser completa (:=) o incompleta (:).
- Si la asignación es incompleta le agrega ‘=’, lanzando warning.
- Agrega la asignación a la tabla de símbolos con su número de token correspondiente (si no lo esta).

- Hace un set del lexema “:=” a yylval.
- Retorna el token ASIGNACION.

Acción semántica 11

- Reconoce una inequidad que puede ser completa (!=) o incompleta (!).
- Si la inequidad es incompleta le agrega ‘=’, lanzando warning.
- Agrega la inequidad a la tabla de símbolos con su número de token correspondiente (si no lo esta).
- Hace un set del lexema “!=” a yylval.
- Retorna el token DISTINTO.

Acción semántica 12

- Reconoce un comparador MAYORIGUAL y lo agrega a la tabla de símbolos con su número de token correspondiente (si no lo esta).
- Indica que se debe leer el siguiente carácter.
- Hace un set del lexema “>=” a yylval.
- Retorna el token MAYORIGUAL.

Acción semántica 13

- Reconoce un comparador ‘>’ o ‘<’ y devuelve su código ascii.

Acción semántica 14

- Reconoce un comparador MENORIGUAL y lo agrega a la tabla de símbolos con su número de token correspondiente (si no lo esta).
- Indica que se debe leer el siguiente carácter.
- Hace un set del lexema “<=” a yylval.
- Retorna el token MENORIGUAL.

Acción semántica 15

- Devuelve el código ascii del carácter actual, debido a que se reconoce un token de un solo carácter.
- Indica que se debe leer el siguiente carácter.

Acción semántica 16

- Lee el siguiente carácter.
- Retorna nulo.

Acción semántica 17

- Reconoce un identificador que no es de ningún tipo específico.
- Lo añade a la tabla de símbolos si no está, junto con su número de token.
- Hace un set del lexema del identificador a yylval.
- Retorna el token ID.

Acción semántica 18

- Reconoce un carácter invalido y lo elimina, lanzando warning.
- Indica que se debe leer el siguiente carácter.
- Retorna vacío.

Matriz de Acciones Semánticas.

Para modelar qué acción semántica se debe abordar en cada transición de estado utilizamos una matriz en donde las columnas indican el símbolo actual y las filas el estado actual. El contenido de cada celda hace alusión a la acción semántica.

Estado	d	o	0	[	u	v	w	s	#	>	<	!	:	.	+	-	*	/	=	(	)	,	;	]	l	_	/r	/n	invalid
0	1	1	1	16	1	1	1	1	16	1	1	1	1	15	15	15	15	15	15	15	15	15	15	15	1	18	16	16	18
1	1	1	1	4	4	4	4	4	4	4	4	4	4	1	4	4	4	4	4	4	4	4	4	4	4	4	4	4	16
2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	16
3	1	3	3	3	3	3	3	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	16
4	1	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	2	2	2	2	2	2	2	2	2	2	2	2	16
5	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	16
6	1	1	5	5	5	5	5	5	5	5	5	5	5	1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	16
7	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	6	6	6	6	16
8	1	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	1	1	8	8	16
9	1	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	1	1	9	9	16
10	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
11	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
12	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
13	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	16
14	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	16
15	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	12	13	13	13	13	13	13	13	13	13	16
16	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	14	13	13	13	13	13	13	13	13	13	16
17	1	17	17	17	1	1	1	1	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	1	1	17	17	16

Para una mejor visualización de la matriz: Matriz de Acción Semántica

Manejo de las Matrices

Se mapearon las matrices de transición de estados y acciones semánticas a una misma matriz de tipo Par, donde Par es una clase que contiene un dato int para el estado y un objeto del tipo AccionSemantica.

```
public class Par {
    private int estado;
    private AccionSemantica as;
}
```

El mapeo se realizó con las librerías de Java BufferedReader y FileReader, donde cada estado se convierte en un dato de tipo int y cada número de acción semántica se convierte en una nueva acción semántica del número que indica. Ambos datos instancian a un objeto Par que es el que se coloca en la celda de la matriz correspondiente.

Al momento en que se lee un archivo y se comienza a construir el token, por cada símbolo nuevo se accede a un método que hace un mapping y devuelva la columna que debe ser accedida en la matriz mencionada anteriormente.

Manejo de Errores y Warning

El manejo de errores léxicos es una parte crucial en la fase de análisis léxico de un

compilador. Los errores léxicos ocurren cuando el analizador encuentra secuencias de caracteres que no forman tokens válidos en el lenguaje fuente. Este tipo de errores puede incluir desde caracteres no reconocidos hasta patrones mal formados que interrumpen el flujo normal del análisis.

Para el manejo de errores se utilizaron notificaciones de error y notificaciones de warning junto con las acciones necesarias para resolverlos en las acciones semánticas correspondientes.

Error constante mal escrita:

En la acción semántica 2 se maneja el posible error de tener una constante de tipo single mal escrita. Por ejemplo cuando se ingresa el número 25. o el número 50.0s, en ambos casos se notifica al usuario el ingreso de una constante mal escrita y se les agrega el número 0. El número 25. al cual le falta su parte decimal se le agrega por defecto 0 quedando como resultado un número single válido 25.0 y al número 50.0s al cual le falta indicar su exponente se toma al número 0 como su exponente por defecto resultando 50.0s0.

Warning entero fuera de rango:

En la acción semántica 4 se soluciona la posibilidad de que el número entero ingresado este fuera de rango, Si el número ingresado es mayor al máximo número entero representable denominado como límite, Se devuelve este último como el número ingresado.

Warning octal fuera de rango:

En la acción semántica 5 se tratan los números octales.

Si un número octal es mayor al límite el cual es una variable que contiene el máximo valor representable de los números octales se llama a la función truncaEntero en la cual se hace un and bit a bit con el valor hexadecimal 0x3FFFF, esto lo que logra es truncar el número octal y quedarse con los 18 bits de menor peso.

Warning asignación incompleta:

Cuando el compilador encuentra el carácter ‘:’ sin otro carácter de relevancia se asume

que es una asignación a la cual le falta el carácter ‘=’.

En la acción semántica 10 se agrega el carácter ‘=’ y se le notifica al usuario un warning de asignación incompleta junto con el número de línea en la que esto sucede.

Warning inequidad incompleta:

Similar a la asignación incompleta, cuando hay un ‘!’ La acción semántica 11 se encarga de agregar el carácter ‘=’ y notificar al usuario la inequidad incompleta.

Warning carácter invalido:

Cuando el analizador léxico detecta un carácter que no es válido, La acción semántica 18 se encarga de notificarle al usuario mediante un warning que el carácter no es válido y que ha sido eliminado y continúa con la lectura de caracteres.

Decisiones de Implementación

Se tomó la decisión de utilizar el patrón singleton para el analizador léxico, debido a que resulta útil tener una única instancia para utilizarla dentro del parser o analizador sintáctico, cuando se requiera llamar a algún método del mismo.

Si bien los rangos de las constantes de tipo unsigned y octal son verificados en la etapa del léxico, no ocurre lo mismo con las constantes de tipo single, que también pueden ser negativas. Por esto, se tomó la decisión de verificar su rango y truncarse cuando sea necesario en la etapa del analizador sintáctico, que es cuando puede determinarse si una constante positiva precedida por un signo negativo, es una constante negativa.

ALIZADOR SINTÁCTICO

El *analizador sintáctico* es una parte fundamental del compilador que se encarga de solicitar tokens al *analizador léxico* y agruparlos en frases gramaticales que el compilador usará en las siguientes etapas.

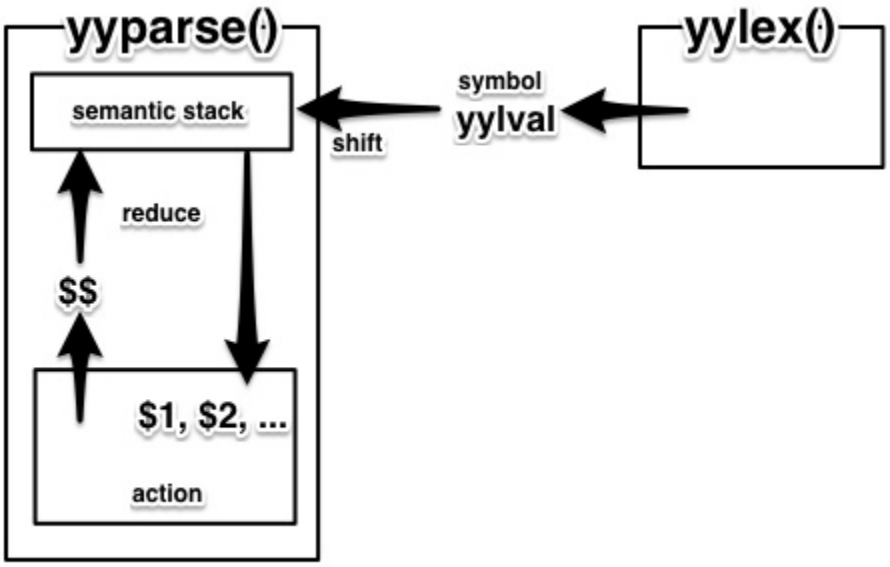


Figura 2. Interacción del analizador léxico y sintáctico.

Como base principal esta etapa debe contener una gramática independiente del contexto, donde las reglas se representan por medio de producciones y cada producción define un símbolo no terminal en función de símbolos terminales o tokens, y otros símbolos no terminales.

Para la construcción del analizador sintáctico se utilizó la herramienta *YACC (Yet Another Compiler-Compiler)* cuya función es tomar una gramática construida y generar el código fuente del parser en lenguaje Java.

La clase *Parser* crea una instancia única de Léxico con el archivo recibido como argumento del programa y le solicita tokens llamando al método yylex.
Por cada llamado, el Lexico lee el siguiente token del código fuente del archivo, almacena

la información adecuada en la tabla de símbolos y entrega como resultado del método yylex el número entero del token. También entrega, cuando es necesario, en un objeto ParserVal la clave String para acceder al token en la tabla de símbolos. Dicha clave es el lexema del token.

PROCEDIMIENTO DE DESARROLLO

Construcción de la Gramática.

En el trabajo aquí presentado se construyó una gramática independiente del contexto, en donde diferenciamos los siguiente *no terminales*:

- **Programa:** Describe la estructura de un programa.
- **Conjunto de Sentencias:** Es el conjunto de sentencias ya sean ejecutables o declarativas que pueden estar dentro del programa.
- **Bloque de sentencias:** Son el conjunto de sentencias que se reconocen entre un *Begin* y *End*. Se utiliza principalmente como cuerpo de una sentencia condicional o un bucle.
- **Declarativa:** Conjunto de reglas que infieren a una declaración.
- **Declarvar:** Regla utilizada para la declaración de una o más variables.
- **Declar_compuesto:** Regla utilizada para declarar una o más variables de un tipo definido por el usuario.
- **DeclarFun:** Define la declaración de una función.
- **CuerpoFun:** Específica el cuerpo de una función.
- **Retorno:** Especifica cómo debe estar estructurado el retorno en una función.
- **Invocacion_fun:** Regla utilizada para inferir la invocación de una función.
- **ListaVar:** Indica una o varias variables separadas por una coma.
- **Def_triple:** Especifica la definición de una variable definida por el usuario.
- **Tipo_compuesto:** Especifica los tipos que puede definir el usuario.
- **Ejecutable:** Regla utilizada para las diferentes sentencias ejecutables.
- **Asig:** Especifica una asignación.
- **SentenciaIf:** Define la sentencia IF.
- **Repeat_until:** Define la sentencia REPEAT UNTIL.

- **Condicion:** Define la estructura de una condición, que puede ser simple o una comparación entre lista de expresiones.
- **Comparador:** Especifica los diferentes tipos de comparadores.
- **Goto:** Define la sentencia GOTO.
- **Salida:** Define la sentencia OUTF.
- **Exp_arit:** Regla utilizada para denotar las diferentes expresiones aritméticas permitidas (suma y resta).
- **Lista de Exp_arit:** Regla utilizada para definir una lista de expresiones aritméticas separadas por una coma.
- **Termino:** Regla utilizada para inferir las diferentes operaciones que se pueden utilizar en el programa.

Por otra parte se declararon los *tokens* que serán utilizados por el usuario:

- MENORIGUAL
- ID
- ASIGNACION
- DISTINTO
- MAYORIGUAL
- SINGLE_CONSTANTE
- ENTERO_UNSIGNED
- OCTAL
- MULTILINEA
- REPEAT
- IF
- THEN
- ELSE
- BEGIN
- END
- END_IF
- OUTF
- TYPEDEF
- FUN
- RET

- GOTO
- TRIPLE
- OCTAL
- TIPO_UNSIGNED
- TIPO_SINGLE
- TIPO_OCTAL
- UNTIL

Por último se definieron dos tipos de precedencia, los símbolos ‘ - ’, ‘ + ’, ‘ / ’ y ‘ * ’ tiene precedencia a la izquierda, lo que significa que en una expresión compuesta se agrupa de izquierda a derecha.

Problemas Surgidos (solución de conflictos shiftreduce y reduce-reduce)

Si bien no surgieron gran cantidad conflictos shift reduce al momento de escribir la gramática del lenguaje, debido a la utilización de recursividad a izquierda que previene ambigüedades, no encontramos la forma de permitir identificar constantes negativas de tipo single. La estructuración inicial se realizó de la siguiente manera:

```
constante: SINGLE_CONSTANTE {
             lexico.getTablaSimbolos().editarLexema($1.sval,
truncarFueraRango($1.sval, lexico.getContadorLinea()));
           }
          |ENTERO_UNSIGNED
          |OCTAL
          |'-' SINGLE_CONSTANTE {
             lexico.getTablaSimbolos().editarLexema($2.sval,
trucarFueraRango($2.sval+$1.sval));
           }
       ;
```

Sin embargo, la regla que define constante como '-' SINGLE_CONSTANTE producía conflictos shift-reduce. Para solucionar este problema se planteó quitar el uso de constante y tratar esas reglas dentro de factor.

Aun así se detectó un segundo problema, que surge en el manejo de errores de una

condición. Dentro de la regla *condición* permitimos el hecho de tener dos expresiones aritméticas sin un comparador permitiendo que se siga compilando con normalidad en tanto surja este error. Esto trae como consecuencia un conflicto *shift-reduce* ya que al no tener un comparador podría darse el caso de no poder diferenciar si se trata de una SINGLE CONSTANTE positiva o negativa. Por este motivo, se optó por **no** manejar dicho error de condición.

Por otro lado, surgieron varios conflictos de ambigüedad de la gramática al comenzar con el manejo de errores. Dichos conflictos fueron solucionados en varios casos utilizando el token “error” proveído por YACC para poder seguir compilado luego de un error de sintaxis. En otros casos, bastó con reescribir las reglas de forma que no interfieran con las reglas de la gramática original del lenguaje.

A continuación se detallan los errores tenidos en cuenta en la gramática y cómo se implementaron para que el sintáctico pueda seguir compilando a pesar de ellos.

Manejo de Errores

Se escribió una gramática de error, combinada con el token error proveido por YACC, para detectar los siguientes errores de sintaxis:

Falta de nombre de programa y falta de delimitador de programa.

```
programa : ID BEGIN conjunto_sentencias END ';' {
                lexico.getTablaSimbolos().agregarUso($1.sval,
                 NOMBRE_PROGRAMA);
                 System.out.println("Se detecto: Programa");
           }
         | BEGIN conjunto_sentencias END ';' {
                 System.out.println("Error, Falta nombre de programa");
           }
         | ID conjunto_sentencias END ';' {
                System.out.println("Error de delimitador de programa ");
           }
        ;
```

Los manejos de errores de delimitación sólo se restringieron a detectar la falte de *BEGIN* al inicio del programa, debido a que no se puede identificar que el conjunto de sentencias terminó cuando se llega el fin de archivo.

Falta de ";" al final de las sentencias.

```
conjunto_sentencias : declarativa ';'
                    | declarativa {
                        System.out.println("Falta ; " + "en linea: "
+ lexico.getContadorLinea());
                      }
                    | ejecutable ';'
                    | ejecutable {
                        System.out.println("Falta ; " + "en linea: "
+ lexico.getContadorLinea());
                      }
                    | conjunto_sentencias declarativa ';'
                    | conjunto_sentencias declarativa{
                        System.out.println("Falta ; " + "en linea: "
+ lexico.getContadorLinea());
                      }
                    | conjunto_sentencias ejecutable ';'
                    | conjunto_sentencias ejecutable{
                        System.out.println("Falta ; " + "en linea: "
+ lexico.getContadorLinea());
                      }
;
```

Falta de nombre en función. Falta de nombre de parámetro formal en declaración de función. Falta de sentencia RET en función.

```
declaracionFun : tipo FUN ID '(' parametro ')' BEGIN conjunto_sentencias retorno
END
               |  tipo FUN ID '(' parametro ')' BEGIN retorno END
               | tipo FUN ID '(' parametro ')' BEGIN conjunto_sentencias END {
                    System.out.println("Error, falta retorno en funcion");
                }
               | tipo FUN  '(' parametro ')' BEGIN conjunto_sentencias retorno END
                {
                    System.out.println("Error, Falta nombre de funcion");
                }
               | tipo FUN  '(' parametro ')' BEGIN retorno END {
                    System.out.println("Error, Falta nombre de funcion");
                }
               |tipo FUN ID '(' ')' BEGIN conjunto_sentencias retorno END {
                    System.out.println("Error, Falta parametro de funcion");
                }
               |tipo FUN ID '(' ')' BEGIN retorno END {
                    System.out.println("Error, Falta parametro de funcion");}
```

Cantidad errónea de parámetros en invocación de función.

```
invocacion_fun : ID '(' exp_arit ')'
                |ID '(' tipo '(' exp_arit ')' ')'
                |ID '(' ')' {

                  System.out.println("Error de falta de parámetro en
invocación a función en linea: " + lexico.getContadorLinea());

                 }
               ;
```

Falta parámetro en sentencia OUTF. Parámetro incorrecto en sentencia OUTF.

```
 salida : OUTF '(' MULTILINEA ')'
        | OUTF '(' exp_arit ') '
        |OUTF '(' ')'  {
            System.out.println("Error, falta parámetro " + "en linea: " +
lexico.getContadorLinea());
         }
        |OUTF '(' error ')' {
            System.out.println("Error, parametro invalido " + "en linea: " +
lexico.getContadorLinea());
         }
        ;
```

Falta de paréntesis en condición de selecciones e iteraciones. Falta de comparador en comparación.

```
condicion : '(' exp_arit comparador exp_arit ')' {
             System.out.println("Se detecto: comparación");
            }
          | '(' '(' lista_exp_arit ')' comparador '(' lista_exp_arit
')' ')' {
             System.out.println("Se detecto: comparación múltiple");
            }
          | '(' '(' lista_exp_arit  comparador '(' lista_exp_arit ')'
')' {
             System.out.println("Error, falta de parentesis en la
condicion " + "en linea: " + lexico.getContadorLinea());
            }
          | '(' '(' lista_exp_arit ')' comparador lista_exp_arit ')'
')' {
```

```
                    System.out.println("Error, falta de parentesis en la
condicion " + "en linea: " + lexico.getContadorLinea());
                  }
                | '(' lista_exp_arit ')' comparador '(' lista_exp_arit ')'
')' {
                    System.out.println("Error, falta de parentesis en la
condicion " + "en linea: " + lexico.getContadorLinea());
                  }
                | '(' '(' lista_exp_arit ')' comparador '(' lista_exp_arit
')' {
                    System.out.println("Error, falta de parentesis en la
condicion " + "en linea: " + lexico.getContadorLinea());
                  }
                | exp_arit comparador exp_arit ')' {
                    System.out.println("Error, falta de parentesis en la
condicion " + "en linea: " + lexico.getContadorLinea());
                  }
                | '(' exp_arit comparador exp_arit  {
                    System.out.println("Error, falta de parentesis en la
condicion " + "en linea: " + lexico.getContadorLinea());
                  }
                | '(' '(' lista_exp_arit ')' '(' lista_exp_arit ')' ')' {
                    System.out.println("Error, falta de comparador " + "en
linea: " + lexico.getContadorLinea());
                  }
                | '(' '(' lista_exp_arit ')'  ')' {
                    System.out.println("Error, falta de lista de expresión
aritmetica en comparación " + "en linea: " + lexico.getContadorLinea());
                  }
                ;
```

Falta de cuerpo en iteraciones. Falta de UNTIL en ciclo REPEAT UNTIL.

```
repeat_until : REPEAT bloque_sentencias_ejecutables UNTIL  condicion
             |REPEAT UNTIL condicion {
                System.out.println("Error, falta cuerpo en la iteracion " + "en
linea: " + lexico.getContadorLinea());
               }
             | REPEAT bloque_sentencias_ejecutables condicion {
                System.out.println("Error, falta de until en la iteracion repeat"
+ "en linea: " + lexico.getContadorLinea());
                }
```

```
            ;
```

Falta de END_IF. Falta de contenido en bloque THEN/ELSE.

```
sentencia_if : IF  condicion  THEN bloque_sentencias_ejecutables END_IF
             | IF  condicion  THEN bloque_sentencias_ejecutables {
                 System.out.println("Error, Falta END_IF de cierre " + "en
linea: " + lexico.getContadorLinea());
               }
             | IF condicion THEN bloque_sentencias_ejecutables ELSE
bloque_sentencias_ejecutables END_IF {
                 System.out.println("Se detecto: Sentencia if " + "en
linea: " + lexico.getContadorLinea());

               }
             | IF condicion THEN bloque_sentencias_ejecutables ELSE
bloque_sentencias_ejecutables {
                 System.out.println("Error, Falta END_IF de cierre " + "en
linea: " + lexico.getContadorLinea());
               }
             | IF  condicion  THEN END_IF {
                 System.out.println("Error, Falta de contenido en el bloque
then " + "en linea: " + lexico.getContadorLinea());
               }
             |IF condicion THEN bloque_sentencias_ejecutables ELSE END_IF {
                 System.out.println("Error, Falta de contenido en el bloque
else " + "en linea: " + lexico.getContadorLinea());
               }
             ;
```

Falta de operando en expresión.

Se pudo tener en cuenta el error de que falte el segundo operando después un signo + o un signo - . Sin embargo, esto implica que se genere el error de que falta ‘;’ porque fue tomado como parte del error.

```
exp_arit : exp_arit '+' error ';' {
              System.out.println("Error: Falta el término después de
'+' en expresion aritmetica en línea: " + lexico.getContadorLinea());}
           | exp_arit '-' error ';'  {
              System.out.println("Error: Falta el término después de
'-' en expresión aritmetica en línea: " + lexico.getContadorLinea());}
```

Falta de operador en expresión.

No se pudo tener en cuenta el error de que falte el operador + o - debido a que se presentaban conflictos shift reduce.

Falta de "," en lista de variables/expresiones.

Este error no se tiene en cuenta, debido a que el error que se detectara primero es el de ";" en la declaración de una sola variable.

Ejemplo:

```
progrma BEGIN
          TIPO_SINGLE w q;
        END;
```

En la segunda línea se detectará el error de que falta ';' luego de la declaración de la variable w. Luego se detectará error de sintaxis para q;.

Decisiones de Implementación.

Dentro del sintáctico se tomaron decisiones en pos del funcionamiento general de la sintaxis.

Una decisión de implementación se hizo en un bloque de sentencias ejecutable, el cual comienza con la palabra reservada BEGIN, luego sigue con un conjunto de sentencias, para finalizar con la palabra reservada END. A posterior de esta palabra reservada **no** se coloca punto y coma.

Por otro lado, se implementó la regla *tipo_compuesto* con el objetivo de si en el futuro se quiere implementar otro tipo que pueda generar el usuario, sólo deberá agregarse en esta regla.

Generador de código intermedio.

Luego de la generación de reglas por parte del analizador sintáctico comienza una etapa crucial en la compilación de un programa, *la generación de código intermedio*. El código intermedio es una representación previa a la traducción del programa a un código máquina.

En esta etapa se generará código de las sentencias que usan variables, invocan funciones y utilizan los tokes, utilizando sus atributos y asociando información a los símbolos de la gramática.

Por otro lado se plantea un chequeo semántico en el que se hará:

- Comprobación de tipo.
- Comprobación de flujo de control.
- Comprobación de unicidad.

Tercetos.

Los *tercetos* son una forma de representación intermedia del código que consta de tres partes. La primer parte almacena el operando u operación, mientras que la segunda y tercera pueden almacenar una referencia a otro terceto o un lexema. Visualmente el terceto tiene la siguiente forma:

$$T1.\ (\ +,\ 1,\ 8\ )$$

$$T2.\ (\ :=,\ a,\ [T1]\ )$$

Para la representación en código de esta estructura se planteó la creación de una clase terceto que tiene como atributos tres variables de tipo String que representan los operandos, y una variable de tipo Integer que representa el tipo.

Generador.

A medida que se recorre el programa y se genera la lista de reglas, se deben ir creando los tercetos correspondientes y tienen que almacenarse de forma tal que luego puedan ser accedidos.

Para lograr este cometido se implementó la clase *Generador* la cual contiene los siguientes atributos:

```
private ArrayList<Terceto> tercetos;
private Stack<String> pila;
private ArrayList<String> etiquetas;
private HashMap<String> mapGoto;
private String[][] tablaComp;
```

Dicha clase debe tener una instancia única en todo el programa, por este motivo se utilizó el patrón de diseño *singleton*.

Creación de Tercetos.

Los tercetos deben ser creados en las siguientes reglas:

- Invocación a función.
- Declaración de función.
- Asignación.
- Goto.
- Sentencia if.
- Sentencia REPEAT_UNTIL.
- Etiqueta.
- Salida.

Tipos.

Al momento en que el analizador léxico detecta un identificador, o variable, y su lexema es almacenado en la tabla de símbolos, no siempre será posible saber el tipo de dicho identificador.

Por este motivo dentro de la gramática en cada regla que represente una declaración de variable se deberá cargar en la tabla de símbolos como atributo el tipo correspondiente según la declaración.

Dentro del trabajo los tipos son representados numéricamente de la siguiente manera:

- **Palabra Reserv:** 0.
- **Unsigned :** 1.
- **Single :** 2.
- **Octal :** 3.
- **Multilinea :** 4.
- **Triple Unsigned:** 5.
- **Triple Single:** 6.
- **Triple Octal:** 7.
- **Etiqueta:** 8.
- **Salto:** 9.
- **Función:** 10.
- **Desconocido :** 50.
- **Símbolo:** 60.

Por otra parte, es necesario que cada terceto tenga como característica de qué tipo es la operación. La distinción de tipos se hace de la misma manera.

Conversiones.

El compilador **no** permitirá conversiones en comparaciones, operaciones o asignaciones, por lo tanto se verifica en cada una de estas reglas si el tipo de las variables son iguales o si el tipo de la variable es igual al del terceto con el que se quiere operar. En caso de no ser así se arroja un error.

A pesar de que el compilador no permite las conversiones antes mencionadas si admite una conversión explícita en el parámetro real en un llamado a una función. Esto es posible ya que al momento de declarar una función el identificador de la misma está guardado en la tabla de símbolos, donde se almacenará el tipo del parámetro. Esto permite que al momento de invocar una función verificar en la tabla de símbolos el tipo del parámetro y verificar si es idéntico al del parámetro real, o en caso de hacer una conversión explícita si es posible hacerla entre esos tipos.

En pos de lograr una conversión explícita y que se vea reflejado en el código intermedio, dentro de la clase *Generador* se implementó una matriz que contiene la instrucción que debe guardarse en el terceto, la cual permitirá generar la conversión posteriormente. En caso de no poderse dar dicha conversión almacena un valor nulo. La matriz se puede ver reflejada en la siguiente tabla:

	Unsigned	**Single**	**Octal**
Unsigned	*itoI*	*stoI*	*null*
Single	*itoS*	*stoS*	*null*
Octal	*null*	*null*	*oto0*

////**AGREGAR LO DE TIPOS DEFINIDOS POR USUARIOS…….**

Uso.

Este atributo, tal y como lo dice su nombre, nos indica qué uso se le dará a cada identificador detectado. Esta información es de gran utilidad al momento de generar código máquina.

Los tipos de uso están diferenciados por valores numéricos y son diferenciados de la siguiente manera:

- **Nombre de variable:** 101.
- **Nombre de función:** 102.
- **Nombre de tipo:** 103.

- **Nombre de parámetro:** 104.
- **Nombre de etiqueta:** 105.
- **Nombre de programa:** 106.

Ámbitos.

El ámbito podemos definirlo como el contexto dentro del cual una variable, función, o cualquier identificador es reconocible y accesible. Determina dónde una entidad definida en el código puede ser utilizada y dónde no es visible o accesible.

El lenguaje de programación utilizado en este trabajo tiene un binding de alcance estático, por ende el compilador debe poder reconocer y diferenciar los ámbitos a donde pertenecen las variables.

Para lograr este objetivo se implementó el método *name manglin* que consta en tener una variable de tipo String que almacena y actualiza el ámbito en función del programa que se esté ejecutando. Esta variable de ámbito comienza con el ámbito ".global" y al momento se detecta una función actualiza el ámbito concatenando lo que ya tenía con el nombre de la función encontrada. Por cada declaración de variable que encuentre cambiará el lexema del identificador, concatenandole el ámbito.

En la implementación del código dentro de la gramática, al momento de hacer algún tipo de operación que requiera ir a buscar un identificador a la tabla de símbolos a esté se le concatenará el valor de ámbito actual y se procederá a buscarlo. En caso de no encontrarlo con dicho ámbito se desconcatenará el último ámbito y se volverá a buscar. El proceso continúa hasta que el ámbito de la variable sea el global.

Sentencias Condicionales.

En el manejo de sentencias condicionales hay que tener especial cuidado, ya que realmente lo que se debe lograr es hacer un salto por falso a una etiqueta específica en caso que la condición sea falsa, y un salto incondicional a otra etiqueta en caso de verdadero.

El problema surge que al momento de crear el terceto que genera el salto no es posible determinar a qué etiqueta hay que saltar ya que aún no ha sido creada.

Para lograr este cometido se requirió implementar una pila de tipo String en la clase *generador* que almacenará posiciones de tercetos que aún no tengan definido los saltos, para posteriormente poder completarlos.

Sentencia IF.

La sentencia IF está constituida de la siguiente manera:

```
Sentencia_if : condicion_if  bloque_sentencias_ejecutables condicion_ELSE
                    bloque_sentencias_ejecutables END_IF
```

Donde

condicion_if : **IF** condicion **THEN**

condicion_else : ELSE

Se optó por separar la regla de esta manera ya que es necesario distinguir el momento en que termino una condición para poder crear un terceto de salto por falso y almacenarlo en la pila, para después al momento de que se lea un *ELSE* poder crear la etiqueta, desapilar el terceto, completarlo, crear otro terceto de salto incondicional y almacenarlo en la pila. Sobre el final de la *sentencia_if* se podrá crear una etiqueta de salto y desapilar, y completar el terceto faltante.

Sentencia REPEAT UNTIL.

La sentencia Repeat Until esta formada por:

```
repeat_until : sentencia_repeat bloque_sentencias_ejecutables UNTIL
                                condicion
```

donde

sentencia_repeat: REPEAT

El propósito es similar al de la *sentencia_if* antes mencionada. Se debe reconocer primero la palabra reservada REPEAT para poder crear una etiqueta y luego almacenar dicha etiqueta en el terceto de salto incondicional que se creará después de la condición.

Saltos Incondicionales.

El lenguaje presentado cuenta con la función **GOTO** que permite saltar a una etiqueta dentro del programa, siempre y cuando se encuentre en el mismo ámbito.

El punto clave se encuentra en que la etiqueta puede estar declarada antes o después de el llamado a **GOTO**, por ende se debe tener un especial cuidado con dicho manejo.

Para la implementación de esta funcionalidad se implementó dentro de la clase *generador* una estructura de lista en donde se almacenan las etiquetas creadas(concatenadas con su ámbito). Por otra parte se utilizó una estructura de tipo hash con una clave Integer y un valor de tipo String, en la cual se almacenarán los tercetos que contengan un salto a una etiqueta aún inexistente. En está última estructura la clave representa el número de terceto y el valor de la etiqueta aún no creada.

Al momento en que se detecta una etiqueta se verifica si existe. Si la misma no se encuentra en la lista, se agrega a la misma y se recorre el hash map buscando algún terceto que la haya referenciado. En el caso de encontrar un terceto así se procede a eliminarlo de la estructura.

Por otro lado, siempre que se realice un salto se almacenará el terceto de salto con la etiqueta detectada, y en el caso que no se encuentre la misma se procede a almacenar el terceto en la estructura de hash.

Funciones.

asdasdas …..

Problemas surgidos.

asdasda

Generador de código Assembler.

Salida del Compilador.

A medida que se lee un archivo se irán detectando tokens y esta detección se imprimirá por consola, donde se podrá observar el lexema identificado y el valor del token. Si el token identificado es un carácter especial (*ejemplo:* > ó ;) solo se mostrará el valor del carácter en ASCII.

Por otra parte, al momento que el compilador identifica una regla, la misma se indicará

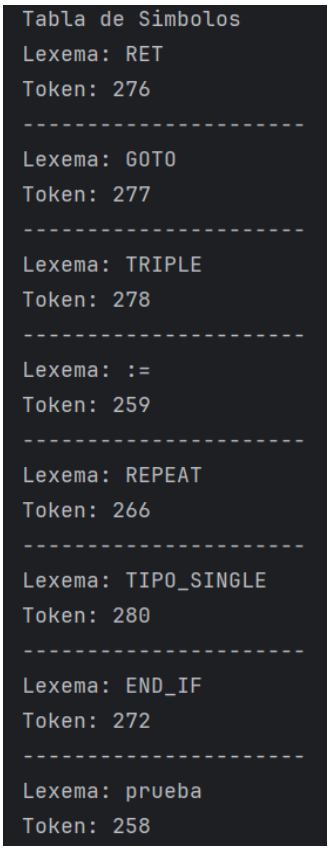
por pantalla junto con su número de línea. Lo mismo sucede si hay un error o se devuelve una advertencia.

Programa:

```
1   prueba BEGIN
2   TIPO_OCTAL x;
3   x:= 1;
4   END ;
```

Salida:

```
Token: prueba - Valor: 258

Token: BEGIN - Valor: 270

Token: TIPO_OCTAL - Valor: 281

Token: x - Valor: 258

Token:  - Valor: 59
Se detecto: Declaración de variable en linea: 2

Token: x - Valor: 258

Token: := - Valor: 259

Token: 1 - Valor: 263

Token:  - Valor: 59
Se detecto: Asignacion en linea: 3

Token: END - Valor: 271

Token:  - Valor: 59
Se detecto: Programa

Token:  -Fin de Archivo (0)
```



Como se puede observar, se muestra cada lexema junto con su número de token. En la tabla de símbolos se encuentran todas las palabras reservadas del lenguaje junto con los tokens añadidos durante la lectura del código fuente.

Ejecución del Compilador.

Se adjunta el archivo Compilador.jar compilado desde la versión de Java 11.

Para ejecutar dicho archivo se le debe pasar como argumento del programa un archivo de texto que contenga el código fuente a compilar. Esto se debe realizar mediante consola con el siguiente comando: **java -jar** *Compilador.jar* <*archivo*>.

Algunos de los códigos de prueba se adjuntan en la carpeta Pruebas, separando por sentencias de la gramática para verificar su detección junto con sus errores sintácticos.

Dichos archivos de prueba son:

- Prueba_general_1
- Prueba_general_2
- Prueba_general_3
- Prueba_variables
- Prueba_if
- Prueba_repeat_until
- Prueba_programa_sin_nombre
- Prueba_goto
- Prueba_funcion
- Prueba_falta_punto_coma
- Prueba_falta_begin
- Prueba_expresiones
- Prueba_declaracion
- Prueba_comparaciones
- Prueba_caracter_invalido
- Prueba_cadena
- Prueba_asignacion

Por último, junto al ejecutable se encontrará un archivo **.zip** con el código fuente.

CONCLUSIONES

En esta primera etapa del compilador, nos centramos en el análisis léxico y sintáctico. Para el análisis léxico, desarrollamos un autómata que se encargó de reconocer los tokens del lenguaje, como palabras clave, operadores y constantes, y también manejamos errores básicos, como tokens mal formados. Además, implementamos una tabla de símbolos utilizando la estructura HashMap para almacenar información relevante sobre los identificadores y otros elementos que aparecen en el código fuente que luego será utilizada por las siguientes etapas del compilador.

Con YACC, logramos automatizar el análisis sintáctico, definiendo las reglas de la gramática del lenguaje y asegurándonos de que el programa fuente siga una estructura válida. En este proceso, también trabajamos en la integración de los dos análisis, de

modo que las decisiones del analizador léxico sirvan de base para construir las estructuras sintácticas más grandes.

En resumen, el proyecto funciona de forma óptima en estas primeras dos etapas, detectando tokens válidos, corrigiendo tokens inválidos para continuar la compilación, componiendo el árbol sintáctico en base a la gramática definida que describe al lenguaje y permitiendo que el sintáctico continúe su ejecución al encontrarse con errores sintácticos.
En las próximas etapas se realizará la generación de código assembler para el lenguaje, utilizando estructuras ya implementadas como la tabla de símbolos y métodos del léxico.