Diseño de compiladores 1

Informe Compilador

Trabajo Práctico 1 y Trabajo Práctico 2

Grupo 26

Alvarez, Maximiliano

maxi25294@gmail.com

Attilio, Nicolas

nattilio@alumnos.exa.unicen.edu.ar

Gonzalez, Francisco

gonzalezfranl10@gmail.com

Introducción

El objetivo de este trabajo es realizar un compilador completo. En esta primera entrega se tratarán el analizador léxico y el sintáctico.

El analizador léxico se encarga de leer el programa fuente y agrupar los caracteres en unidades llamadas tokens. El token es una secuencia de caracteres que forman una unidad significativa, este tiene un ID y un Lexema o valor léxico, el analizador léxico es el encargado de hacer la correspondencia entre cada tipo de token y el número entero. Existen tokens que se corresponden con un único lexema y otros que pueden representar lexemas diferentes, es por esto que el A.L debe enviar información adicional al sintáctico, en la práctica, la información adicional para cada token se almacena en la Tabla de Símbolos (estructura de datos que contiene un registro para cada identificador). Este analizador entrega al sintáctico los números enteros que corresponden a cada tipo de token.

El analizador sintáctico agrupa los tokens del programa fuente en frases gramaticales que el compilador usará en las siguientes etapas. Este obtiene una cadena de tokens del A.L., y verifica que la cadena de tokens presentes en la entrada pueda ser generada mediante la gramática del lenguaje.

Temas Asignados

- **Enteros largos sin signo:** Constantes enteras con valores entre 0 y $2^{32} - 1$. .Se debe incorporar a la lista de palabras reservadas la palabra ULONG.
- **Dobles:** Números reales con signo y parte exponencial. El exponente comienza con la letra E (mayúscula) y el signo es opcional. La ausencia de signo, implica exponente positivo. La parte exponencial puede estar ausente.
  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 13. 0. 1.2E10 Considerar el rango 2.2250738585072014E-308 < x < 1.7976931348623157E+308 -1.7976931348623157E+308 < x < -2.2250738585072014E-308 0.0 Se debe incorporar a la lista de palabras reservadas la palabra DOUBLE.
- Incorporar a la lista de palabras reservadas la palabra REPEAT.
- **Comentarios de 1 línea:** Comentarios que comiencen con "**" y terminen con el fin de línea.
- **Cadenas multilínea:** Cadenas de caracteres que comiencen y terminen con " % " . Estas cadenas pueden ocupar más de una línea, y en dicho caso, al final de cada línea, excepto la última, y al comienzo de cada línea de continuación debe aparecer un signo " + ". (En la Tabla de símbolos se guardará la cadena sin los signos „+", y sin los saltos de línea.

  Ejemplo: % ¡Hola +

  + mundo!%

- **Tema 18 en TP1: Precondiciones con CATCH.**

Declaraciones de Funciones:

- A continuación de los encabezados de declaraciones de funciones, se podrá incluir una precondición con la siguiente estructura:

  PRE: (<condición>);

Donde:

<condición> se definirá como las condiciones descriptas para la sentencia de selección.

Ejemplo:

```
FUNC INT f(INT x)
SINGLE y, z;
BEGIN
   PRE: (x > 3);
   RETURN (x + 1);
END;
```

Sentencias ejecutables:

- Incorporar la siguiente estructura:

  TRY

  <sentencia_ejecutable> /// No se permiten bloques TRY CATCH anidados

  CATCH

  BEGIN

  <bloque de sentencias_ejecutables>

  END

Ejemplo:

```
TRY
      x := f(z) * 4;
CATCH
BEGIN
      x := 10;
END;
```

Atención: Se debe incorporar al Análisis Léxico el reconocimiento de las palabras reservadas **PRE**, **TRY** y **CATCH** y el símbolo ':'.

- **Tema 23 en TP1: Asignación de funciones a variables con TYPEDEF.**

Sentencias declarativas:

- Incorporar la declaración de nuevos tipos, con la siguiente estructura:

  TYPEDEF ID = <encabezado_de_funcion>;

  Donde:

  <encabezado_de_funcion> tendrá la siguiente estructura:

  <tipo> FUNC (<tipo>)

- Incorporar la declaración de variables para el tipo definido por el usuario.

  Ejemplo:

```
TYPEDEF f1 = INT FUNC (INT);
f1 x, y, z;
```

Sentencias ejecutables:

- Sin cambios

Atención: Se debe incorporar al Análisis Léxico el reconocimiento de la palabra reservada TYPEDEF y el símbolo "=".

Decisiones de diseño e implementación

Analizador léxico:

Utilizamos el lenguaje de programación Java debido a que es el lenguaje con el que estamos más familiarizados y además, en dicho lenguaje las herramientas utilizadas ya se han probado y funcionan.

A la hora de diseñar el proyecto, se decidió la implementación de las siguientes clases:

Léxico: Contiene el código fuente, la posición actual en dicho código, las matrices (tanto de transición como de acción), la declaración de los tokens y una lista de palabras reservadas. Lo más importante en esta clase es su método getToken(), que va a ir leyendo el código hasta detectar un token y retornarlo, todo esto mediante las matrices.

Acción: Contiene un string buffer que recibe caracteres de a uno a la vez hasta formar un token. Al utilizar se Java debe tener una clase abstracta Acción de la cual heredan todas las acciones semánticas y todos los errores. Esto permite que la matriz de Acción las pueda contener. Las acciones semánticas que heredan se pueden apreciar en el diagrama de transición de estados, y los errores también, ya que cubren las opciones que no están contempladas en este (por ejemplo si viene un carácter para el cual no hay definida acción semántica)

Tabla de Símbolos: Desde la necesidad de tener una estructura de datos dinámica que pueda tener dos valores (número token y lexema en caso de necesitarlo) se crea esta clase que tiene dentro una tabla de Hash y métodos relacionados a su comportamiento. La clave de dicho Hash es el lexema, y el valor, el código del token. Para aquellos que no necesiten ambos, tendrán el primero nulo.

Analizador Sintactico:

El analizador sintáctico está basado en la herramienta YACC, la cual necesita que se le brinde una gramática en un archivo con extensión .y para funcionar. En dicha gramática se contemplan las consignas estipuladas en el segundo trabajo práctico, y la versión final es el resultado de la corrección de los errores encontrados, tanto del tipo reduce/reduce, como del tipo shift/reduce.

A la hora del manejo de errores, se busca impedir que la ejecución se interrumpa, y se hizo principal foco en contemplar sentencias con elementos faltantes, informando acerca de este elemento que falta y continuando sin interrupciones. También se hace uso del token especial de YACC llamado 'error', muy utilizado en manejo de errores, ya que ante algo no contemplado la herramienta arroja un 'syntax error' que imposibilita seguir leyendo.

Ejemplo manejo de errores:

```
programa : IDENTIFICADOR bloque_declarativo bloque_ejecutable

           |error_programa

            ;

error_programa : bloque_declarativo bloque_ejecutable

             | IDENTIFICADOR bloque_ejecutable bloque_declarativo

              ;
```

Como se puede observar en el ejemplo, se contempla el hecho de que no se ingrese nombre al programa, o que los bloques no están en el orden correspondiente. En cada caso se le notifica al usuario mediante impresiones por pantalla.

Algo particular además de las impresiones, es la implementación de un método que chequea que un signo '-' sea de un número negativo y no perteneciente a una operación (operando). Esto lo hacemos desde la misma gramática, donde se detallan los factores:

```
factor        : CTE_DOUBLE

       | CTE_ULONG

        | '-' factor {chequearFactorNegado();}

       | IDENTIFICADOR

       ;
```

Como en nuestras consideraciones especiales se encuentran las constantes ULONG (enteros largos sin signo) y DOUBLE (coma flotante) hay tres posibilidades. La primera es que el ULONG tenga signo negativo, lo cual es un error y en ese caso optamos por desechar dicho. La segunda opción es que el DOUBLE sea negativo por lo que se debe chequear rangos, y si está dentro de los valores establecidos se deberá modificar su valor en la tabla de símbolos. La tercera opción es que el factor sea un IDENTIFICADOR, y la consideración que se toma es que se permita tener identificadores negados.

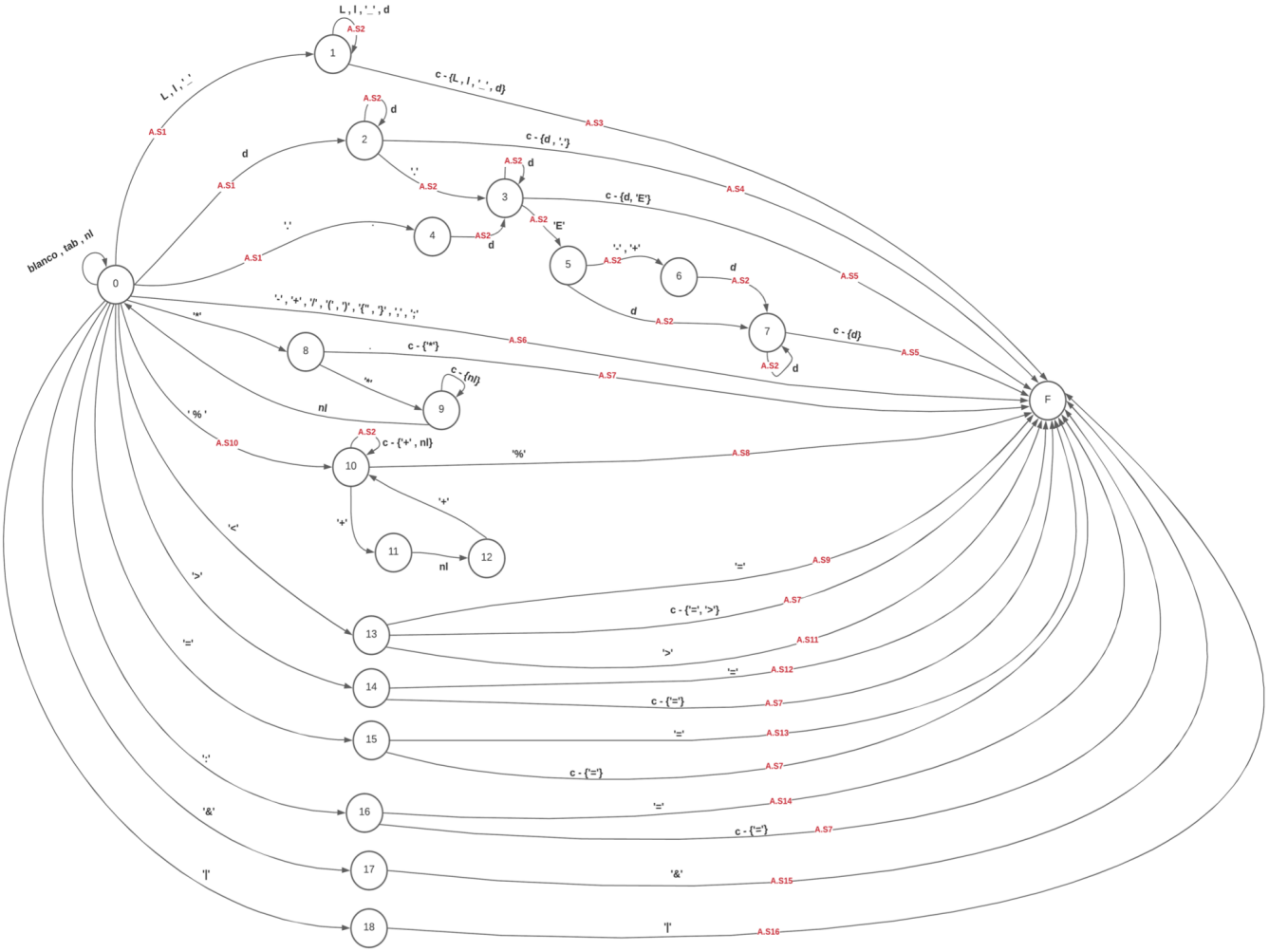
Otra decisión de implementación es agregar la palabra ENDREPEAT como palabra reservada para indicar el cierre de una sentencia de control, ya que vemos la necesidad de que esto esté claro y en las consignas no se hace mención. De esta forma, la sentencia de control queda con el siguiente formato:

```
control : REPEAT'('IDENTIFICADOR ASIGNACION CTE_ULONG';'condicion';' CTE_ULONG')' bloque_control
ENDREPEAT
```

Diagrama de transición de estados

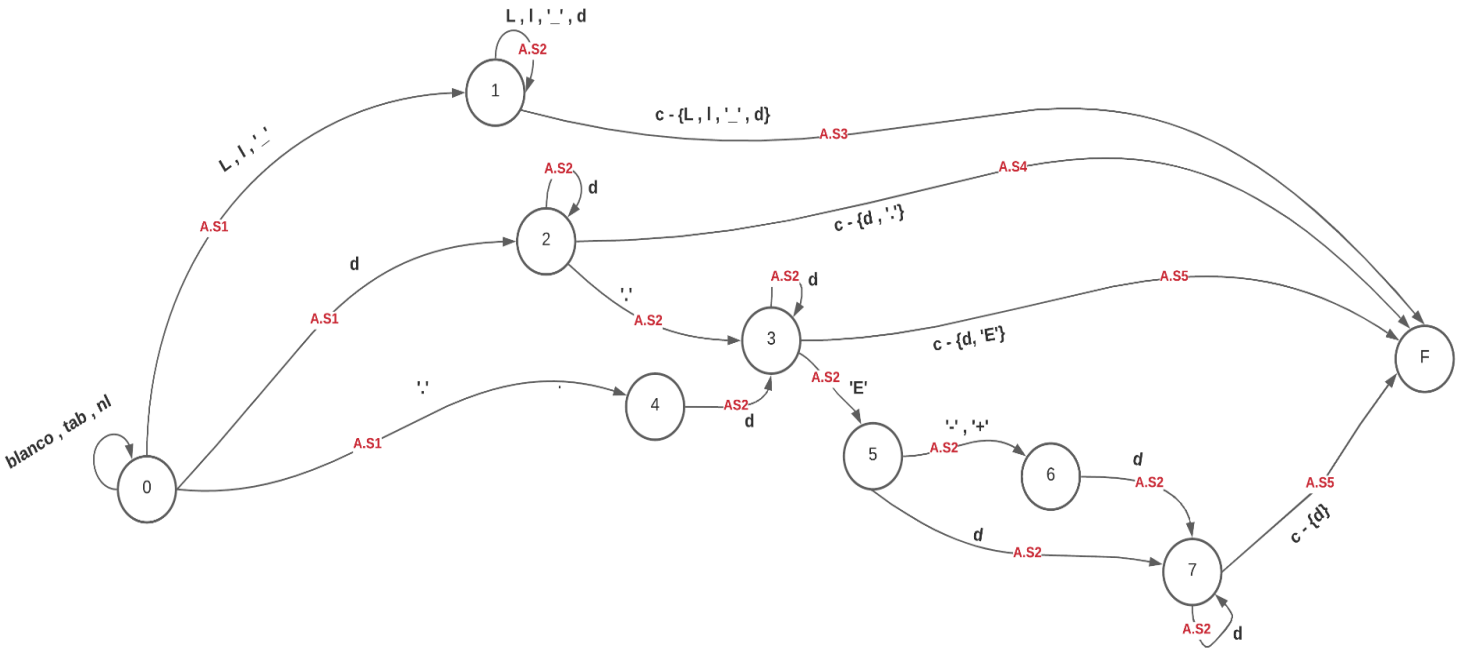
c = Conjunto de caracteres aceptados por el lenguaje.

Por extensión **c = {** l , L , d ,'E', '_' , '+' , '-' , '*' , '/' , ':=' , '>=' , '<=' , '==' , '<>' , '&&' , '||' , '(' , ')' , ',', ';' , nl , blanco , tab **}**

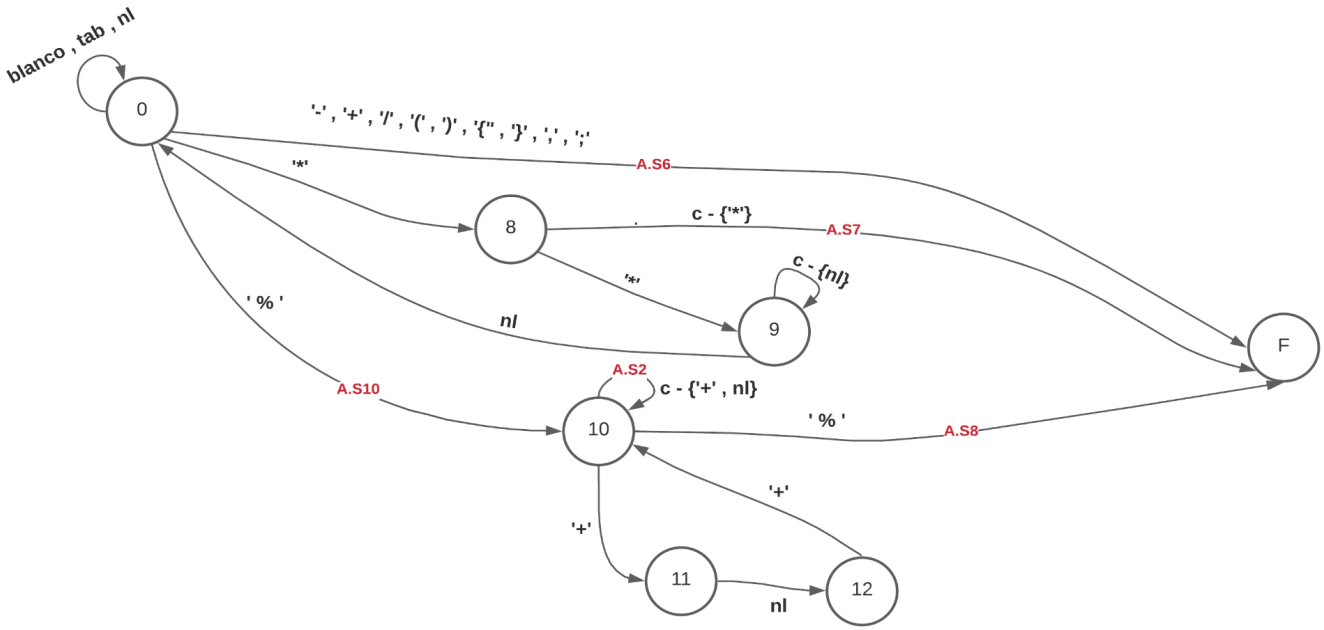


Detallado por conjunto de Tokens:

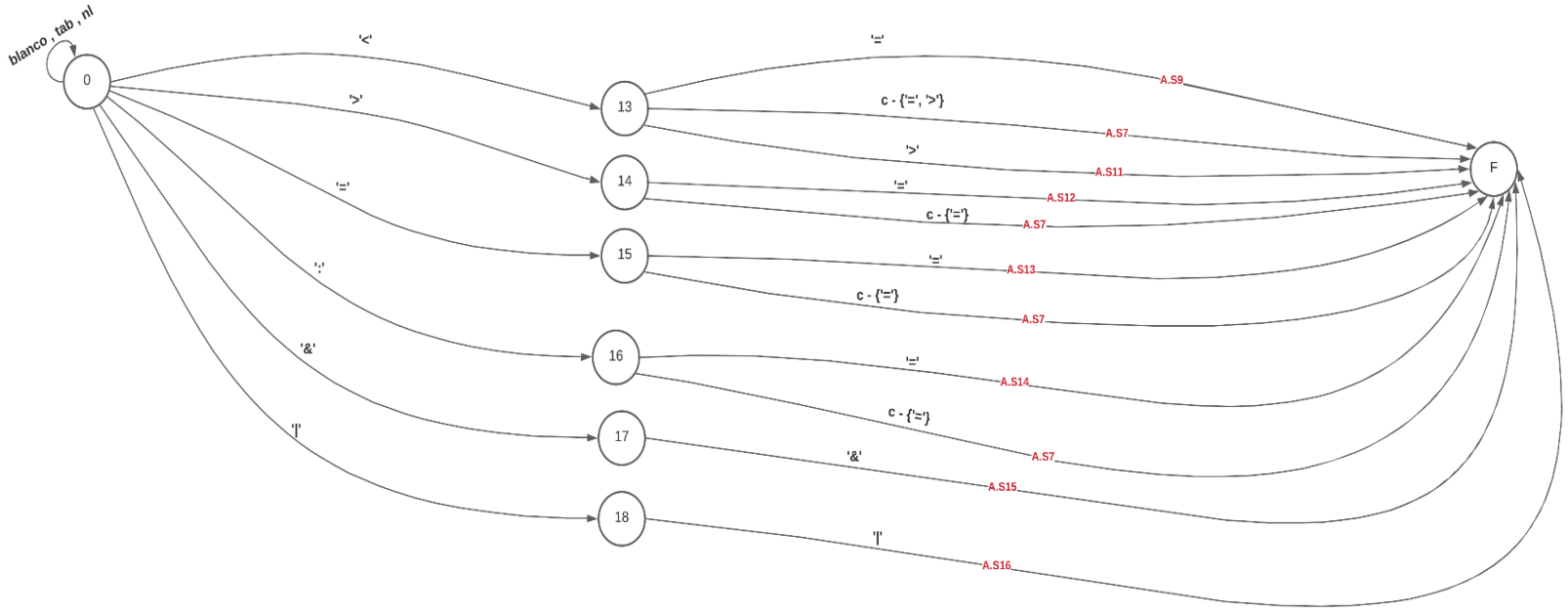
Identificador, Palabra Reservada, Double, Ulong:



Caracteres simples, Comentario de 1 línea, Cadena Multilínea :



Mayor, Mayor Igual, Distinto, Menor, Menor Igual, Asignación, And, Or:



Matriz de transición de estados

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	L	l	d	E	_	+	-	%	=	<	>	*	:	&	|	.	blanco	tab	\n	otros
0	1	1	2	1	1	F	F	10	15	13	14	8	16	17	18	4	0	0	0	F
1	1	1	1	1	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
2	F	F	2	F	F	F	F	F	F	F	F	F	F	F	F	3	F	F	F	F
3	F	F	3	5	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
4	-1	-1	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	7	-1	-1	6	6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	F	F	7	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
8	F	F	F	F	F	F	F	F	F	F	F	9	F	F	F	F	F	F	F	F
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	0	9
10	10	10	10	10	10	11	10	F	10	10	10	10	10	10	10	10	10	10	-1	10
11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	12	-1
12	-1	-1	-1	-1	-1	10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
13	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
14	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
15	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
16	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	F	-1	-1	-1	-1	-1	-1
18	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	F	-1	-1	-1	-1	-1

Lista de acciones semánticas asociadas a las transiciones del autómata

A.S1: Inicializa el buffer y agrega el primer carácter al mismo

A.S2: Agrega un carácter al buffer.

A.S3: Devuelve a la entrada el último carácter leído. Verifica en la tabla de Palabras Reservadas si esta lo que el buffer tiene dentro. Si está devuelve el id de la PR. Si no está busca en la Tabla de Símbolos y si está devuelve el id + el lexema, si no está en la tabla de símbolos lo agrega y devuelve su id +lexema

A.S4: Devuelve a la entrada el último carácter leído. Verifica que la constante se encuentre dentro del rango 0 y 2^32 -1. En caso de que cumpla , agrega el token a la tabla de símbolos y lo retorna. En caso de que no cumpla se genera un error.

A.S5: Devuelve a la entrada el último carácter leído. Verifica que la constante se encuentre dentro del rango dado. Si cumple el rango lo da de alta en la tabla de símbolos, sino genera un error.

A.S6: Reconocer literal y devolver token.

A.S7: Devuelve a la entrada el último caracter leído. Retorna el token correspondiente al carácter leído anteriormente (antes del carácter que es devuelto).

A.S8: Da de alta en la tabla de símbolos el token correspondiente a la cadena y devuelve su identificador junto con su lexema.

A.S9: Retorna el token con el número de token correspondiente al '<='.

A.S10: Inicializa el buffer en vacío.

A.S11: Retorna el token con el número de token correspondiente al '<>'.

A.S12: Retorna el token con el número de token correspondiente al '>='.

A.S13: Retorna el token con el número de token correspondiente al '=='.

A.S14: Retorna el token con el número de token correspondiente al ':='.

A.S15: Retorna el token con el número de token correspondiente al '&&' .

A.S16: Retorna el token con el número de token correspondiente al '||' .

Descripción del mecanismo empleado para implementar la matriz de transición de estados y la matriz de acciones semánticas

La implementación de ambas matrices está hecha en base al diagrama de transición de estados (estos se pueden representar por medio de matrices). En el caso de la matriz de transición de estados, es una traducción del autómata, que tiene en cuenta el estado en el que se encuentra y el símbolo que recibe para determinar el siguiente estado. En cuanto a la matriz de acciones se toman más consideraciones, ya que si bien varias de estas acciones están expresadas como A.S en el diagrama, a nivel código se deben establecer ciertos comportamientos para los casos que ahí no están contemplados, quedando de la siguiente manera:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	L	l	d	E	_	+	-	%	=	<	>	*	:	&	|	.	blanco	tab	\n	otros
0	as1	as1	as1	as1	as1	as6	as6	as10	null	null	null	null	null	null	null	as1	null	null	null	as6
1	as2	as2	as2	as2	as2	as3	as3	as3	as3	as3	as3	as3	as3	as3	as3	as3	as3	as3	as3	as3
2	as4	as4	as2	as4	as4	as4	as4	as4	as4	as4	as4	as4	as4	as4	as4	as2	as4	as4	as4	as4
3	as5	as5	as2	as2	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5
4	err1	err1	as2	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1
5	err2	err2	as2	err2	err2	as2	as2	err2	err2	err2	err2	err2	err2	err2	err2	err2	err2	err2	err2	err2
6	err1	err1	as2	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1	err1
7	as5	as5	as2	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5	as5
8	as7	as7	as7	as7	as7	as7	as7	as7	as7	as7	as7	null	as7	as7	as7	as7	as7	as7	as7	as7
9	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
10	as2	as2	as2	as2	as2	null	as2	as8	as2	as2	as2	as2	as2	as2	as2	as2	as2	as2	err3	as2
11	err4	err4	err4	err4	err4	err4	err4	err4	err4	err4	err4	err4	err4	err4	err4	err4	err4	err4	null	err4
12	err3	err3	err3	err3	err3	null	err3	err3	err3	err3	err3	err3	err3	err3	err3	err3	err3	err3	err3	err3
13	as7	as7	as7	as7	as7	as7	as7	as7	as9	as7	as11	as7	as7	as7	as7	as7	as7	as7	as7	as7
14	as7	as7	as7	as7	as7	as7	as7	as7	as12	as7	as7	as7	as7	as7	as7	as7	as7	as7	as7	as7
15	as7	as7	as7	as7	as7	as7	as7	as7	as13	as7	as7	as7	as7	as7	as7	as7	as7	as7	as7	as7
16	as7	as7	as7	as7	as7	as7	as7	as7	as14	as7	as7	as7	as7	as7	as7	as7	as7	as7	as7	as7
17	err5	err5	err5	err5	err5	err5	err5	err5	err5	err5	err5	err5	err5	as15	err5	err5	err5	err5	err5	err5
18	err6	err6	err6	err6	err6	err6	err6	err6	err6	err6	err6	err6	err6	err6	as16	err6	err6	err6	err6	err6

null refleja que no hay una acción asignada, y los errores los casos no contemplados.

Estas dos matrices están declaradas en la clase Léxico, y en el caso de la matriz de acciones, está implementada como una matriz que contiene elementos de la clase Acción (abstracta de la que heredan las clases AccionSemanticaX y ErrorX)

Errores léxicos considerados

El manejo de los errores léxicos se hace mediante la implementación como lo describe la anterior matriz, con las siguientes acciones:

err0 = Se ingresa un caracter invalido.

err1 = Se espera un dígito pero se recibe un caracter que no lo es

err2 = Se espera un dígito, un '+' , o un '-' y recibe otro caracter

err3 = Se espera un '+' antes o después de un salto de línea.

err4= Se espera un salto de línea después de un '+'.

err5= Se espera un '&' y recibe otra cosa.

err6 = Se espera un '|' y recibe otra cosa.

Errores sintácticos considerados

Los errores sintácticos se manejan, como fue explicado en las consideraciones de diseño e implementación, mediante el token 'error' de YACC y mediante los siguientes no terminales:

error_programa: se debe indicar un nombre para el programa o las palabras de las sentencias declarativas están mal posicionadas.

error_bloque_ejecutable: se detecto un END pero falta un BEGIN para iniciar el bloque y viceversa.

error_ejecucion: Falta de ";" en las sentencias de control, selección, impresión, asignación e invocación

error_declaracion: sentencia mal declarada, falta ";" o falta indicar el tipo de la función

error_control: error por falta de "(", ")", ";", "=", condición, bloque de sentencias, constante, etc.

error_sentencia_control: Falta de ";"

error_asignacion: falta identificador de lado izquierdo de la asignación, falta ":=" o falta expresión aritmética del lado derecho de la asignación

error_seleccion: falta IF, "(", ")", condición, "{", "}", THEN, bloque de sentencias, ENDIF, ELSE.

error_invocacion: falta identificador, "(", ")" o el parámetro.

error_lista_de_variables: falta "," entre los identificadores

error_funcion: falta FUNC, nombre, "(", ")", parámetro o bloque de sentencias.

error_parametro: falta el nombre del identificador.

error_bloque_funcion: Falta el TRY, error en una función, la sentencia ejecutable después del TRY, falta el CATCH.

error_bloque_ejecucion_funcion: Falta BEGIN, bloque de sentencias ejecutables, RETURN, "(", ")", indicar el retorno, ";", END, y mismo caso con PRE.

Descripción del proceso de desarrollo del Analizador Sintactico

Si volvemos atrás en el proceso de desarrollo del analizador sintáctico debemos comenzar con el planteo de la gramática. Esta fue creada partiendo del programa en general, y a partir de este se fue especificando cada parte de este, especificando errores y siguiendo nuestras consignas especiales. En este proceso surgieron algunos inconvenientes como la precedencia de operadores, que se solucionó utilizando varias expresiones (otra alternativa era el uso de 'left'). También nos encontramos varias veces con errores del tipo reduce/reduce (cuando tiene dos reducciones posibles) y del tipo shift/reduce (cuando puede avanzar o reducir), para solucionarlos tuvimos que reescribir la gramática varias veces. Además también nos encontramos con el error de tipo 'syntax' que concluimos que se da cuando la gramática no ofrece salida ante un determinado error no contemplado. Esto se solucionó con el token error provisto por YACC (ya nombrado anteriormente).

Finalmente una vez creado los archivos java, a la hora de las pruebas nos encontramos con errores que no aparecieron en consola, que tuvimos que ir modificando mediante diversas pruebas para llegar a la última versión de la gramática final.

Lista de no terminales

programa: regla inicial de la gramática

bloque_declarativo: bloque que puede contener una o más declaraciones

bloque_ejecutable: bloque que comienza con un BEGIN, dentro tiene otros bloques ejecutables y finaliza con un END.

bloque_sentencias: bloque de sentencias que puede contener una sentencia_ejecucion o más.

sentencia_ejecucion: sentencia ejecutable que puede ser de control, selección, impresión, invocación o asignación

declaración: declaración de función o lista de variables

tipo: ULONG o DOUBLE

control: REPEAT

bloque_control: bloque que puede contener una o más sentencias de control

sentencia_control: puede ser una sentencia_ejecucion o un BREAK;

asignación

selección: IF con una condición, seguido de un THEN y un bloque de sentencias, puede tener ELSE o no y finaliza con ENDIF;

impresión: impresión de cadena

invocación

lista_de_variables

funcion: FUNC con IDENTIFICADOR y parametro, posee bloque_funcion dentro

parámetro: constituido por el tipo y el identificador

funcion_type: FUNC (tipo) y bloque_type

bloque_type: Identificador y una lista de variables

bloque_funcion: bloque_declarativo seguido de bloque_ejecucion_funcion, también puede ser un bloque_declarativo seguido de un TRY, una sentencia_ejecucion, CATCH y un bloque_ejecutable

bloque_ejecucion_funcion: es un bloque de funcion que posee un BEGIN seguido de bloque_sentencias RETURN, una condición y un END, también puede tener PRE.

Precedencias:

condición: expresion u OR

expresión: expresion1 o AND

expresion1: expresion2 o comparador

expresion2: termino o +,-

término: termino * factor o termino / factor, o factor.

factor

comparador: <,>, ==, >=, <=, <>

Conclusiones

Durante el desarrollo de esta parte del compilador entendimos y asociamos ambos analizadores, tanto el léxico como el sintáctico. El hecho de verlo en código y no solamente teórico es una ayuda fundamental a la hora de captarlo. Si bien se tuvieron ciertas dificultades a lo largo de las implementaciones, creemos que pudimos alcanzar el objetivo requerido.