LENGUAJE DEL PROYECTO

PARTE LÉXICA (EXPRESIONES)

EXPRESIÓN	TERMINAL
\s+	ESPACIOS
\ / \ /.*	COM DE UNA LINEA
\ /*([^"!>"]|[\r|\f|\s|\t|\n])\/**	COM MULTILINEA
"int"	INT
"double"	DOUBLE
"boolean"	BOOLEAN
"char"	CHAR
"string"	STRING
"true"	TRUE
"false"	FALSE
"new"	NEW
"void"	VOID
"print"	PRINT
"println"	PRINTLN
"return"	RETURN
"toLower"	TOLOWER
"toUpper"	TOUPPER
"round"	ROUND
"length"	LENGTH
"typeof"	TYPEOF
"tostring"	TOSTRING
"tochararray"	TOCHARARRAY
"run"	RUN
"if"	IF
"else"	ELSE
"switch"	SWITCH
"case"	CASE
"default"	DEFAULT
"break"	BREAK
"for"	FOR
"while"	WHILE
"do"	DO
"continue"	CONTINUE
"+"	MAS
"-"	MENOS

"*"	MUL
"/"	DIV
"^"	EXP
"%"	MOD
"=="	IGUAL
"!="	DESIGUAL
"<="	MENORIGUAL
">="	MAYORIGUAL
"<"	MENOR
">"	MAYOR
"="	ASIGNAR
":"	DOSPUNTOS
"?"	TERNARIO
"||"	OR
"&&"	AND
"!"	NOT
"("	PARABRIR
")"	PARCIERRE
"["	COABRIR
"]"	CORCIERRE
"{"	LLAVABRIR
"}"	LLAVCIERRE
";"	PUNTOCOMA
","	COMA
\'("\n"|"\\\\"|"\t"|"\r"|\\\'|\\\"|.)\'	CARÁCTER
[0-9]+("."[0-9]+)	DOBLE
[0-9]+	ENTERO
([a-zA-Z_])([a-zA-Z0-9_])*	ID
["] **<string>[^"\\]+** **<string>"\\\""** **<string>"\\n"** **<string>\s** **<string>"\\t"** **<string>"\\\\"** **<string>"\\\'"** **<string>["]**	CADENA (UTILIZA LA FUNCION “ESTADOS” QUE ANALIZADORES COMO BISON O JISON, EN SU DEFECTO, TIENEN EN SUS OPCIONES. REFERENCIA: https://gerhobbelt.github.io/jison/docs/#lexical-analysis

PARTE SINTACTICA (GRAMATICA)

Gramática del Proyecto

G = {N, T, P, S)

N (No terminales) = {INICIO, SENTENCIAS, SENTENCIA, DVARIABLES, LISTAID, AVARIABLES, AARREGLOS, DARREGLOS, UDIMENSION, BDIMENSION, LISTAVALORES, VALORES, DMETODO, DFUNCION, PARAMETRO, PARAMETRO, LLAMADA, LLAMADAS, INSTRUCCIONES, INSTRUCCIÓN, IF, SWITCH, CASES, CASO, DEFAULT, WHILE, DOWHILE, FOR, DVAR, AVAR, RETURN, BREAK, CONTINUE, PRINT, UPPER, LOWER, ROUND, LENGTH, TYPEOF, TOSTRING, TOCHAR, RUN, ENTRADAS, TIPO, PRIMITIVO, EXPRESION}

T (Terminales) = {int, double, boolean, char, string, true, false, new, void, print, println, return, tolower, toupper, round, length, typeof, tostring, tochararray, run, if, else, switch, case, default, break, for, while, do, continue, mas, menos, mul, div, exp, mod, igual, desigual, menorIgual, mayorIgual, menor, mayor, asignar, dospuntos, ternario, or, and, not, parr, parC, corA, corC, llavA, llavC, puntocoma, coma, carácter, doble, entero, identificador, texto}

S (Estado Inicial) = {INICIO}

P (Producciones) = {

INICIO ::= SENTENCIAS EOF

SENTENCIAS::= SENTENCIAS SENTENCIA

| SENTENCIA

SENTENCIA::= DVARIABLES

| DARREGLOS

| DMETODO

| DFUNCION

| RUN

| error puntocoma

DVARIABLES::= TIPO LISTAID asignar EXPRESION puntocoma

| TIPO LISTAID puntocoma

LISTAID::= LISTAID coma identificador

| identificador

AVARIABLES::= identificador asignar EXPRESION puntocoma

AARREGLOS::= identificador corA EXPRESION corC asignar EXPRESION puntocoma

| identificador corA EXPRESION corC corA EXPRESION corC asignar EXPRESION puntocoma

DARREGLOS::= UDIMENSION

UDIMENSION::= TIPO identificador corA corC asignar new TIPO corA EXPRESION corC puntocoma

| TIPO identificador corA corC asignar corA LISTAVALORES corC puntocoma

| TIPO identificador corA corC asignar EXPRESION puntocoma

BDIMENSION::= TIPO identificador corA corC corA corC asignar new TIPO corA EXPRESION corC corA EXPRESION corC puntocoma

| TIPO identificador corA corC corA corC asignar corA VALORES corC puntocoma

LISTAVALORES::= LISTAVALORES coma PRIMITIVO

| PRIMITIVO

VALORES::= VALORES coma corA LISTAVALORES corC

| corA LISTAVALORES corC

DMETODO::= identificador parA parC llavA INSTRUCCIONES llavC

|identificador parA parC dospuntos void llavA INSTRUCCIONES llavC

| identificador parA PARAMETROS parC llavA INSTRUCCIONES llavC

|identificador parA PARAMETROS parC dospuntos void llavA INSTRUCCIONES llavC

DFUNCION::= identificador parA parC dospuntos TIPO llavA INSTRUCCIONES llavC

|identificador parA PARAMETROS parC dospuntos TIPO llavA INSTRUCCIONES llavC

PARAMETROS::= PARAMETROS coma PARAMETRO {

| PARAMETRO

PARAMETRO::= TIPO identificador

LLAMADA::= identificador parA parC puntocoma

| identificador parA ENTRADAS parC puntocoma

LLAMADAS::= identificador parA parC

| identificador parA ENTRADAS parC

INSTRUCCIONES::= INSTRUCCIONES INSTRUCCION

| INSTRUCCION

INSTRUCCION::= DVARIABLES

| AVARIABLES

| DARREGLOS

| AARREGLOS

| RETURN

| LLAMADA

| PRINT

| IF

| SWITCH

| BREAK

| CONTINUE

| WHILE

| DOWHILE

| FOR

| error puntocoma

IF::= if parA EXPRESION parC llavA INSTRUCCIONES llavC

| if parA EXPRESION parC llavA INSTRUCCIONES llavC else llavA INSTRUCCIONES llavC

|if parA EXPRESION parC llavA INSTRUCCIONES llavC else IF

SWITCH::= switch parA EXPRESION parC llavA CASES DEFAULT llavC

CASES::= CASES CASO

| CASO

CASO::= case EXPRESION dospuntos INSTRUCCIONES

DEFAULT::= default dospuntos INSTRUCCIONES

WHILE::= while parA EXPRESION parC llavA INSTRUCCIONES llavC

DOWHILE::= do llavA INSTRUCCIONES llavC while parA EXPRESION parC puntocoma

FOR::= for parA DVAR puntocoma EXPRESION puntocoma AVAR parC llavA INSTRUCCIONES llavC

| for parA AVAR puntocoma EXPRESION puntocoma AVAR parC llavA INSTRUCCIONES llavC

DVAR::= TIPO LISTAID asignar EXPRESION

AVAR::= identificador asignar EXPRESION

RETURN::= return puntocoma
| return EXPRESION puntocoma

BREAK::= break puntocoma

CONTINUE::= continue puntocoma

PRINT::= print parA EXPRESION parC puntocoma
| println parA EXPRESION parC puntocoma

UPPER::= toUpper parA EXPRESION parC

LOWER::= toLower parA EXPRESION parC

ROUND::= round parA EXPRESION parC

LENGTH::= length parA EXPRESION parC

TYPEOF::= typeof parA EXPRESION parC

TOSTRING::= tostring parA EXPRESION parC

TOCHAR::= tochar parA EXPRESION parC

RUN::= run identificador parA parC puntocoma

| run identificador parA ENTRADAS parC puntocoma

ENTRADAS::= ENTRADAS coma EXPRESION

| EXPRESION

TIPO::= int

| double

| boolean

| char

| string

PRIMITIVO::= entero

| doble

| true

| false

| texto

| caracter

| identificador

```
EXPRESION::= EXPRESION mas EXPRESION
        | EXPRESION menos EXPRESION
        | EXPRESION mul EXPRESION
        | EXPRESION div EXPRESION
        | EXPRESION mod EXPRESION
        | EXPRESION exp EXPRESION
        | menos EXPRESION %prec umenos
        | parA EXPRESION parC
        | EXPRESION igual EXPRESION
        | EXPRESION desigual EXPRESION
        | EXPRESION menor EXPRESION
        | EXPRESION menorIgual EXPRESION
        | EXPRESION mayor EXPRESION
        | EXPRESION mayorIgual EXPRESION
        | EXPRESION or EXPRESION
        | EXPRESION and EXPRESION
        | not EXPRESION
        | EXPRESION ternario EXPRESION dospuntos EXPRESION
        | entero
        | doble
        | true
        | false
        | texto
        | caracter
        | identificador
        | parA TIPO parC EXPRESION
```

| EXPRESION mas mas

| EXPRESION menos menos

| identificador corA EXPRESION corC

| identificador corA EXPRESION corC corA EXPRESION corC

| LLAMADAS

| UPPER

| LOWER

| ROUND

| LENGTH

| TYPEOF

| TOSTRING

| TOCHAR

PARTE SEMANTICA (ANALIZADOR)

El funcionamiento de la creación se separa en dos: Su parte lógica y la parte gráfica (se hizo así para evitar conflictos y errores). La parte gráfica busca ir desde los terminales y crear nodos hijos. Gracias a la recursividad del analizador, estos nodos hojas van subiendo a cada no terminal y se almacenan en la lista de hijos que pueda estar manejando ese no terminal. Este sigue subiendo y subiendo hasta que, paulatinamente llega a la raíz (o el no terminal que da inicio al recorrido). Con toda esa estructura, pasa a un analizador que traduce en código DOT los nodos y conexiones que maneja el árbol y esto es enviado al frontend donde se gráfica.

La parte lógica funciona de manera similar. Para evitar tener que cargar de mas trabajo al servidor, se aprovecha que el analizador funciona de manera ascendente (similar a un árbol) y se van analizando y comprimiendo en los datos e información relevante en objetos de tipo instrucción. Los objetos de tipo instrucción varían de acuerdo al tipo de no terminal y manejan información diferente teniendo en común nada mas una variable que las clasifica por su tipo (declaración, asignación, etc.). Cabe aclarar que el lenguaje maneja bloques de instrucciones. La mayoría de instrucciones para no terminales funcionan similar a estos, por lo que se puede ver que una instrucción puede estar conformada por instrucciones que pueden o no traer mas instrucciones dentro (por ejemplo, un ciclo puede tener una instrucción que sirva para definir la condición y una lista de instrucciones que debe ejecutar cuando se llame al método). Lo que sale del analizador cuando finaliza es una lista pequeña con todas las instrucciones que debe de ejecutar.

De forma simple, la lista pasa después a un ciclo que va a ir ejecutando cada una de ellas, retornando los mensajes de error o impresiones que van a ser mostradas en el frontend. De forma detallada, es mas complejo que eso. Parte importante del análisis es el manejo de tipos y entornos en los que se ejecutan las instrucciones. La razón para encapsular el código en super instrucciones es para manejar esto: Se posee un entorno global durante toda la ejecución, y cada vez que se vaya a ejecutar algún bloque se crea un nuevo entorno, conectado al que le precede, y desde ahí se van ejecutando cada una de las sentencias. Al finalizar, el bloque deja de tener uso junto a todos los cambios y eventos que se hayan dado dentro. Aparte de eso, también se verifica constantemente como concuerdan los tipos de la información.