Diseño de Compiladores I

Trabajo Práctico Especial Partes 1 y 2.

Integrantes:

- Marín, Cynthia Romina (cynthia.romina.marin@gmail.com)
- Sarti, Mauro (msarti@alumnos.exa.unicen.edu.ar)

Ayudante Asignado: Tommasel, Antonela

Número de Grupo: 10

Temas Incluidos: 3,6,10,13,16,17,20.

Parte 1 – Analizador Léxico

Introducción y Objetivos

El objetivo de este trabajo práctico es desarrollar un Analizador Léxico que reconozca una cierta cantidad de tokens generales como identificadores, constantes, operadores aritméticos, comparadores, palabras reservadas, además de los temas particulares de nuestro grupo, que son los siguientes:
Tema 3: Incorporar enteros largos, constantes enteras con valores entre –231 y 231 – 1. Se debe incorporar a la lista de palabras reservadas la palabra LONG.
Tema 6: Incorporar los números dobles. 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 DOUBLE.
Tema 10: Incorporar a la lista de palabras reservadas las palabras SWITCH y CASE.
Tema 13: Incorporar a la lista de palabras reservadas las palabras FUNCTION, RETURN y MOVE.
Tema 16: Sin conversiones. Se definirá en el Trabajo Práctico 3.
Tema 17: Comentarios de 1 línea: Comentarios que comiencen con "**" y terminen con el fin de línea.
Tema 20: Incorporar las cadenas multilínea. Estas son cadenas de caracteres que comienzan con y terminan con " **""** " . Estas cadenas pueden ocupar más de una línea.

La función principal del Analizador Léxico es obtener una serie de Tokens, informar de errores léxicos, eliminar espacios y comentarios a medida que reconoce elementos del programa fuente.

El reconocimiento de los Tokens se hace mediante un autómata finito. Este autómata tiene un conjunto finito de estados, un conjunto finito de símbolos de entrada, una función de transición, un estado inicial y un conjunto de estados finales (en nuestro caso es un solo estado).

Los Tokens reconocidos por el Analizador Léxico y la información pertinente a cada uno se almacenan en una Tabla de Símbolos.

Por otro lado, tenemos Acciones Semánticas que están asociadas a los estados entre el autómata. Estas acciones sirven para ir actuando a medida que se va reconociendo un Token u otro, lo que nos permite generalizar comportamientos similares.

Decisiones de Diseño e Implementación

Para este trabajo se decidió hacer una clase Léxico y una clase Token. Esta clase Léxico tiene el archivo que se quiere cargar para poder analizar (archivoACargar) y asociado a esto tiene una lista de líneas (locs) que corresponden a las líneas del archivo previamente ingresado. También contiene una lista de errores (errores) para tener referencia a que tokens pertenecen. Cuenta

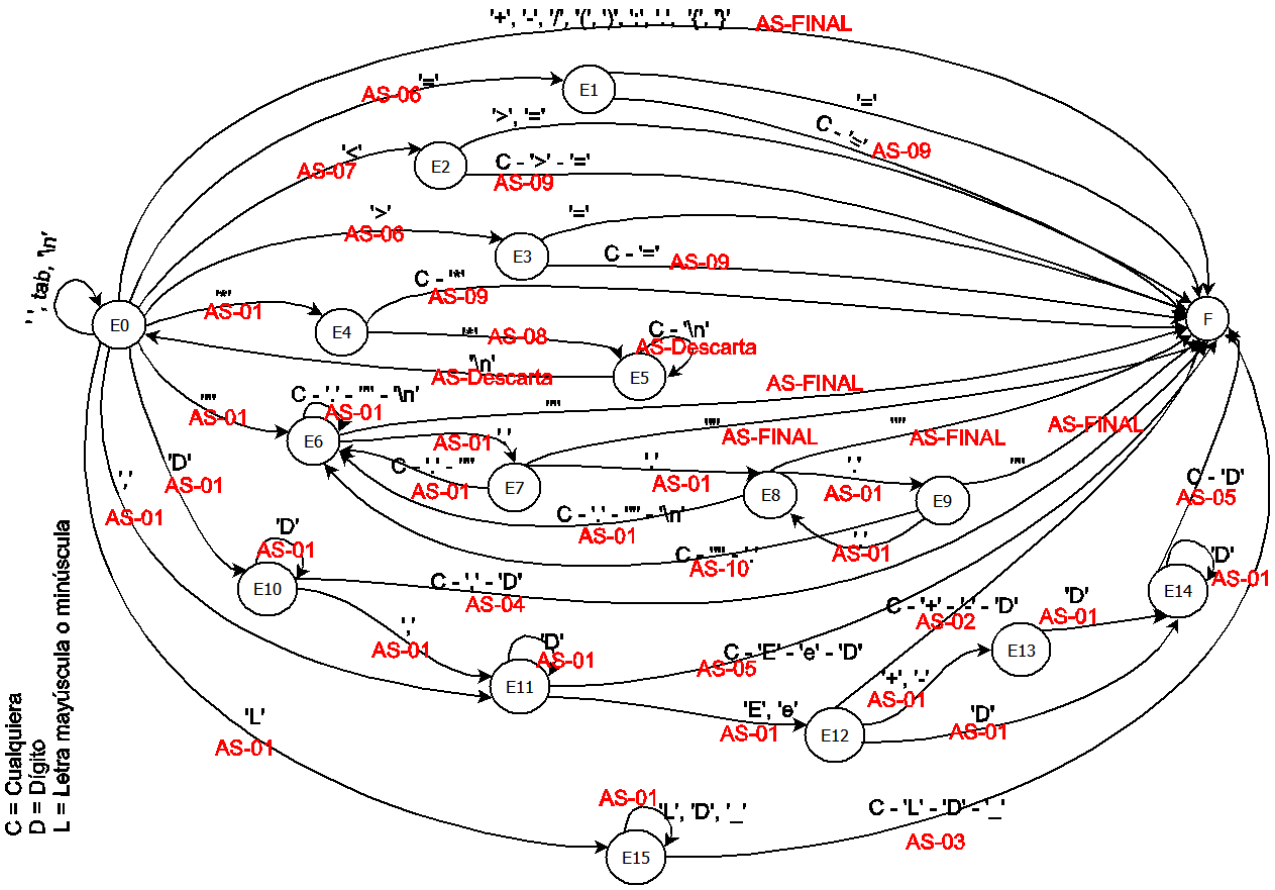
además con una tabla de palabras reservadas (Reservadas) en donde se almacena la clave y la palabra y hay otra tabla, la tabla de símbolos (tablaSimbolos). En este caso, la tabla de símbolos tiene una clave (un número) y una lista de Tokens, esto es porque para el caso de los identificadores por ejemplo, puedo tener varios identificadores y todos tienen el mismo número de clave.

También en esta clase podemos encontrar la matriz de transición de estados y las variables que sirven para el manejo de la misma (fila, estado). Y otra variable adicional (pos) que nos sirve para recorrer el archivo que estamos leyendo.

Por último, en la clase Léxico encontramos un objeto de la clase “Principal” que es la interfaz gráfica.

Otra cuestión importante a resaltar es la creación de una clase Token. Esta clase contiene información sobre el Token que es útil y que consideramos nos puede servir más adelante. Esta clase contiene una cadena de caracteres (lexema) que es el Token identificado, la clave asociada al mismo, el tipo de Token (identificador, literal, etc.) y el número de línea donde aparece.

Diagrama de Transición de Estados



Matriz de Transición de Estados

En esta matriz podemos observar que la matriz de transición de estados y la matriz de acciones semánticas están juntas por una cuestión de comodidad. Las filas indican el “Estado Actual” y las columnas el “Carácter Leído” y la celda respectiva es el “Estado al Cual Voy” y la “Acción Semántica” a ejecutar. Esta matriz tiene un elemento “Par” en la implementación para poder generar las dos matrices en una, disminuyendo así tiempos de acceso.

	L	D	+	-	*	/	=	<	>	(
E00	E15	E10	FINAL	FINAL	E04	FINAL	E01	E02	E03	FINAL
	AS-01	AS-01	AS-FINAL	AS-FINAL	AS-01	AS-FINAL	AS-06	AS-07	AS-06	AS-FINAL
E01	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-FINAL	AS-09	AS-09	AS-09
E02	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-FINAL	AS-09	AS-FINAL	AS-09
E03	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-FINAL	AS-09	AS-09	AS-09
E04	FINAL	FINAL	FINAL	FINAL	E05	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-09	AS-09	AS-09	AS-08	AS-09	AS-09	AS-09	AS-09	AS-09
E05	E05	E05	E05	E05	E05	E05	E05	E05	E05	E05
	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta
E06	E06	E06	E06	E06	E06	E06	E06	E06	E06	E06
	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01
E07	E06	E06	E06	E06	E06	E06	E06	E06	E06	E06
	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01
E08	E06	E06	E06	E06	E06	E06	E06	E06	E06	E06
	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01
E09	E06	E06	E06	E06	E06	E06	E06	E06	E06	E06
	AS-10	AS-10	AS-10	AS-10	AS-10	AS-10	AS-10	AS-10	AS-10	AS-10
E10	FINAL	E10	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-04	AS-01	AS-04	AS-04	AS-04	AS-04	AS-04	AS-04	AS-04	AS-04

	(	)	,	:	.	{	}	E, e	"	\n	esp, tab	_
E00	FINAL	FINAL	E11	FINAL	FINAL	FINAL	FINAL	ERROR	E06	E00	E00	ERROR
	AS-FINAL	AS-FINAL	AS-01	AS-FINAL	AS-FINAL	AS-FINAL	AS-FINAL	AS-ERROR	AS-01			AS-ERROR
E01	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09
E02	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09
E03	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09
E04	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09
E05	E05	E05	E05	E05	E05	E05	E05	E05	E05	E00	E05	E05
	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta	AS-Descarta
E06	E06	E06	E06	E06	E07	E06	E06	E06	FINAL	ERROR	E06	E06
	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-FINAL	AS-ERROR	AS-01	AS-01
E07	E06	E06	E06	E06	E08	E06	E06	E06	FINAL	ERROR	E06	E06
	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-FINAL	AS-ERROR	AS-01	AS-01
E08	E06	E06	E06	E06	E09	E06	E06	E06	FINAL	E06	E06	E06
	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-01	AS-FINAL	AS-01	AS-01	AS-01
E09	E06	E06	E06	E06	E08	E06	E06	E06	FINAL	E06	E06	E06
	AS-10	AS-10	AS-10	AS-10	AS-01	AS-10	AS-10	AS-10	AS-FINAL	AS-10	AS-10	AS-10
E10	FINAL	FINAL	E11	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-04	AS-04	AS-01	AS-04	AS-04	AS-04	AS-04	AS-04	AS-04	AS-04	AS-04	AS-04

	L	D	+	-	*	/	=	<	>	(
E11	FINAL	E11	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-05	AS-01	AS-05	AS-05	AS-05	AS-05	AS-05	AS-05	AS-05	AS-05
E12	FINAL	E14	E13	E13	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-02	AS-01	AS-01	AS-01	AS-02	AS-02	AS-02	AS-02	AS-02	AS-02
E13	ERROR	E14	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
	AS-ERROR	AS-01	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR
E14	FINAL	E14	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-01	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09
E15	E15	E15	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-01	AS-01	AS-03	AS-03	AS-03	AS-03	AS-03	AS-03	AS-03	AS-03

	)	,	:	.	{	}	E, e	"	\n	esp, tab	_
E11	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	E12	FINAL	FINAL	FINAL	FINAL
	AS-05	AS-05	AS-05	AS-05	AS-05	AS-05	AS-01	AS-05	AS-05	AS-05	AS-05
E12	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-02	AS-02	AS-02	AS-02	AS-02	AS-02	AS-02	AS-02	AS-02	AS-02	AS-02
E13	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR	AS-ERROR
E14	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL
	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09	AS-09
E15	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	FINAL	E15
	AS-03	AS-03	AS-03	AS-03	AS-03	AS-03	AS-03	AS-03	AS-03	AS-03	AS-01

Implementación de la Matriz de Transición de Estados y Acciones Semánticas

La matriz de transición de estados y la matriz de acciones semánticas fueron implementadas en una misma matriz. Hay una clase dentro del proyecto llamada MatrizTransicionEstados que es la que contiene esta matriz doble.

La cantidad de filas es la cantidad de estados de y la cantidad de columnas es la cantidad de caracteres que puedo leer.

Cada celda de esta matriz es un “Par” para poder generar las dos matrices en una, disminuyendo así tiempos de acceso. Dentro de ese “Par” tenemos que el primer elemento es un número (el número del siguiente estado) y el segundo elemento es una acción semántica.

Ejemplo: `matrix [0][0] = new Pair<Integer, AccionSemantica>(15, new AS01());`

Acciones Semánticas

Las acciones semánticas se implementaron como clases. Existe una interfaz AccionSemantica y todas las acciones semánticas implementan la funcionalidad de esta. A continuación se listan todas las acciones semánticas con su función.

- **public class** AS01 **implements** AccionSemantica{}. La acción semántica 01 lo que hace es ir agregando el último carácter leido al token que venia leyendo.
- **public class** AS02 **implements** AccionSemantica {}. La acción semántica 02 hace varias cosas, primero devuelve dos elementos para ser leidos nuevamente, ya que se tuvo que hacer esto para poder reconocer un token, luego completa datos del literal (lo que leyó) y por último devuelve el Token (un Literal).
- **public class** AS03 **implements** AccionSemantica {}. La acción semántica 03 devuelve a la entrada del último carácter leído, obtiene el ID de token, si corresponde a una palabra reservada entonces completa datos y devuelve. Si no, completa datos para un identificador, revisa que no se pase del limite de caracteres,pero si se pasa corrige y genera el error. Por último, devuelve el token generado (Palabra reservada o Identificador).
- **public class** AS04 **implements** AccionSemantica {}. La acción semántica 04 nos devuelve a la entrada el último carácter leído, revisa que no se pase del rango, si se pasa, corrige y genera el error(que el Long es muy largo) y por último devuelve el Token armado (un LONG).
- **public class** AS05 **implements** AccionSemantica {}. La acción semántica 05 nos devuelve a la entrada el último carácter leído, si el lexema es una "," entonces genera el literal. Si no, revisa que no se pase del rango,pero si se pasa, corrige y genera el error. Por último, devuelve el Token armado (un DOUBLE).
- **public class** AS06 **implements** AccionSemantica {}. La acción semántica 06 tiene que verificar primero si el proximo caracter es un "=", si se cumple avanza la posicion para no leer dos veces lo mismo, genera el token Comparador y devuelve el Token armado (un Comparador).
- **public class** AS07 **implements** AccionSemantica {}. La acción semántica 07 debe mirar si el proximo caracter es un "=" o un ">", si esto ocurre, avanza la posicion para no leer dos veces lo mismo, genera el token Comparador y devuelve el Token armado (un Comparador).
- **public class** AS08 **implements** AccionSemantica {}. La acción semántica 08 borra el lexema leido, aumenta la linea donde esta leyendo y pone el estado en 0 (porque lo que acaba de detectar es un comentario).
- **public class** AS09 **implements** AccionSemantica {}. La acción semántica 09 devuelve a la entrada el último carácter leído, pregunta por el contenido del lexema (si es un operador aritmético, un operador de asignación o un comparador), arma el Token correspondiente y lo devuelve.
- **public class** AS10 **implements** AccionSemantica {}. La acción semántica 10 pregunta si lo que leyó es un salto de línea, en caso de serlo, se borran los últimos 3 caracteres, que serán los '.' Si no se agrega el carácter al string.
- **public class** ASDescarta **implements** AccionSemantica{}. La acción semántica Descarta pregunta si lo leído es un salto de línea, en ese caso, se aumenta la línea y se pone el estado en cero (Esto es porque lo que detectó es un comentario y los comentarios de descartan).

- `public class ASError implements AccionSemantica{}.` La acción semántica Error no ejecuta ninguna acción.
- `public class ASFinal implements AccionSemantica {}.` La acción semántica Final nos devuelve a la entrada el último carácter leído, pregunta por el contenido del lexema, arma el Token correspondiente y luego devuelve el Token.

Errores Léxicos Considerados

Se consideraron algunos errores en la etapa del analizador léxico, ellos son:

- En la AS03: El tamaño del identificador excede el maximo permitido. La acción correctiva que se tomo fue cortar el identificador a los 15 caracteres permitidos.
- En la AS04: El tamaño del LONG excede el máximo permitido. La acción correctiva que se tomo fue acotar el número al máximo permitido.
- En la AS05: El tamaño del DOUBLE excede el máximo permitido. La acción correctiva que se tomo fue acotar el número al máximo permitido.

Parte 2 – Analizador Sintáctico

Introducción y objetivos

En esta parte del trabajo, se construyó un parser para invocar al Analizador Léxico creado en la parte 1. Esto nos permite reconocer elementos de un lenguaje que fue previamente especificado en las consignas del Trabajo Práctico 2.

El Analizador Sintáctico requerido en este trabajo tiene la responsabilidad de agrupar tokens en frases gramaticales que el compilador usará en las siguientes etapas. Este analizador recibe una cadena de tokens generada por el Analizador Léxico y verifica que pueda ser generada mediante la gramática del lenguaje.

Las construcciones del lenguaje pueden ser descritas mediante la gramática que se describirá mas adelante. Todas sus reglas se representan por medio de producciones, y cada una de ellas define un símbolo NO TERMINAL en función de símbolos terminales, otros no terminales y tokens que se hayan definido con anterioridad.

Conceptualmente el Analizador Sintáctico construye un árbol de parsing, el cual describe la estructura sintáctica del código. Este árbol demuestra cómo la secuencia de tokens puede ser derivada a través de las reglas de la gramática.

En la gramática que se describe a continuación existen dos conflictos shift-reduce que se podrian solucionar si se elimina la recursion en el no terminal “sentencias” pero haciendo esto no nos permite tener varias declaraciones o asignaciones, etc juntas.

Gramática

programa : bloque_main ;
El no terminal programa define que solo puede haber un bloque principal.

bloque_main : bloque_comun| bloque_main bloque_comun ;
El no terminal bloque principal define que puede ser un bloque comun o varios bloques comunes. Es recursivo.

bloque_comun : bloques| declaracion_funcion ;
El no terminal bloque comun puede ser, un conjunto de bloques o una declaración de una función. Una función solo puede estar en este nivel y sin recursividad porque la gramática no admite anidamiento de funciones.

bloques : bloque_para_funcion
| BEGIN bloques END'.' {this.sintactico.showMessage("Bloque: BEGIN - END");} ;
El no terminal bloques esta definido por otro no terminal denominado bloque para función o un bloque que comienze con BEGIN y termine con END.

bloque_para_funcion : sentencia_if_else | sentencia_if | sentencia_switch| sentencias;
Este no terminal es solamente para una función ya que limita que dentro no pueda haber una declaración de funcion. Solamente se pueden definir sentencias IF – IF/ELSE – SWITCH/CASE – Asignaciones – Declaraciones – Salidas por pantalla – Llamados a funciones.

sentencia_if_else : IF '(' condicion ')' THEN bloques ELSE bloques END_IF'.'
{this.sintactico.showMessage("Sentencia: IF - ELSE");} ;
Este no terminal define una sentencia IF con parte ELSE.

sentencia_if : IF '(' condicion ')' THEN bloques END_IF'.'
{this.sintactico.showMessage("Sentencia: IF");} ;
Este no terminal define una sentencia IF sin parte ELSE.

sentencia_switch : SWITCH '(' IDENTIFICADOR ')' '{' rep_switch '}''.'
{this.sintactico.showMessage("Sentencia: SWITCH");} ;
Este no terminal define una sentencia SWITCH y dentro tiene otro no terminal denominado rep_switch que es el que se encarga de la recursión del CASE.

sentencias : sentencia_unica | sentencias sentencia_unica ;
El no terminal sentencias puede ser definido como una única sentencia o como un conjunto de sentencias, es recursivo a izquierda.

sentencia_unica : declaracion | asignacion| salida| llamado_funcion ;
Este no terminal es utilizado para definir una sola sentencia ya que si no tengo un bloque BEGIN/END no puedo tener varias sentencias juntas.

declaracion_funcion : tipo FUNCTION IDENTIFICADOR '{' bloques RETURN '(' expresion ')''.' '}'
{this.sintactico.showMessage("Declaracion de Funcion");}

| tipo MOVE FUNCTION IDENTIFICADOR '{' bloques RETURN '(' expresion ')''.' '}' {this.sintactico.showMessage("Declaracion de Funcion con MOVE");} ;
Este no terminal se utiliza para definir una función. No es recursivo porque la gramática no admite recursividad de funciones.

llamado_funcion : IDENTIFICADOR '('')''.' {this.sintactico.showMessage("Llamado a función");} ;
Este no terminal define como se realiza un llamado a una función.

asignacion : IDENTIFICADOR '=' expresion'.' {this.sintactico.showMessage("Asignación");} ;
Este no terminal define como se hace una asignación.

declaracion : IDENTIFICADOR',' declaracion {this.sintactico.showMessage("Declaracion de variable multiple");}
| IDENTIFICADOR ':' tipo'.' {this.sintactico.showMessage("Declaracion de variable");} ;
El no terminal declaración se utiliza para reconocer declaraciones de variables.

rep_switch : CASE CONSTANTE ':' bloques {this.sintactico.showMessage("Sentencia: CASE");}
| rep_switch CASE CONSTANTE ':' bloques ;
Este no terminal se encarga de la recursión de la sentencia SWITCH ya que dentro de la misma puede haber varias sentencias CASE.

condicion : condicion operador expresion | expresion operador termino {this.sintactico.showMessage("Condición");} ;
El no terminal arriba definido se encarga de reconocer una condición que se utiliza dentro de la sentencia IF.

operador : '<' | '>' | '<=' | '>=' | '<>' | '==' ;
Este no terminal esta definido por los terminales de comparación de expresiones.

expresion : expresion '+' termino {this.sintactico.showMessage("Expresión");}
| expresion '-' termino {this.sintactico.showMessage("Expresión");} | termino ;
El no terminal expresión es recursivo y define las operaciones de suma y resta.

termino : termino '*' factor {this.sintactico.showMessage("Término");}
| termino '/' factor {this.sintactico.showMessage("Término");} | factor ;
El no terminal término es recursivo y define las operaciones de multiplicación y división.

factor : IDENTIFICADOR | CONSTANTE ;
Este no terminal esta definido por dos terminales: identificadores y constantes.

salida : OUT '(' CADENA ')''.' {this.sintactico.showMessage("Sentencia: OUT");} ;
Este no terminal define como se reconoce una salida por pantalla.

tipo : LONG | DOUBLE ;
El no terminal tipo esta definido por dos terminales que son constantes: long y double.

Errores Sintácticos considerados

Hay ciertas reglas de error que se agregaron a la gramática básica definida anteriormente. De todas maneras no se pusieron reglas de error a las cuales les falta el punto al final, o partes significativas de la sentencias porque surgen problemas de sincronización. Es decir, los puntos al final de las sentencias no se consideraron como posibles errores. Ejemplo: a=5. (es válido y lo reconoce) a=5 (no es válido, porque le falta el punto final, y no será reconocido).

Los errores considerados se detallan con respecto a cada uno de los no terminales asociados:

sentencia_if_else :

- Falta '('
- Falta 'condicion'
- Falta ')'
- Falta 'THEN'
- Falta 'ELSE'

sentencia_if :

- Falta '('
- Falta 'condicion'
- Falta ')'
- Falta 'THEN'

sentencia_switch :

- Falta '('
- Falta 'IDENTIFICADOR'
- Falta ')'
- Falta '{'

rep_switch :

- Falta ':'

declaracion_funcion :

- Falta 'tipo'
- Falta 'FUNCTION'
- Falta 'IDENTIFICADOR'
- Falta '{'

llamado_funcion :

- Falta '('
- Falta ')'

asignacion :

- Falta '='

declaracion :

- Falta ':'
- Falta 'tipo'

Conclusiones

Luego de mucho trabajo se logró finalizar el Analizador Lexico y el Analizador Sintáctico del compilador, tal como era solicitado en la consigna de los trabajos.

El desafío principal fue determinar correctamente la gramática para que esta no tenga ambigüedad.

Esto se vio al determinar las reglas de la gramática, se encontraron inconsistencias y se las debió reformular muchas veces, así como la definición de errores, no hay muchos pero engloban la mayoría de los casos en los que aparecen como para lograr detectarlos.