Imagen que contiene dibujo, plato, taza

Descripción generada automáticamenteUniversidad de San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Organización de Lenguajes y Compiladores 2

Guillermo Alfredo Peitzner Estrada – 201504468.

# Manual Técnico

## Lenguaje utilizado

Python 3.8: <https://www.python.org/>.

## Librerías

PyQt5 5.15.0: <https://pypi.org/project/PyQt5/>.

ply 3.11: <https://pypi.org/project/ply/>.

graphviz 0.14: <https://pypi.org/project/graphviz/>.

## Dependencias de terceros

Graphviz: <https://graphviz.org/>.

## Licencia del software

GPL-3.0: <http://www.gnu.org/licenses/gpl-3.0.html>

## Repositorio oficial

<https://github.com/gpeitzner/OLC2_Proyecto2>

## Glosario de Términos

### Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. ​ Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.

### PLY

PLY es una herramienta de análisis escrita exclusivamente en Python. Es, en esencia, una reimplementación de Lex y Yacc originalmente en lenguaje C. Fue escrito por David M. Beazley. PLY utiliza la misma técnica de análisis LALR que Lex y Yacc.

### PyQT5

PyQt es un binding de la biblioteca gráfica Qt para el lenguaje de programación Python. La biblioteca está desarrollada por la firma británica Riverbank Computing y está disponible para Windows, GNU/Linux y Mac OS X bajo diferentes licencias.

### Graphviz

Graphviz es un conjunto de herramientas de software para el diseño de diagramas definido en el lenguaje descriptivo DOT. ​ Fue desarrollado por AT&T Labs​ y liberado como software libre con licencie tipo Eclipse.​

### Interprete

En ciencias de la computación, intérprete o interpretador es un programa informático capaz de analizar y ejecutar otros programas. Los intérpretes se diferencian de los compiladores o de los ensambladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción.

### Patrón interprete

Es un patrón de diseño que, dado un lenguaje, define una representación para su gramática junto con un intérprete del lenguaje. Se usa para definir un lenguaje para representar expresiones regulares que representen cadenas a buscar dentro de otras cadenas.

Fuente del glosario de términos: <https://es.wikipedia.org/>.

## Archivos de código fuente utilizados

|  |  |
| --- | --- |
| Nombre | Descripción |
| AMBITO.PY | Contiene a la clase “Ámbito” y es utilizada en el intérprete de Augus. |
| ASCENDENTE.PY | Contiene el analizador léxico y sintáctico de MinorC, también se genera el árbol sintáctico. |
| ASCENDENTE\_AUGUS.PY | Contiene el analizador léxico y sintáctico de Augus, también se genera el árbol sintáctico. |
| CLASES.PY | Contiene todas las clases utilizadas en la construcción del árbol sintáctico, sirviendo como modelo de los objetos que lo conforman. |
| EXPRESIONES.PY | Contiene todas las clases que representan expresiones y que son utilizadas para construir el árbol sintáctico de Augus. |
| INSTRUCCIONES.PY | Contienen todas las clases que representan una instrucción y que son utilizadas para construir el árbol sintáctico de Augus. |
| INTERPRETE.PY | Ejecuta el árbol sintáctico generado por los analizadores de Augus. |
| PARSERTAB.PY | Archivo generado por PLY. |
| PRINCIPAL.PY | Contiene toda la interfaz gráfica del programa, además que ejecuta las principales funciones solicitadas por el usuario final. |
| RECURSOS.PY | Contiene recursos gráficos utilizados en la interfaz gráfica. |
| TABLASIMBOLOS.PY | Contiene la tabla de símbolos utilizada por el intérprete del Augus. |
| TRES\_DIRECCIONES.PY | Contiene todas las funciones necesarias para la generación de código de tres direcciones, optimizaciones y estructuras con contenido para la generación de reportes. |
| VALORES.PY | Contiene todas las clases utilizadas en la creación del árbol sintáctico de Augus. |

## Clases utilizadas en la creación del AST

|  |  |
| --- | --- |
| Clase | Descripción |
| Declaracion | Clase que representa de manera general a una declaración. |
| DeclaracionFinal | Clase que representa específica a una declaración. |
| Estructura | Clase que representa a un “struct”. |
| Función | Clase que representa a una función. |
| Parametro | Clase que representa a un parámetro en una función. |
| Metódo | Clase que representa la llamada a un método. |
| Etiqueta | Clase que representa a una etiqueta. |
| Salto | Clase que representa a un salto. |
| AsignacionNormal | Clase que representa a una asignación estándar. |
| AsignacionEstructura | Clase que representa a una asignación en una estructura. |
| AsignacionAumento | Clase que representa una asignación con aumento. |
| AsignacionDecremento | Clase que representa una asignación con decremento. |
| \_If | Clase que representa a la sentencia de control “IF”. |
| \_ElseIf | Clase que representa a un “else if” de un “IF”. |
| \_Else | Clase que representa al “else” de un “IF”. |
| \_Switch | Clase que representa a la sentencia de control “Switch”. |
| \_Case | Clase que representa a un “case” de un “Switch”. |
| \_DefaultCase | Clase que representa a un “default” de un “Switch”. |
| \_While | Clase que representa a una sentencia de control “While”. |
| \_Do | Clase que representa a una sentencia de control “Do”. |
| \_For | Clase que representa a una sentencia de control “For”. |
| Expresión | Clase que representa a una expresión aritmética, relacional, lógica, binaria, etc. |
| \_SizeOf | Clase que representa a la función “sizeof()”. |
| Valor | Clase que representa a un entero, carácter, decimal o una cadena. |
| \_Continue | Clase que representa a la función “continue”. |
| \_Break | Clase que representa a la función “break”. |
| \_Return | Clase que representa a la función “return”. |

## Generalidades

### Construcción del árbol sintáctico abstracto

El árbol sintáctico abstracto fue construido utilizando clases que representaran de manera especifica a cada hoja, de esta forma se garantiza la realización sencilla de acciones complejas con la manipulación de hojas, esto se logra sintetizando los objetos y con la ayuda de listas.

### Uso del patrón de interprete

El patrón interprete es usado en el momento de generar código de tres direcciones, ya que se recorre cada elemento del árbol en preorden y dependiendo de qué tipo de instancia sea la hoja, se realiza la generación de código de tres direcciones, para el manejo de ámbitos se utilizaron tres clases: Símbolo, TablaSimbolos y Ámbito, con estas clases se manejo el correcto uso de variables y su asociación con los temporales para la fácil generación de código.

### Reporte gramatical

El reporte gramatical es generado cuando se realiza el análisis ascendente aprovechando que se realiza esta tarea para ir concatenando cada elemento de la tabla que conforma el reporte gramatical, esto mejora el resultado final de rendimiento de la aplicación.

### Reporte de AST

Para poder generar el reporte del árbol sintáctico abstracto, se recorre todo el árbol y aprovechando de que cada hoja es una instancia de una clase, se comprueba y se concatena el código necesario utilizado por graphviz, al final este se muestra automáticamente haciendo uso de la biblioteca única para Python.

### Optimización de código

Para la optimización de código se crearon funciones especificas para cada grupo de reglas, gracias al uso de expresiones regulares se pueden comprobar cada uno de los elementos para hacer validaciones por cada regla y se optimiza el código de manera incremental.

## Principales funciones utilizadas en la generación de Código 3D

|  |  |
| --- | --- |
| Método | Descripción |
| generar\_codigo | Método de inicio de generación de código. |
| obtener\_funciones | Carga las funciones en memoria. |
| mostrar\_mensaje\_consola | Muestra un mensaje en la terminal del editor. |
| existe\_main | Comprueba que exista la función main en las funciones. |
| cargar\_variables\_globales | Carga las variables globales en memoria. |
| obtener\_expresion | Genera el código de una expresión y devuelve el temporal de esta. |
| obtener\_expresion\_aritmetica | Genera el código de una expresión aritmética. |
| obtener\_expresion\_relacional | Genera el código de una expresión relacional. |
| obtener\_expresion\_logica | Genera el código de una expresión lógica. |
| obtener\_expresion\_bit | Genera el código de una expresión bit. |
| obtener\_expresion\_unaria | Genera el código de una expresión unaria. |
| obtener\_expresion\_ternaria | Genera el código de una expresión ternaria. |
| obtener\_expresion\_casteo | Genera el código de una expresión de casteo. |
| obtener\_expresion\_identificador\_arreglo | Genera el código de un acceso a un arreglo. |
| obtener\_expresion\_estructura | Genera el código de un acceso a una estructura. |
| obtener\_expresion\_aumento\_decremento | Genera el código de un acceso a una variable con aumento o decremento. |
| obtener\_expresion\_sizeof | Genera el código de un sizeof. |
| obtener\_registro\_temporal | Genera un registro temporal. |
| obtener\_etiqueta\_temporal | Genera una etiqueta temporal. |
| generar\_codigo\_main | Genera el código del método principal. |
| generar\_codigo\_instrucciones | Genera el código de instrucciones. |
| generar\_codigo\_etiqueta | Genera el código de una etiqueta. |
| generar\_codigo\_salto | Genera el código de un salto. |
| generar\_codigo\_declarcion | Genera el código de una declaración. |
| generar\_codigo\_asignacion | Genera el código de una asignación. |
| generar\_codigo\_if | Genera el código de un IF. |
| generar\_codigo\_switch | Genera el código de un SWITCH. |
| generar\_codigo\_break | Genera el código de un BREAK. |
| generar\_codigo\_while | Genera el código de un WHILE. |
| generar\_codigo\_do | Genera el código de un DO. |
| generar\_codigo\_for | Genera el código de un FOR. |
| generar\_codigo\_metodo | Genera el código de una llamada a un método. |
| generar\_codigo\_printf | Genera el código de un PRINTF. |
| optimizar\_codigo | Optimiza el código generado. |

## Gramática Ascendente

|  |  |
| --- | --- |
| PRODUCCION | REGLA |
| INIT -> CUERPO\_GLOBAL | INIT.VAL = CUERPO\_GLOBAL.VAL; |
| CUERPO\_GLOBAL -> LISTA\_GLOBAL | CUERPO\_GLOBAL.VAL = LISTA\_GLOBAL.VAL; |
| CUERPO\_GLOBAL -> | CUERPO\_GLOBAL.VAL = NONE; |
| LISTA\_GLOBAL -> LISTA\_GLOBAL1 INSTRUCCION\_GLOBAL | LISTA\_GLOBAL1.ADD(INSTRUCCION\_GLOBAL); LISTA\_GLOBAL.VAL = LISTA\_GLOBAL1.VAL; |
| LISTA\_GLOBAL -> INSTRUCCION\_GLOBAL | LISTA\_GLOBAL.VAL = Lista(INSTRUCCION\_GLOBAL.VAL); |
| INSTRUCCION\_GLOBAL -> DECLARACION; | ESTRUCTURA ; | FUNCION | INSTRUCCION\_GLOBAL.VAL = DECLARACION.VAL | ESTRUCTURA.VAL | FUNCION.VAL; |
| ESTRUCTURA -> struct identificador { CARACTERISTICAS } | ESTRUCTURA.VAL = Estructura(identificador, CARACTERISTICAS.VAL, lineno); |
| CARACTERISTICAS -> LISTA\_CARACTERISTICAS | CARACTERTISCIAS.VAL = LISTA\_CARACTERISTICAS.VAL; |
| CARACTERISTICAS -> | CARACTERTISICAS.VAL = NONE; |
| LISTA\_CARACTERISTICAS -> LISTA\_CARACTERISTICAS CARACTERISTICA | LISTA\_CARACTERISTICAS1.APPEND(CARACTERISTICA.VAL); LISTA\_CARACTERISTICAS.VAL = LISTA\_CARACTERISTICAS1.VAL; |
| LISTA\_CARACTERITISCAS -> CARACTERISTICA | LISTA\_CARACTERISTICAS.VAL = Lista(CARACTERISTICA.VAL); |
| CARACTERISTICA -> DECLARACION; | CARACTERISTICA.VAL = DECLARACION.VAL; |
| FUNCION -> TIPO identificador ( PARAMETROS ) { CUERPO\_LOCAL } | FUNCION.VAL = Funcion(TIPO.VAL, identificador, PARAMETROS.VAL, CUERPO.VAL); |
| PARAMETROS -> LISTA\_PARAMETROS | PARAMETROS.VAL = LISTA\_PARAMETROS.VAL; |
| PARAMETROS -> | PARAMETROS.VAL = NONE; |
| LISTA\_PARAMETROS -> LISTA\_PARAMETROS1 , PARAMETRO | LISTA\_PARAMETROS1.ADD(PARAMETRO.VAL); LISTA\_PARAMETROS.VAL = LISTA\_PARAMETROS1.VAL; |
| LISTA\_PARAMETROS -> PARAMETRO | LISTA\_PARAMETROS.VAL = Lista(PARAMETRO.VAL); |
| PARAMETRO -> TIPO identificador | PARAMETRO.VAL = Parametro(TIPO.VAL, false, identificador); |
| PARAMETRO -> TIPO & identificador | PARAMETRO.VAL = Parametro(TIPO.VAL, true, identificador); |
| CUERPO\_LOCAL -> LISTA\_LOCAL | CUERPO\_LOCAL.VAL = LISTA\_LOCAL.VAL; |
| CUERPO\_LOCAL -> | CUERPO\_LOCAL.VAL = NONE; |
| LISTA\_LOCAL -> LISTA\_LOCAL1 INSTRUCCION\_LOCAL | LISTA\_LOCAL1.ADD(INSTRUCCION\_LOCAL.VAL); LISTA\_LOCAL.VAL = LISTA\_LOCAL1.VAL; |
| LISTA\_LOCAL -> INSTRUCCION\_LOCAL | LISTA\_LOCAL.VAL = Lista(INSTRUCCION\_LOCAL.VAL); |
| INSTRUCCION\_LOCAL -> ETIQUETA | SALTO | DECLARACION ; | ASIGNACION ; | IF | SWITCH | WHILE | DO | FOR | PRINT ; | METODO ; | INSTRUCCION\_LOCAL.VAL = ETIQUETA.VAL | SALTO.VAL | DECLARACION.VAL | ASIGNACION.VAL | IF.VAL | SWITCH.VAL | WHILE.VAL | DO.VAL | FOR.VAL | PRINT.VAL | METODO.VAL; |
| PRINT -> printf ( LISTA\_EXPRESIONES ) | PRINT.VAL = PrintF(LISTA\_EXPRESIONES.VAL, lineno); |
| INSTRUCCION\_LOCAL -> continue; | INSTRUCCION\_LOCAL.VAL = Continue(lineno); |
| INSTRUCCION\_LOCAL -> break; | INSTRUCCION\_LOCAL.VAL = Break(lineno); |
| INSTRUCCION\_LOCAL -> return EXPRESION; | INSTRUCCION\_LOCAL.VAL = Return(EXPRESION.val, lineno); |
| INSTRUCCION\_LOCAL -> return; | INSTRUCCION\_LOCAL.VAL = Return(NONE, lineno); |
| METODO -> identificador ( EXPRESIONES ) | METODO.VAL = Metodo(identidicador, EXPRESIONES.VAL, lineno); |
| ETIQUETA -> identificador: | ETIQUETA.VAL = Etiqueta(identificador, lineno); |
| SALTO -> goto identificador; | SALTO.VAL = Salto(identificador, lineno); |
| DECLARACION -> TIPO LISTA\_DECLARACION | DECLARACION.VAL = Declaracion(TIPO.VAL, LISTA\_DECLARACION.VAL); |
| LISTA\_DECLARACION -> LISTA\_DECLARACION1 , DECLARACION\_FINAL | LISTA\_DECLARACION1.ADD(DECLARACION\_FINAL.VAL); LISTA\_DECLARACION.VAL = LISTA\_DECLARACION1.VAL; |
| LISTA\_DECLARACION -> DECLARACION\_FINAL | LISTA\_DECLARACION.VAL = Lista(DECLARACION\_FINAL.VAL); |
| DECLARACION\_FINAL -> identificador INDICES | DECLARACION\_FINAL.VAL = DeclaracionFinal(identificador, INDICES.VAL, NONE, lineno); |
| DECLARACION\_FINAL -> identificador INDICES = EXPRESION | DECLARACION\_FINAL.VAL = DeclaracionFinal(identificador, INDICES.VAL, EXPRESIONES.VAL, lineno); |
| INDICES -> ACCESOS | INDICES.VAL = ACCESOS.VAL |
| INDICES -> | INDICES.VAL = NONE; |
| ACCESOS -> ACCESOS1 ACCESO | ACCESOS1.ADD(ACCESO.VAL); ACCESOS.VAL = ACCESOS1.VAL; |
| ACCESOS -> ACCESO | ACCESOS.VAL = Lista(ACCESO.VAL); |
| ACCESO -> [ EXPRESION ] | ACCESO.VAL = EXPRESION.VAL; |
| ACCESO -> [ ] | ACCESO.VAL = Lista(); |
| ASIGNACION -> identificador INDICES COMPUESTO EXPRESION | ASIGNACION.VAL = AsignacionNormal(identificador, INDICES.VAL, COMPUESTO.VAL, EXPRESION.VAL, lineno); |
| ASIGNACION -> identificador INDICES . identificador INDICES COMPUESTO EXPRESION | ASIGNACION.VAL = AsignacionEstructura(identificador, INDICES.VAL, identificador, INDICES.val, COMPUESTO.VAL, EXPRESION.VAL, lineno); |
| ASIGNACION -> identificador ++ | ASIGNACION.VAL = AsignacionAumento(identificador, lineno); |
| ASIGNACION -> ++ identificador | ASIGNACION.VAL = AsignacionAumento(identificador, lineno); |
| ASIGNACION -> identificador -- | ASIGNACION.VAL = AsignacionDecremento(identificador, lineno); |
| ASIGNACION -> -- identificador | ASIGNACION.VAL = AsignacionDecremento(identificador, lineno); |
| COMPUESTO -> = | += | -= | \*= | /= | %= | <<= | >>= | & | |= | ^= | COMPUESTO.VAL = = | += | -= | \*= | /= | %= | <<= | >>= | & | |= | ^=; |
| IF -> if ( EXPRESION ) { CUERPO\_LOCAL } | IF.VAL = If(EXPRESION.VAL, CUERPO\_LOCAL.VAL, NONE, NONE, lineno); |
| IF -> if ( EXPRESION ) { CUERPO\_LOCAL } ELSE | IF.VAL = If(EXPRESION.VAL, CUERPO\_LOCAL.VAL, NONE, ELSE.VAL, lineno); |
| IF -> if ( EXPRESION ) { CUERPO\_LOCAL } ELSEIF IF\_FINAL | IF.VAL = If(EXPRESION.VAL, CUERPO\_LOCAL.VAL, ELSEIF.VAL, IF\_FINAL.VAL, lineno); |
| IF\_FINAL -> ELSEIF IF\_FINAL | IF\_FINAL.VAL = Lista(ELSEIF) + IF\_FINAL.VAL; |
| IF\_FINAL -> ELSE | IF\_FINAL.VAL = Lista(ELSE.VAL); |
| IF\_FINAL -> | IF\_FINAL.VAL = NONE; |
| ELSEIF -> else if ( EXPRESION ) { CUERPO\_LOCAL } | ELSEIF.VAL -> ElseIf(EXPRESION.VAL, CUERPO\_LOCAL.VAL); |
| ELSE -> else { CUERPO\_LOCAL } | ELSE.VAL = Else(CUERPO\_LOCAL.VAL); |
| SWITCH -> switch ( EXPRESION ) { CASES DEFAULT\_CASE } | SWITCH.VAL = Switch(EXPRESION.VAL, CASES.VAL, DEFAULT\_CASE.VAL, lineno); |
| CASES -> LISTA\_CASE | CASES.VAL = LISTA\_CASE.VAL |
| CASES -> | CASES.VAL = NONE; |
| LISTA\_CASE = LISTA\_CASE1 CASE | LISTA\_CASE1.ADD(CASE.VAL); LISTA\_CASE.VAL = LISTA\_CASE1.VAL; |
| LISTA\_CASE -> CASE | LISTA\_CASE.VAL = Lista(CASE.VAL); |
| CASE -> case EXPRESION : CUERPO\_LOCAL | CASE.VAL = Case(EXPRESION.VAL, CUERPO\_LOCAL.VAL); |
| DEFAULT\_CASE -> default : CUERPO\_LOCAL | DEFAULT\_CASE.VAL = DefaultCase(CUERPO\_LOCAL.VAL); |
| WHILE -> while ( EXPRESION ) { CUERPO\_LOCAL } | WHILE.VAL = While(EXPRESION.VAL, CUERPO\_LOCAL.val, lineno); |
| DO -> do { CUERPO\_LOCAL } while ( EXPRESION ) ; | DO.VAL = Do(CUERPO\_LOCAL.VAL, EXPRESION.VAL, lineno); |
| FOR -> for ( INICIO\_FOR ; EXPRESION ; ASIGNACION ) { CUERPO\_LOCAL } | FOR.VAL = For(INICIO\_FOR.VAL, EXPRESION.VAL; ASIGNACION.VAL, CUERPO\_LOCAL.VAL, lineno); |
| INICIO\_FOR -> DECLARACION | ASIGNACION | INICIO\_FOR.VAL = DECLARACION.VAL | ASIGNACION.VAL; |
| EXPRESION -> EXPRESION1 [+|-|\*|/|%] EXPRESION2 | EXPRESION.VAL -> ExpresionAritmetica(EXPRESION1.VAL, [+|-|\*|/|%], EXPRESION2.VAL); |
| EXPRESION -> EXPRESION1 [==|!=|>|<|>=|<=] EXPRESION2 | EXPRESION.VAL = ExpresionRelacional(EXPRESION1.VAL, [==|!=|>|<|>=|<=], EXPRESION2.VAL); |
| EXPRESION -> EXPRESION1 [and|or] EXPRESION2 | EXPRESION.VAL = ExpresionLogica(EXPRESION1.VAL, [and|or], EXPRESION2.VAL); |
| EXPRESION -> EXPRESION1 [<<|>>|&|"|"|^] EXPRESION2 | EXPRESION.VAL = ExpresionBit(EXPRESION1, [<<|>>|&|"|"|^], EXPRESION2); |
| EXPRESION -> EXPRESION1 ? EXPRESION2 : EXPRESION3 | EXPRESION.VAL = ExpresionTernaria(EXPRESION1, EXPRESION2, EXPRESION3); |
| EXPRESION -> [-|!|~] EXPRESION1 | EXPRESION.VAL = ExpresionUnaria([-|!|~], EXPRESION1); |
| EXPRESION -> & identificador | EXPRESION.VAL = ExpresionReferencia(identificador); |
| EXPRESION -> METODO | EXPRESION.VAL = METODO.VAL; |
| EXPRESION -> ( EXPRESION1 ) | EXPRESION.VAL = EXPRESION1.VAL; |
| EXPRESION -> identificador INDICES . identificador INDICES | EXPRESION.VAL = ExpresionEstructura(identificador, INDICES.VAL, identificador, INDICES.VAL); |
| EXPRESION -> identificador ACCESOS | EXPRESION.VAL = ExpresionIdentificadorArreglo(identificador, ACCESOS); |
| EXPRESION -> { EXPRESIONES } | EXPRESION.VAL = ExpresionElementos(EXPRESIONES.VAL); |
| EXPRESION -> sizeof ( TIPO ) | EXPRESION.VAL = SizeOf(TIPO.VAL); |
| EXPRESION -> identificador ++ | EXPRESION.VAL = ExpresionAumentoDecremento(identificador, ++, post); |
| EXPRESION -> ++ identificador | EXPRESION.VAL = ExpresionAumentoDecremento(identificador, ++, pre); |
| EXPRESION -> identificador -- | EXPRESION.VAL = ExpresionAumentoDecremento(identificador, --, post); |
| EXPRESION -> -- identificador | EXPRESION.VAL = ExpresionAumentoDecremento(identificador, --, pre); |
| EXPRESION -> scanf ( ) | EXPRESION.VAL = ExpresionScan(); |
| EXPRESION -> ( TIPO ) EXPRESION | EXPRESION.VAL = ExpresionCasteo(TIPO.VAL, EXPRESION.VAL); |
| EXPRESION -> carácter | EXPRESION.VAL = caracter; |
| EXPRESION -> cadena | EXPRESION.VAL = cadena; |
| EXPRESION -> entero | EXPRESION.VAL = entero; |
| EXPRESION -> decimal | EXPRESION.VAL = decimal; |
| EXPRESION -> identificador | EXPRESION.VAL = identificador; |
| EXPRESIONES -> LISTA\_EXPRESIONES | EXPRESIONES.VAL = LISTA\_EXPRESIONES.VAL |
| EXPRESIONES -> | EXPRESIONES.VAL = NONE; |
| LISTA\_EXPRESIONES -> LISTA\_EXPRESIONES1 , EXPRESION | LISTA\_EXPRESIONES1.ADD(EXPRESION.VAL); LISTA\_EXPRESIONES.VAL = LISTA\_EXPRESIONES1.VAL; |
| LISTA\_EXPRESIONES -> EXPRESION | LISTA\_EXPRESIONES.VAL = Lista(EXPRESION.VAL); |
| TIPO -> int | char | double | float | void | TIPO.VAL = Tipo([int|char|double|float|void], None); |
| TIPO -> struct identificador | TIPO.VAL = Tipo(struct, identificador); |