Diseño del analizador léxico, del gestor de errores y de la tabla de símbolos.

Grupo 1

30/11/2011

Contenido

Componentes del grupo

- ✓ Alina Gheorghita
- ✓ Cristina García
- ✓ Pilar Torralbo
- ✓ Tomás Restrepo
- ✓ Guillermo José Hernández
- ✓ Laura Reyero

Identificación de tokens

Clasificamos los tokens que reconocerá nuestro analizador según el tipo al que pertenezcan, de la siguiente manera:

Tipo de token	Codificación
EOF	(FIN, null)
LIT_CARACTER	(LIT_CARACTER, [caracteres])
LIT_CADENA	(LIT_CADENA, [índice tabla])
CADENA	(CADENA, [puntero a tabla])
NUM_REAL	(NUM_REAL, [valor real])
NUM_ENTERO	(NUM_ENTERO, [valor entero])
SEPARADOR	(SEPARADOR, [enumerado asociado])
OP_ARITMETICO	(OP_ARITMETICO, [enumerado asociado])
OP_COMPARACION	(OP_COMPARACION, [enumerado asociado])
OP_LOGICO	(OP_LOGICO, [enumerado asociado])
OP_ASIGNACION	(OP_ASIGNACION, [enumerado asociado])

LIT_CARACTER: secuencia de caracteres que empieza y acaba por comillas simples

LIT_CADENA: secuencia de caracteres que empieza y acaba por comillas dobles

CADENA: secuencia de caracteres que determinará los identificadores y palabras reservadas del analizador.

NUM_REAL: número real.

NUM_ENTERO: número entero.

SEPARADOR: enumerado “.” | ";" | "{" | "}" | "[" | "]" | "#" | "##" | "(" | ")" | "<:" | ":>" | "<%" | "%>" | "%:" | "%:%:"

OP_ARITMETICO: enumerado "+" | "-" | "++" | "--" | "*" | "/" | "%"

OP_COMPARACION: enumerado "==" | "!=" | "<" | ">" | "<=" | ">="

OP_LOGICO: enumerado "&&" | "||" | "!" | "&" | "|" | "~" | "^ " | "<<" | ">>" | "and" | "and_eq" | "bitand" | "bitor" | "compl" | "not" | "not_eq" | "or" | "or_eq" | "xor" | "xor_eq"

OP_ASIGNACIÓN: enumerado "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "^=" | "&=" | "|=" | ">>=" | "<<=" | “->”

Gramática del analizador léxico

Notación "habitual"

literal → numero | cadCar | litBooleano | litString | litPuntero

litPuntero → nullptr

litString → "((CajonDesastre - {",\,\n}) | secEscape | nombredecaracteruniversal})+ "

litBooleano → true | false

cadCar → ' (CajonDesastre - {', \, \n, secEscape, nombredecaracteruniversal})+ '

nombredecaracteruniversal → \u cuartetoHex | \U cuartetoHex cuartetoHex

cuartetoHex → digHex digHex digHex digHex

secEscape → secSimpleEsc | secOctalEsc | secHexEsc

secSimpleEsc → \ , ' | * | ? | \ | a | b | f | n | r | t | v

secOctalEsc → \ , octal | octal octal | octal octal octal

secHexEsc → \x digHex+

numero → entero | real

entero → (decimal | hexadecimal | octal), [sufEntero]

hexadecimal → 0x | 0X , digitoHex+

octal → 0, digOct

decimal → digito | digSinCero digito* [exponente]

sufEntero → (sufSinSigno [sufijoLargo]) | (sufSinSigno [sufijoLargoLargo]) | (sufLargo [sufijoSinSigno]) | (sufLargoLargo [sufijoSinSigno])

sufijoSinSigno → u | U

sufijoLargo → l | L

sufijoLargoLargo → ll | LL

real → **.** digito**+** [exponente] [sufReal] | digito**+ .** digito* [exponente] [sufReal] | digito**+** exponente [sufReal]

exponente → e | E , [+ | -], digito**+**

sufReal → f | l | F | L

identificador → noDigito (noDigito | digito)*

digOct → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

digHex → digito | A | B | C | D | E | F | a | b | c | d | e | f

noDigito → “_” | letra

cajonDesastre → letra | digito | “_” | “{“ | “}” | “[“ | “]” | “#” | “(“ | “)” | “<“ | “>“ | “%” | “:” | “;” | “.” | “?” | “*” | “+” | “-” | “/” | “^” | “&” | “|” | “~” | “!” | “=“ | “,” | “\” | “'“ | “”“

letra → "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" |"R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

digito → digSinCero | 0

digSinCero → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

DELIM → ‘ ’ (blanco) | TAB | EOL | EOF | ‘;’ | ‘|’ | ‘:’ | ‘+’ | ‘-’ | ‘/’ | ‘*’ | ‘<’ | ‘>’ | ‘=’ | ‘&’ | ‘^’ | ‘%’ | ‘!’ | ‘~’ | ‘{‘ | ‘}’ | ‘[‘ | ‘]’ | ‘#’ | ‘(‘ | ‘)’

DELIM2 → DELIM | digito | noDigito

Notación EBNF

literal = numero | cadCar | litBooleano | litString | litPuntero ;

litPuntero = nullptr ;

litString = ‘ “ ‘ , ((cajonDesastre2 | secEscape | nombredecaracteruniversal),
{ (cajonDesastre2 | secEscape | nombredecaracteruniversal}, ‘ ” ‘ ;

litBooleano = true | false ;

cadCar = “ ‘ “, (CajonDesastre - {“ ‘ “, “\”, “\n”, secSalida, nombredecaracteruniversal}),
{CajonDesastre - {‘,\,\n,secEscape,nombredecaracteruniversal}}, “ ‘ “ ;

nombredecaracteruniversal = ‘\u’, cuartetoHex | ‘\U’, cuartetoHex, cuartetoHex ;

cuartetoHex = digHex, digHex, digHex, digHex ;

secEscape = simpleSeq | octalSeq | hexSeq ;

secSimpleEsc = \ , ‘ | * | ? | \ | a | b | f | n | r | t | v ;

secOctalEsc = \ , octal | (octal, octal) | (octal, octal, octal) ;

secHexEsc = '\x', digHex, {digHex} ;

numero = entero | real ;

entero = (decimal | hexadecimal | octal), [sufEntero] ;

hexadecimal = '0x' | '0X' , digHex, {digHex} ;

octal = 0, 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 ;

decimal = digito | digSinCero, {digito}, [exponente] ;

sufEntero = (sufSinSigno [sufijoLargo]) | (sufSinSigno [sufijoLargoLargo]) |
(sufLargo [sufijoSinSigno]) | (sufLargoLargo [sufijoSinSigno]) ;

sufSinSigno = "u" | "U" ;

sufLargo = "l" | "L" ;

sufLargoLargo = 'll' | 'LL' ;

real = (("." , digito, {digito}) | (digito, {digito} , "." , {digito}), [exponente])
| (digito, {digito}, exponente), [sufReal] ;

exponente = e | E , [+ | -], digito, {digito} ;

sufReal = "f" | "l" | "F" | "L" ;

identificador = noDigito, (noDigito | digito)* ;

digHex = digito | A | B | C | D | E | F | a | b | c | d | e | f ;

noDigito = "_" | letra ;

cajonDesastre = letra | digito | "_" | "{" | "}" | "[" | "]" | "#" | "(" | ")" | "<" | ">" | "%" | ":" |
";" | "." | "?" | "*" | "+" | "-" | "/" | "^" | "&" | "|" | "~" | "!" | "=" | "," | "\" | "'" | "'" ;

cajonDesastre2 = letra | digito | "_" | "{" | "}" | "[" | "]" | "#" | "(" | ")" | "<" | ">" | "%" | ":"
| ";" | "." | "?" | "*" | "+" | "-" | "/" | "^" | "&" | "|" | "~" | "!" | "=" | "," | "'" ;

letra = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P"
| "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g"
| "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
"y" | "z" ;

digito = digSinCero | 0 ;

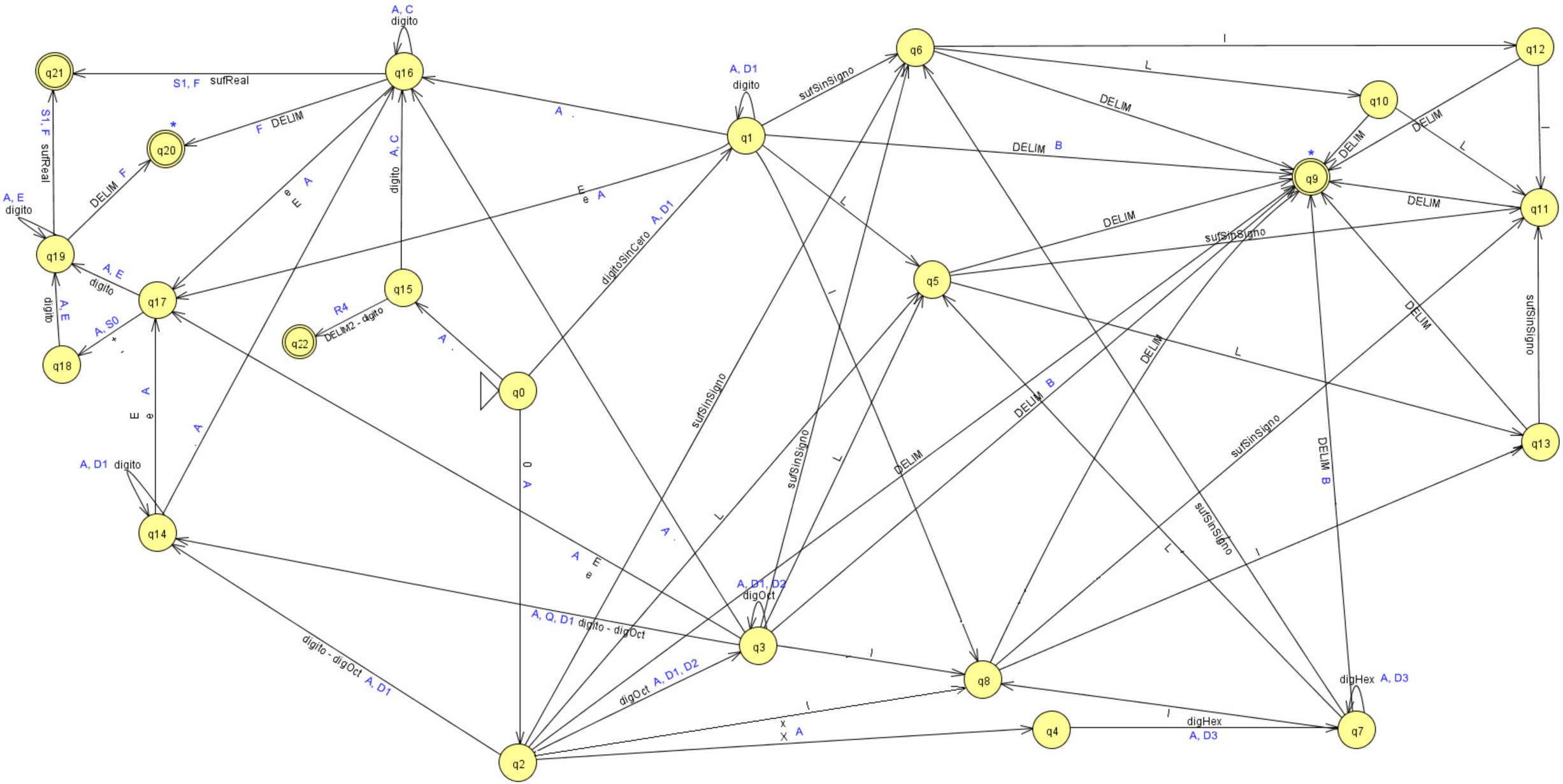
digSinCero = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;

DELIM =‘ ’ (blanco) | TAB | EOL | EOF | ‘;’ | ‘|’ | ‘:’ | ‘+’ | ‘-’ | ‘/’ | ‘*’ | ‘<’ | ‘>’ | ‘=’ | ‘&’ | ‘^’ | ‘%’ | ‘!’ | ‘~’ | ‘{‘ | ‘}’ | ‘[‘ | ‘]’ | ‘#’ | ‘(‘ | ‘)’

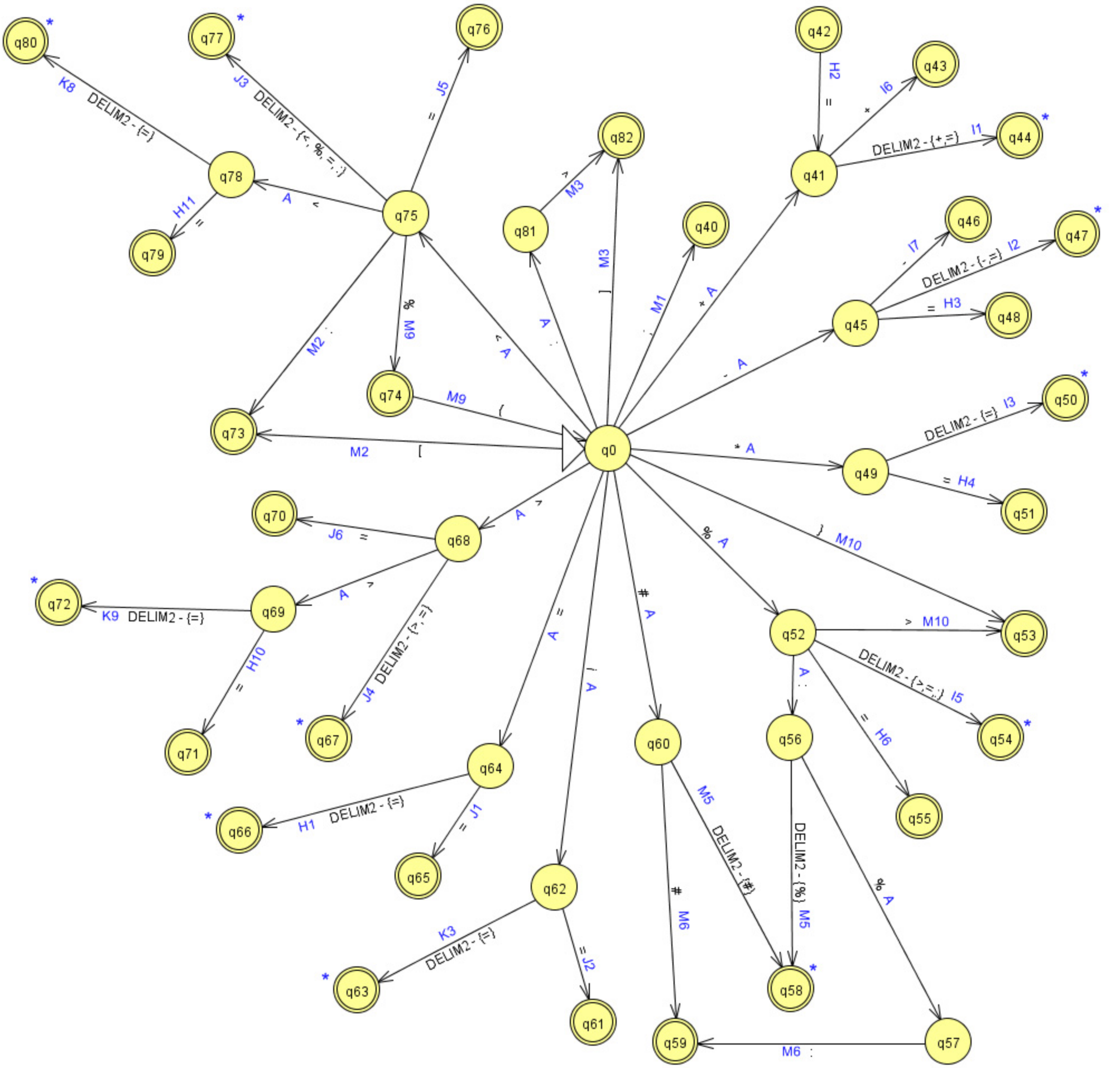
DELIM2 = DELIM | digito | noDigito ;

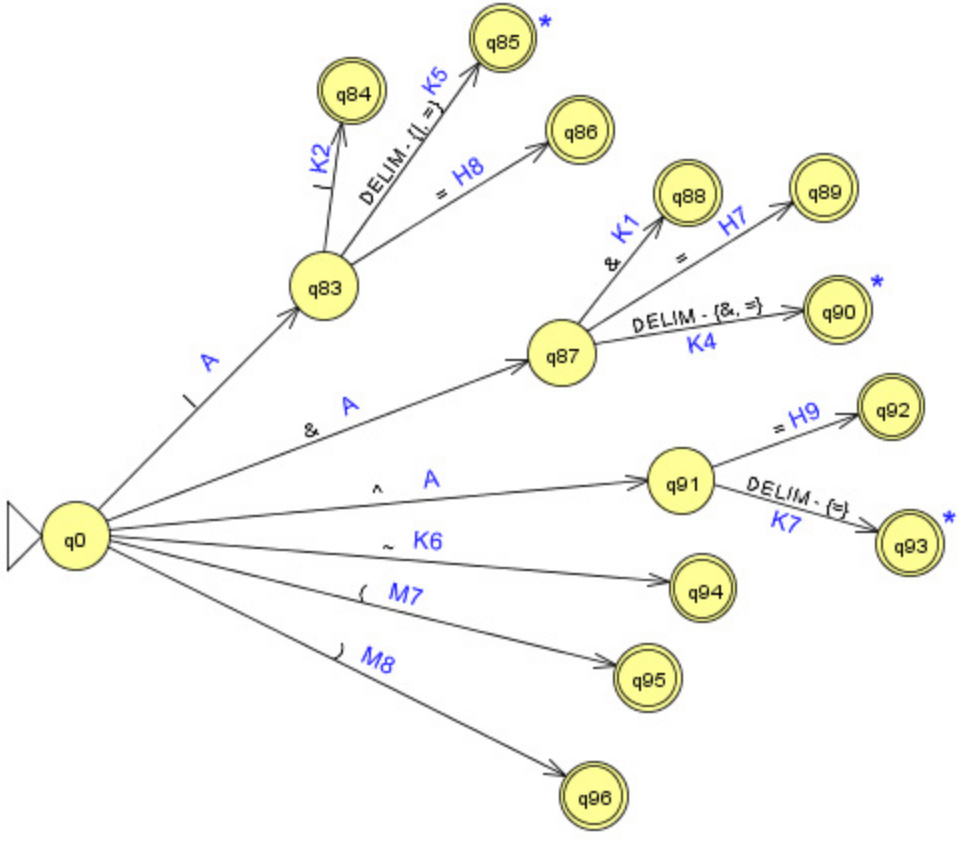
AUTÓMATA FINITO

NÚMEROS

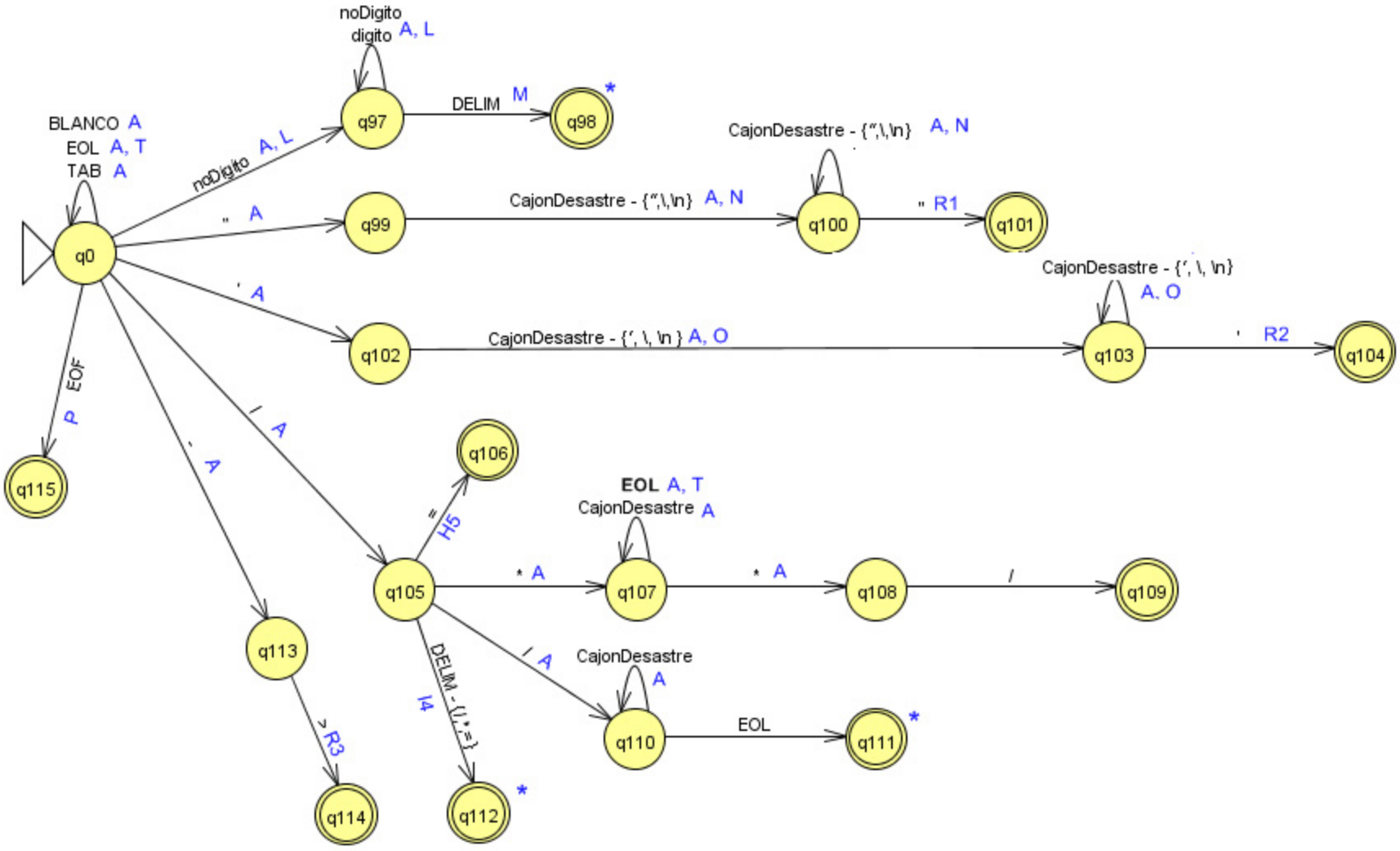


OPERADORES


q0
q1
q2
q3
q4
q5
q6
q7
q8
q9
q10
q11
q12
q13
q14
q15
q16
q17
q18
q19
q20
q21
q22
q23
q24
q25
q26
q27
q28
q29
q30
q31
q32
q33
q34
q35
q36
q37
q38
q39
q40
q41
q42
q43
q44
q45
q46
q47
q48
q49
q50
q51
q52
q53
q54
q55
q56
q57
q58
q59
q60
q61
q62
q63
q64
q65
q66
q67
q68
q69
q70
q71
q72
q73
q74
q75
q76
q77
q78
q79
q80
q81
q82
DELIM2 - {=}
DELIM2 - {<, %, =, ;}
DELIM2 - {+, =}
DELIM2 - {-, =}
DELIM2 - {=}
DELIM2 - {>, =}
DELIM2 - {#}
DELIM2 - {%}
K8
J3
J5
H2
I6
I1
H11
A
M3
M1
I7
I2
H3
M2
6W %
M9
I3
H4
M10
J6
K9
H10
J4
I5
H6
K3
J2
H1
J1
M5
M6
M2


q0
A
q83
K2
q84
DELIM - {|, =}
K5
q85
*
= H8
q86
& A
q87
& K1
q88
= H7
q89
DELIM - {&, =}
K4
q90
*
^ A
q91
= H9
q92
DELIM - {=}
K7
q93
*
~ K6
q94
(M7
q95
) M8
q96

RESTO DE TOKENS



Acciones semánticas

Reconocimiento de números

INIC: Código de inicialización de variables

A: Lee el siguiente carácter de la entrada (código fuente)

***** : Retrocede una posición el puntero de lectura del fichero fuente (devuelve el último carácter leído a la entrada)

B: Genera un token de número entero para el analizador sintáctico

C: Conversión y adición del carácter de preanálisis al valor acumulado de la parte decimal del número.

D1: Conversión y adición del carácter de preanálisis al valor acumulado de la parte entera del número. (base 10)

D2: Conversión y adición del carácter de preanálisis al valor acumulado de la parte entera del número. (base 8)

D3: Conversión y adición del carácter de preanálisis al valor acumulado de la parte entera del número. (base 16)

E: Conversión y adición del carácter de preanálisis al valor acumulado de la parte exponencial del número

S0: Guarda el signo de la E para los enteros (por ejemplo un 1 o un -1).

S1: Aplica un sufijo float a un número real.

S2: Aplica un sufijo entero a un número entero.

F: Genera un token de número real para el analizador sintáctico.

```
{
token = GeneraToken(NUM_REAL, parteEntera + parteDecimal + parteExponencial * signo + sufijoReal);
return token;
}
```

G: Genera un token de número entero para el analizador sintáctico

```
{
token = GeneraToken(NUM_ENTERO,parteEnteraDecimal + parteEnteraOctal + parteEnteraHexadecimal + sufijoEntero);
return token;
}
```

Reconocimiento de operadores

H1: Genera un token de operador de **asignación "="** para el analizador sintáctico.
{
token = GeneraToken (OP_ASIG, ASIG); //Valor del enumerado ASIG, si decidimos juntar los 3 tipos de operadores de asignacion en uno
return token;
}
H2: Genera un token de operador de **asignación con suma "+="** para el analizador sintáctico.
{
token = GeneraToken (OP_ASIG, ASIGSUM);
return token;
}
H3: Genera un token de operador de **asignación con resta "-="** para el analizador sintáctico.
H4: Genera un token de operador de **asignación con multiplicacion "*="** para el analizador sintáctico.
H5: Genera un token de operador de **asignación con division "/="** para el analizador sintáctico.
H6: Genera un token de operador de **asignación con modulo "%="** para el analizador sintáctico.
H7: Genera un token de operador de **asignación con and "&="** para el analizador sintáctico.
H8: Genera un token de operador de **asignación con or "|="** para el analizador sintáctico.
H9: Genera un token de operador de **asignación con xor "^="** para el analizador sintáctico.
H10: Genera un token de operador de **asignación ">>="** para el analizador sintáctico.

H11: Genera un token de operador de **asignacion "<<="** para el analizador sintáctico.

I1: Genera un token de operador **aritmetico suma "+"** para el analizador sintáctico.
{
token = GeneraToken (OP_ARIT, SUMA); //Valor del enumerado SUMA
return token;
}
I2: Genera un token de operador **aritmetico resta "-"** para el analizador sintáctico.
I3: Genera un token de operador **aritmetico multiplicacion "*"** para el analizador sintáctico.
I4: Genera un token de operador **aritmetico division "/"** para el analizador sintáctico.
I5: Genera un token de operador **aritmetico modulo "%"** para el analizador sintáctico.
I6: Genera un token de operador **aritmetico "++"** para el analizador sintáctico.
I7: Genera un token de operador **aritmetico "--"** para el analizador sintáctico.

J1: Genera un token de operador **comparación igual "=="** para el analizador sintáctico.
{
token = GeneraToken (OP_COMP, IGUAL); //Valor del enumerado SUMA
return token;
}
J2: Genera un token de operador **comparación distinto "!="** para el analizador sintáctico.
J3: Genera un token de operador **comparación menor "<"** para el analizador sintáctico.

J4: Genera un token de operador **comparación mayor ">"** para el analizador sintáctico.
J5: Genera un token de operador **comparación menorIgual "<="** para el analizador sintáctico.
J6: Genera un token de operador **comparación mayorIgual ">="** para el analizador sintáctico.

K1: Genera un token de operador **lógico "&&"** para el analizador sintáctico.
{
token = GeneraToken (OP_LOG, AND); //Valor del enumerado SUMA
return token;
}
K2: Genera un token de operador **lógico "||"** para el analizador sintáctico.
K3: Genera un token de operador **lógico "!"** para el analizador sintáctico.
K4: Genera un token de operador **lógico "&"** para el analizador sintáctico.
K5: Genera un token de operador **lógico "|"** para el analizador sintáctico.
K6: Genera un token de operador **lógico "~"** para el analizador sintáctico.
K7: Genera un token de operador **lógico "^ "** para el analizador sintáctico.
K8: Genera un token de operador **lógico "<<"** para el analizador sintáctico.
K9: Genera un token de operador **lógico ">>"** para el analizador sintáctico.

M1: Genera un token separador **punto y coma ";"** para el analizador sintáctico.
{
token = GeneraToken (SEP, PUNTCOMA);
return token;
}
M2: Genera un token separador **corchete abierto "["** para el analizador sintáctico.
M3: Genera un token separador **corchete cerrado "]"** para el analizador sintáctico.
M4: Genera un token separador **punto y coma ";"** para el analizador sintáctico.
M5: Genera un token separador **almohadilla "#"** para el analizador sintáctico.
M6: Genera un token separador **almohadilla doble "##"** para el analizador sintáctico.
M7: Genera un token separador **paréntesis abierto "("** para el analizador sintáctico.
M8: Genera un token separador **paréntesis cerrado ")"** para el analizador sintáctico.
M9: Genera un token separador **llave abierta "{"** para el analizador sintáctico.
M10: Genera un token separador **llave cerrada "}"** para el analizador sintáctico.

Reconocimiento del resto

L: Añade el carácter leído al final del lexema ya acumulado:
{ Concatena(lexema, preanalisis); }

M: Búsqueda del lexema en la tabla de palabras reservadas y si lo encuentra devuelve el token, si no lo encuentra búsqueda/inserción en la tabla de símbolos y generación del token para el analizador sintáctico:
{
```
        token = TS.getPalRes.Busca(lexema);
        if(token != null)
                return token;
token = TS.Busca(lexema);
if (token == NULL)  {
```

```
token = GeneraToken(ID, TS.Inserta(lexema));
return token;
        }
}
```

N: Añade el carácter leído al final de la cadena con comillas dobles ya acumulado:
{ Concatena(cadenaComDobles(litString), preanalisis); }

O: Añade el carácter leído al final de la lexema cadena con comillas simples ya acumulado:
{ Concatena(cadenaComSimples(cadCar), preanalisis); }

P : Genera un token de fin de la entrada para el analizador sintáctico:

```
{
token = GeneraToken (FIN, NULL);
return token;
}
```

Q: Se queda con la parte calculada en base decimal

R1: Genera un token de tipo LIT_CADENA

R2: Genera un token de tipo LIT_CARACTER

R3: Genera un token flecha (para punteros)

R4: Genera un token punto

T: Incrementa el número de líneas

TABLA DE TRANSICIONES

Para mejor visualización y comprensión, hemos separado la "gran tabla" en varias tablas mas pequeñas, agrupadas al igual que el autómata por el tipo de token que se reconoce (números, operadores, otros).

		digSinCero	0	8	9	digito	noDigito	digOct	digHex	sufSinSigno	sufReal	DELIM	L	l	.	E	e	X	x	+	-
NUMEROS	q0	q1	q2												q32						
	q1					q1				q6		q5	q9	q12	q31	q35	q35				
	q2			q14	q14			q3										q4	q4		
	q3			q14	q14			q3		q14		q9	q5	q8		q17	q17				
	q4								q7												
	q5									q11		q9	q13								
	q6											q9	q10	q12							
	q7								q7	q6		q9	q5	q8							
	q8											q9		q13							
	q10											q9	q11								
	q11											q9									
	q12											q9		q11							
	q13									q11		q9									
	q14					q14									q16	q17	q17				
	q16					q16					q21	q20				q17	q17				
	q15					q16															
	q17					q19														q18	q18
	q18					q19															
	q19					q19					q21	q20									

		0	1	2	3	4	5	6	7	8	9	A	a	B	b	C	c	D	d	E	e	F	f	L	l	U	u	X	x	(let)	.	+	-	(esp)	(tab)	(eol)	(eof)	|	;	*	/	%	=	&	!	<	>	^	DELIM
NUMEROS	q0	q2	q1	q1	q1	q1	q1	q1	q1	q1	q1																				q15																		
	q1	q1	q1	q1	q1	q1	q1	q1	q1	q1	q1	V	V	V	V	V	V	V	V	q17	q17	V	V	q5	q8	q6	q6	V	V	V	q16	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q2	q3	q3	q3	q3	q3	q3	q3	q3	q14	q14	V	V	V	V	V	V	V	V	V	V	V	V	V	q8	V	V	q4	q4	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q3	q3	q3	q3	q3	q3	q3	q3	q3	q14	q14	V	V	V	V	V	V	V	V	q17	q17	V	V	q5	q8	q6	q6	V	V	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q4	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI	VI
	q5	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	q13	V	q11	q11	V	V	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q6	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	q10	q12	V	V	V	V	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q7	q5	q8	q6	q6	V	V	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q8	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	q13	q11	q11	V	V	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q10	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	q11	V	V	V	V	V	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q11	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q12	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	q11	V	V	V	V	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q13	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	q11	q11	V	V	V	V	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9	q9
	q14	q14	q14	q14	q14	q14	q14	q14	q14	q14	q14	V	V	V	V	V	V	V	V	q17	q17	V	V	V	V	V	V	V	V	V	q16	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
	q15	q16	q16	q16	q16	q16	q16	q16	q16	q16	q16	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22
	q16	q16	q16	q16	q16	q16	q16	q16	q16	q16	q16	V	V	V	V	V	V	V	V	q17	q17	q21	q21	q21	q21	V	V	V	V	V	V	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20
	q17	q19	q19	q19	q19	q19	q19	q19	q19	q19	q19	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	q18	q18	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
	q18	q19	q19	q19	q19	q19	q19	q19	q19	q19	q19	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
	q19	q19	q19	q19	q19	q19	q19	q19	q19	q19	q19	V	V	V	V	V	V	V	V	V	V	q21	q21	q21	q21	V	V	V	V	V	V	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20	q20

													DELIM2												
		[	]	{	}	(	)	#	~	+	-	:	/	*	^	<	>	%	=	|	;	!	&	digito	noDigito
OPERADORES	q0	q73	q82	q74	q53	q95	q96	q60	q94	q41	q45	q81		q49	q91	q75	q68	q52	q64	q83	q40	q62	q87		
	q41	I	I	I	I	I	I	I	I	q43	I	I	q44	q44	q44	q44	q44	q44	q42	q44	q44	q44	q44	q44	q44
	q45	I	I	I	I	I	I	I	I	I	q46	I	q47	q47	q47	q47	q47	q47	q48	q47	q47	q47	q47	q47	q47
	q49	I	I	I	I	I	I	I	I	I	I	I	q50	q50	q50	q50	q50	q50	q51	q50	q50	q50	q50	q50	q50
	q52	I	I	I	I	I	I	I	I	I	I	q56	q54	q54	q54	q54	q53	q54	q55	q54	q54	q54	q54	q54	q54
	q56	I	I	I	I	I	I	I	I	I	I	I	q58	q58	q58	q58	q58	q57	q58	q58	q58	q58	q58	q58	q58
	q57	II	II	II	II	II	II	II	II	II	II	q59	II	II	II	II	II	II	II	II	II	II	II	II	II
	q60	I	I	I	I	I	I	q59	I	I	I	I	q58	q58	q58	q58	q58	q58	q58	q58	q58	q58	q58	q58	q58
	q62	I	I	I	I	I	I	I	I	I	I	I	q63	q63	q63	q63	q63	q63	q62	q63	q63	q63	q63	q63	q63
	q64	I	I	I	I	I	I	I	I	I	I	I	q66	q66	q66	q66	q66	q66	q65	q66	q66	q66	q66	q66	q66
	q68	I	I	I	I	I	I	I	I	I	I	I	q67	q67	q67	q67	q69	q67	q70	q67	q67	q67	q67	q67	q67
	q69	I	I	I	I	I	I	I	I	I	I	I	q72	q72	q72	q72	q72	q72	q71	q72	q72	q72	q72	q72	q72
	q75	I	I	I	I	I	I	I	I	I	I	q73	q77	q77	q77	q78	q77	q74	q76	q77	q77	q77	q77	q77	q77
	q78	I	I	I	I	I	I	I	I	I	I	I	q80	q80	q80	q80	q80	q80	q79	q80	q80	q80	q80	q80	q80
	q81	III	III	III	III	III	III	III	III	III	III	III	III	III	III	III	q82	III	III	III	III	III	III	III	III
	q83	I	I	I	I	I	I	I	I	I	I	I	q85	q85	q85	q85	q85	q85	q86	q84	q85	q85	q85	q85	q85
	q87	I	I	I	I	I	I	I	I	I	I	I	q90	q90	q90	q90	q90	q90	q89	q90	q90	q90	q88	q90	q90
	q91	I	I	I	I	I	I	I	I	I	I	I	q93	q93	q93	q93	q93	q93	q92	q93	q93	q93	q93	q93	q93

		(eof)	(esp)	(tab)	(eol)	digito	noDigito	digOct	digHex	cajonDesas*		DELIM2	"	'	-	/	=	*	>
RESTO	q0	q115	q0	q0	q0		q97						q99	q102	q113	q105			
	q97	I	I	I	I	q97	q97	I	I	I	I	q98	I	I	I	I	I	I	I
	q99	I	I	I	I	q100	q100	I	I	q100	I	I	I	I	I	I	I	I	I
	q100	I	q100	q100	I	I	I	I	I		I	I	q101	I	I	I	I	I	I
	q102	I	I	I	I	I	I	I	I	q103	I	I	I	I	I	I	I	I	I
	q103	I	I	I	I	I	I	I	I	I	I	I	I	q104	I	I	I	I	I
	q105	I	I	I	I	I	I	I	I	I	I	q112	I	I	I	q110	q106	q107	I
	q107	I	I	I	q107	I	I	I	I	q107	I	I	I	I	I	I	I	q108	I
	q108	IV	IV	IV	IV	IV	IV	IV	IV	IV	IV	IV	IV	IV	IV	q109	IV	IV	IV
	q110	I	I	I	q111	I	I	I	I	q110	I	I	I	I	I	I	I	I	I
	q113	III	III	III	III	III	III	III	III	III	III	III	III	III	III	III	III	III	q114

Tabla de símbolos

Para C++ los ámbitos se definen al abrir llave ({) y cerrar llave (}) y en cualquier zona que se comporte como ámbito, se podrán declarar variables (bucles for, while, registros, case…).

Como decidimos implementar la práctica en Java, podemos usar sus objetos predefinidos, como hash tables, hash map, enum map, listas dinámicas, etc para la gestión de la tabla de símbolos.

Gestión de palabras reservadas.

Dado que las palabras clave son también palabras reservadas, se van a almacenar dentro de la tabla de símbolos al arrancar el compilador.
Vamos a tener una tabla de palabras reservadas donde el campo clave es la cadena de caracteres que identifica a la palabra reservada. Como campo valor almacenaremos un valor numérico que identifique a cada una de las palabras.

Gestión de ámbitos.

Vamos a gestionar nuestra tabla de símbolos con una tabla separada para cada ámbito. Tendremos una tabla de símbolos global, en la que almacenaremos cada identificador encontrado en ese ámbito junto con su tipo, así como una entrada para cada nuevo ámbito que aparezca en el código. Esta entrada tendrá información sobre el número de argumentos y su tipo, la forma de pasar los argumentos, el tipo de retorno y un puntero a la propia tabla de ese ámbito donde estará almacenada la información que él contiene.

En cada una de las tablas asociadas a los ámbitos, aparecerá como último atributo, un puntero que señalará cual es su ámbito padre.

Tendremos un puntero BLOQUE_ACTUAL para facilitar la búsqueda dentro de la tabla de símbolos, y que como su propio nombre indica, apuntará al bloque actual.

Gestión de los errores

Funcionamiento básico

- ✓ Sustituirá a la gestión de errores básica en cualquier generador léxico-sintáctico como Javacc.
- ✓ Debe contabilizar el número de errores Léxicos y el número de errores sintácticos.
- ✓ Ofrecerá métodos distintos para insertar errores léxicos y sintácticos.
- ✓ Un método para imprimir los errores finales.
- ✓ Dos tablas distintas para léxico y sintáctico, cada una con la tabla de errores creada por las distintas transiciones incorrectas, mas errores gramaticales distintos en el sintáctico.
- ✓ Informativo, no soluciona errores.

Tipos

Dos tipos distintos:

- ✓ Error Léxico
- ✓ Error Sintáctico

Principalmente se distinguen en que acceden a dos tablas distintas de errores y que el léxico guarda el carácter que produce el error, mientras que el sintáctico da linea y tipo de error (mucho mejor distinguidos).

Interfaz

- ✓ **void** i**nsertarErrorLex:** Requiere el carácter que ha dado el error, número de linea, número de error. Inserta en una lista un error de tipo Léxico.

- ✓ **void** i**nsertaErrorSin:** Inserta en una lista un error de tipo sintáctico.

- ✓ **int** i**mprimeErrores:**
  - Devuelve un integer: 0 si no han ocurrido errores hasta el momento, 1 en otro caso.
  - Imprime los errores ocurridos de la forma:
    "No hubo errores léxicos"
    "5 errores léxicos:
    "Linea 5. Llego un * se esperaba (lista de valores válidos según el error de la tabla)
    Linea 6..."

La impresión de errores sintácticos incluso mensajes de error más variados: Identificador no declarado, errores de tipos etc.

Tabla de errores léxicos

Asigna distintos String que describen errores léxicos a números de entrada

TABLA DE ERRORES	
I	Carácter inesperado
II	Se esperaba el simbolo ':'
III	Se esperaba el simbolo '>'
IV	Se esperaba el simbolo '/'
V	Numero mal formado
VI	Se esperaba numero en formato Hexadecimal