Trabajo Integrador

Práctica y Construcción de Compiladores

INTEGRANTES

CABRERA, AUGUSTO GABRIEL

VILLAR , FEDERICO IGNACIO

FCEFyN

AÑO 2022

- Las presentes filminas son a modo explicativo sobre el desarrollo del trabajo final de la materia antes mencionada.
- Estas presentaciones no buscan justificar conocimientos teóricos, solamente señalar conceptos mas prácticos sobre el trabajo realizado.

- Recomendación
  - Correr los archivos adjuntos en la IDE, Visual Studio Code.
    - Tener instalado el siguiente Plugin:

Better Comments v3.0.2

Aaron Bond | 4.007.673 | ★★★★★ (153) | Patrocinador

Improve your code commenting by annotating with alert, informational, TODOs, and more!

ANTLR4 grammar syntax support v2.3.1

Mike Lischke | 95.937 | ★★★★½ (32)

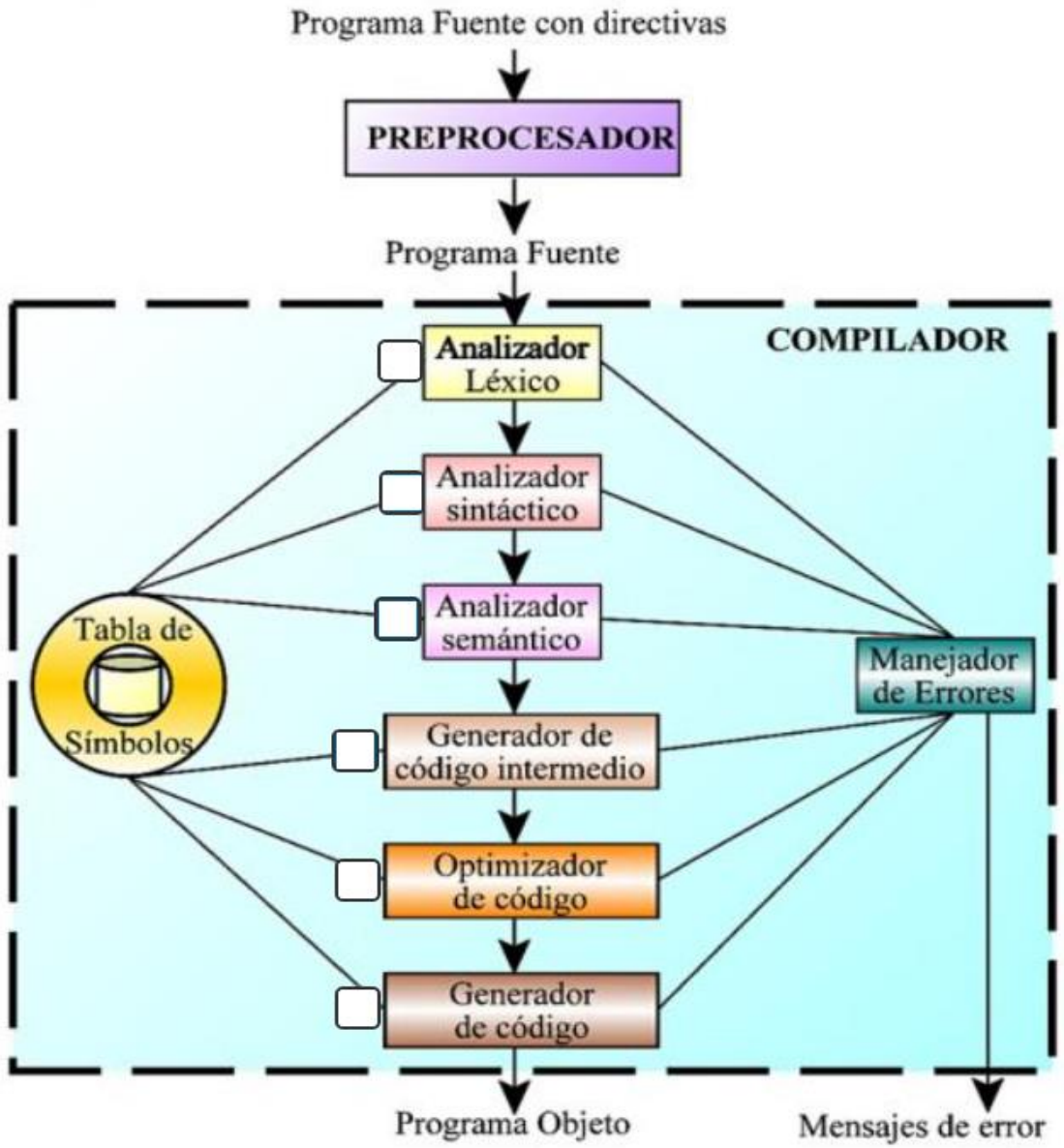
Language support for ANTLR4 grammar files

Extension Pack for Java v0.25.7 *Vista Previa*

Microsoft | 16.428.577 | ★★★★☆ (54)

Popular extensions for Java development that provides Java IntelliSense, debugging, testing, Maven/Gradle support, project mana...

Etapas del Compilador



Analizador Léxico

Consistente en un programa que recibe como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de *tokens* (componentes léxicos) o símbolos.
Estos *tokens* sirven para una posterior etapa del proceso de traducción, siendo la entrada para el analizador sintáctico (en inglés *parser*).
La especificación de un lenguaje de programación a menudo incluye un conjunto de reglas que definen el léxico. Estas reglas consisten comúnmente en expresiones regulares que indican el conjunto de posibles secuencias de caracteres que definen un *token* o lexema.

Analizador Léxico

```
fragment LETRA: [A-Za-z];   /* La palabra reservada "FRAGMENT" significa que no quiero
fragment DIGITO:[0-9];         que la expresión regular DIGITO sea un  TOKEN sino que
                               solo quiero que el conjunto de los DIGITOS sea un token
                               (ósea para formar tokens  con su concatenación)
```

Un token es texto del programa de código fuente que el compilador no divide en elementos de componente

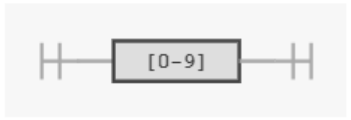
```
PA: '(';
PC: ')';
LLA: '{';
LLC: '}';
PYC: ';';
SUMA: '+';
MULT: '*';
DIVI: '/';
RESTA: '-';

MAIN: 'main';
IF: 'if';
INT: 'int';
STRING: 'string';
BOOLEAN: 'bool';
CHAR: 'char';
FLOAT: 'float';
DOUBLE: 'double';
FALSE: 'false';
TRUE: 'true';
```

```
ID: (LETRA | '_') (LETRA | DIGITO | '_')*; //! ID = IDentificador
NUMERO: DIGITO+;


TEXTO: '"' (LETRA | '_') (LETRA | DIGITO | '_')* '"';

CARACTER: '\'' (LETRA) '\'';

WS:
[ \n\t\r] -> skip; //! si encuentra alguno de esos caracteres, hace salto "skip"

OTRO: .;
```

"Show Railroad Diagrama for Rule"

-> Para ver las reglas tanto léxicas como gramaticales gráficamente

DIGITO

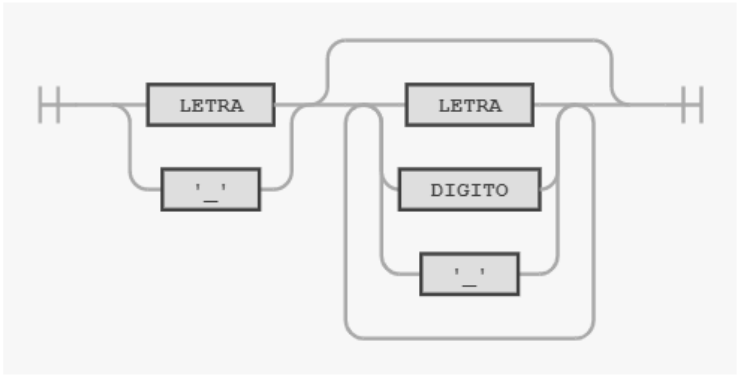


PA

PC

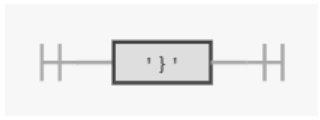
PYC

ID



LLA

LLC



PYC

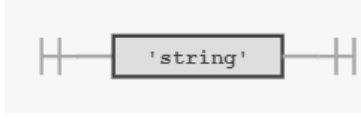
SUMA

MULT

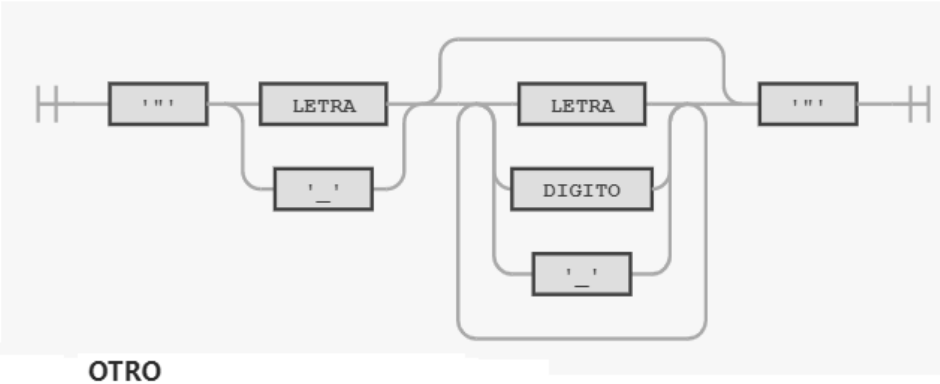
RESTA

IF

STRING

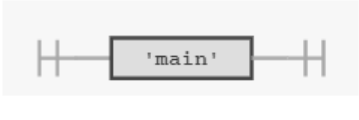


TEXTO

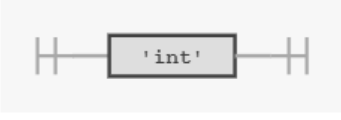


DIVI

MAIN



INT

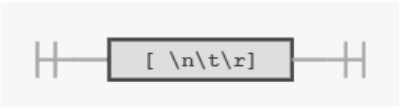


BOOLEAN

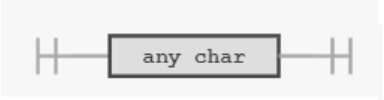
NUMERO

CARACTER

WS



OTRO

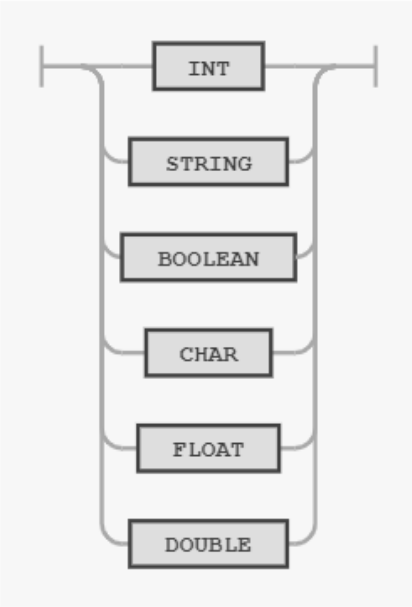


Analizador Sintáctico (Parser)

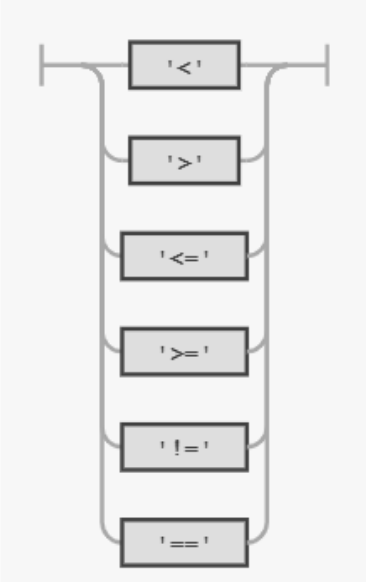
Convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Un analizador crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos
por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

Reglas Gramaticales

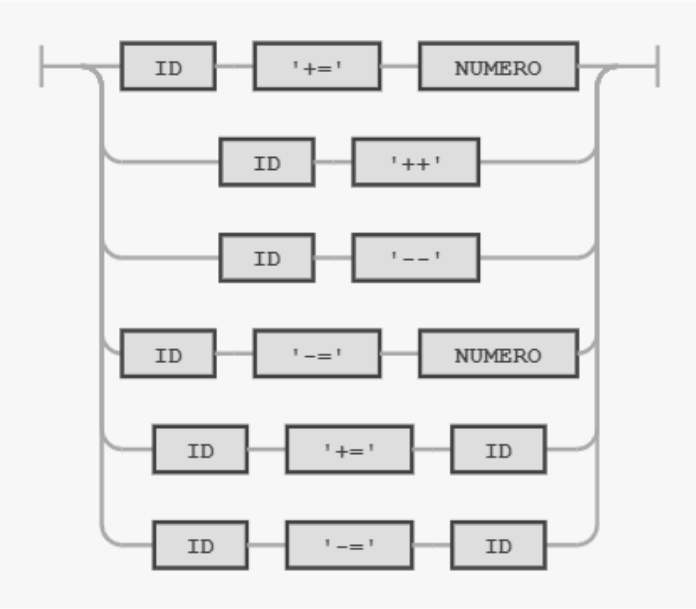
tipo



logica

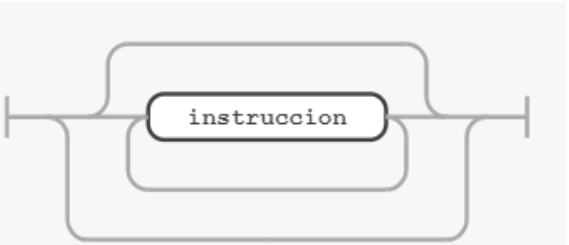


incremento



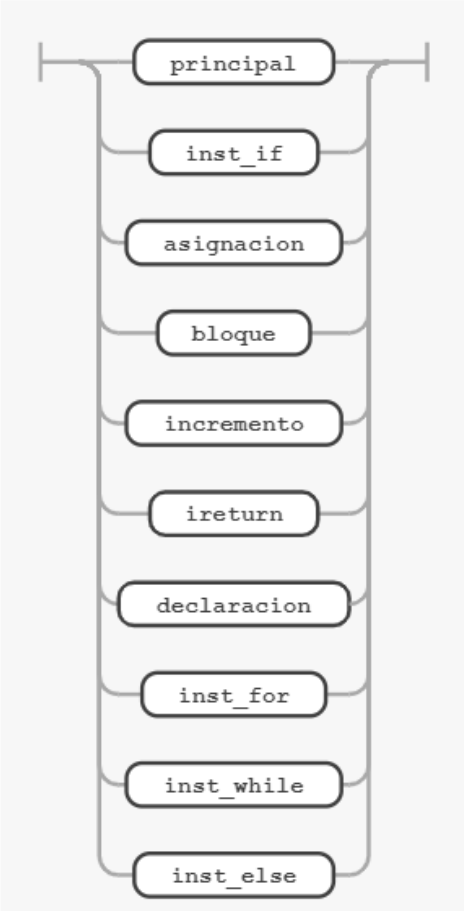
programa

instrucciones



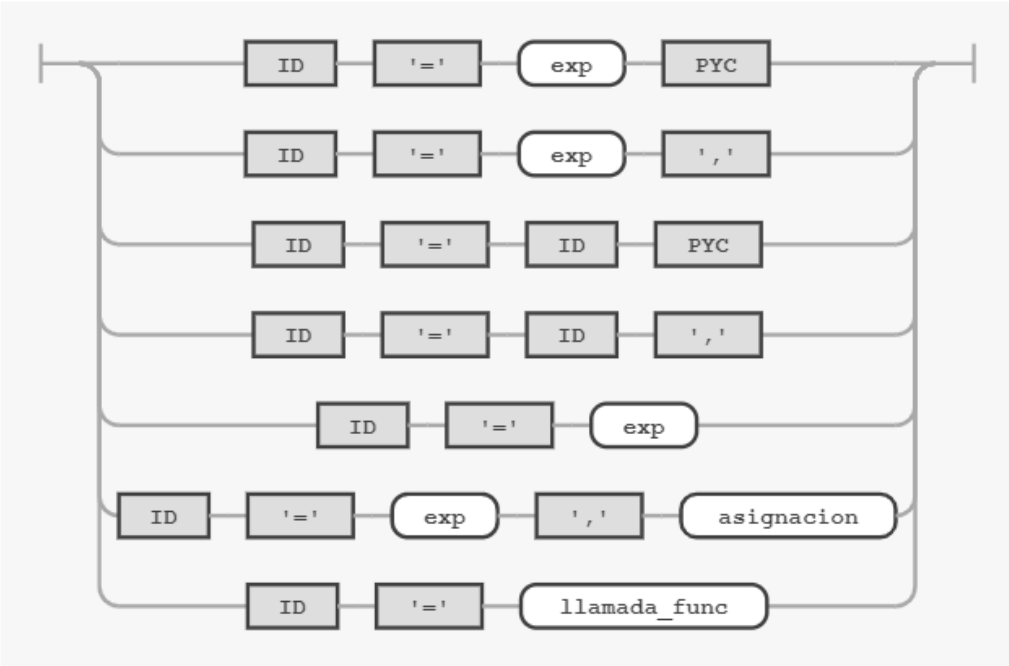
Analizador Sintáctico (Parser)

instruccion

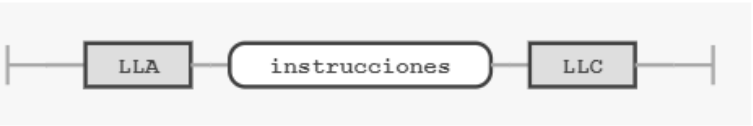


principal

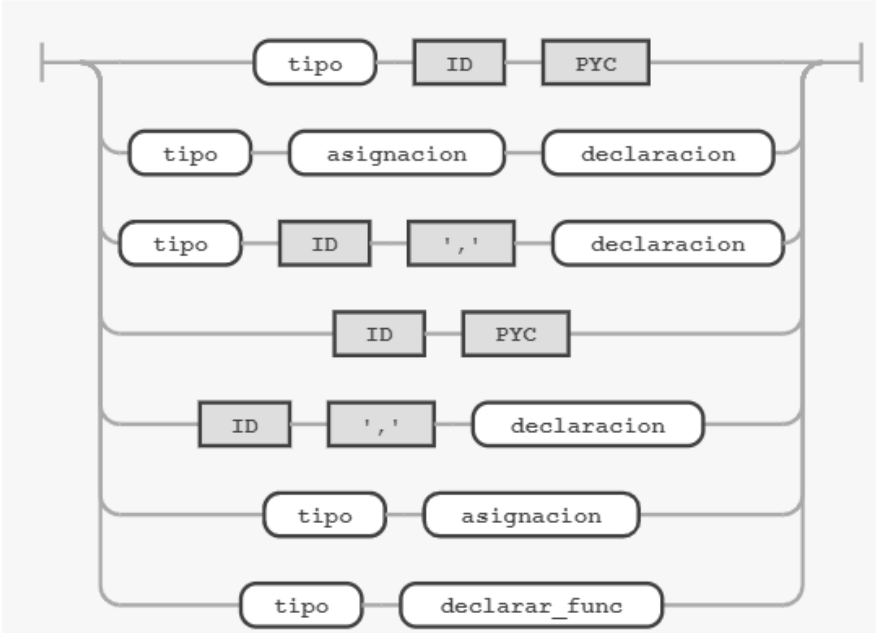
asignacion



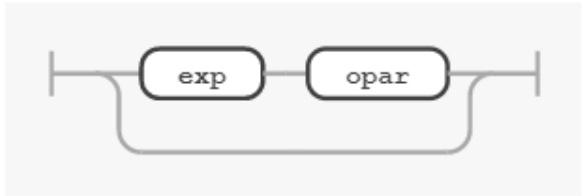
bloque



declaracion



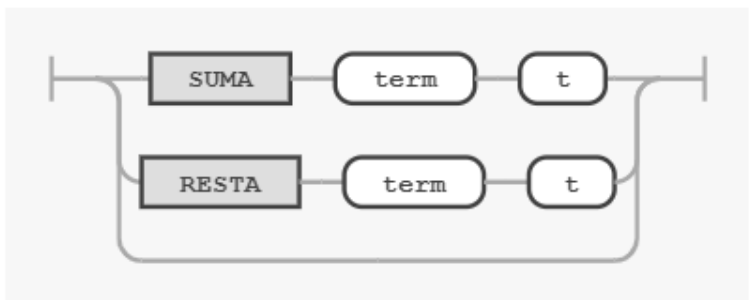
opar



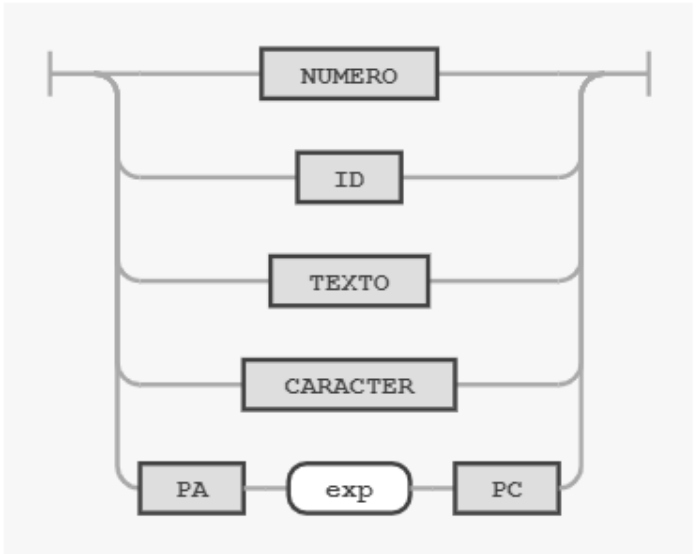
exp

term

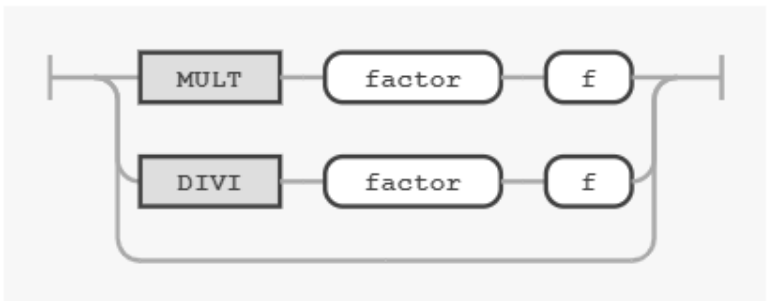
t



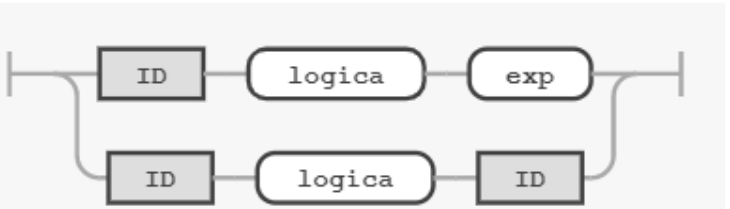
factor



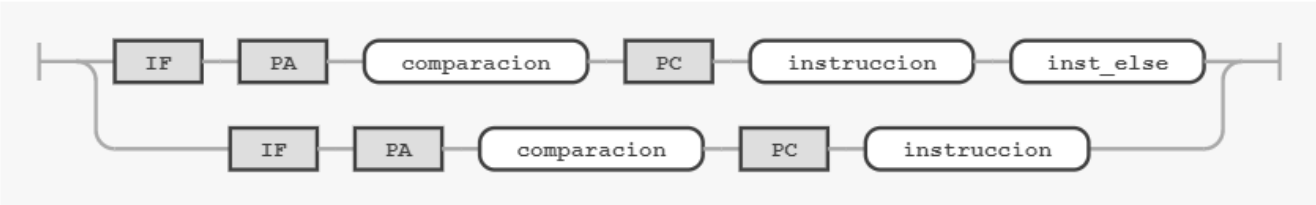
f



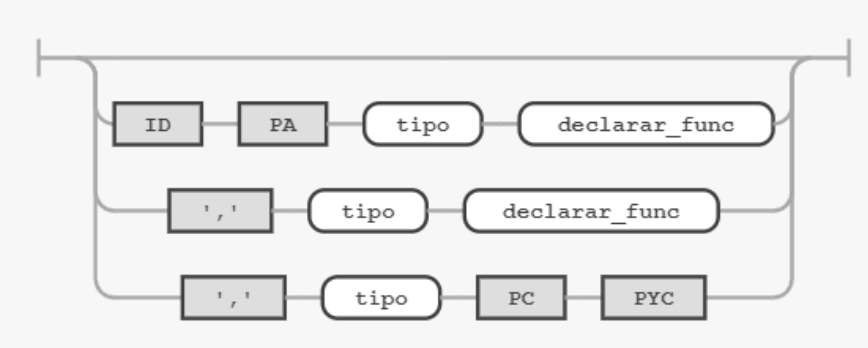
comparacion



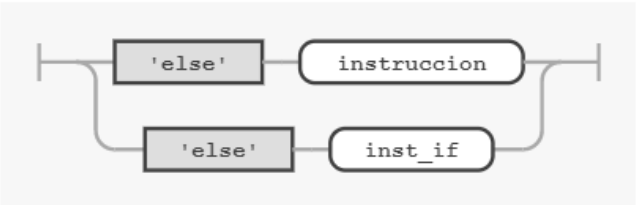
inst_if



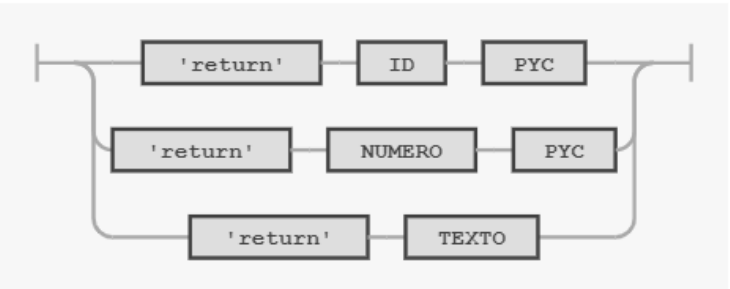
declarar_func



inst_else

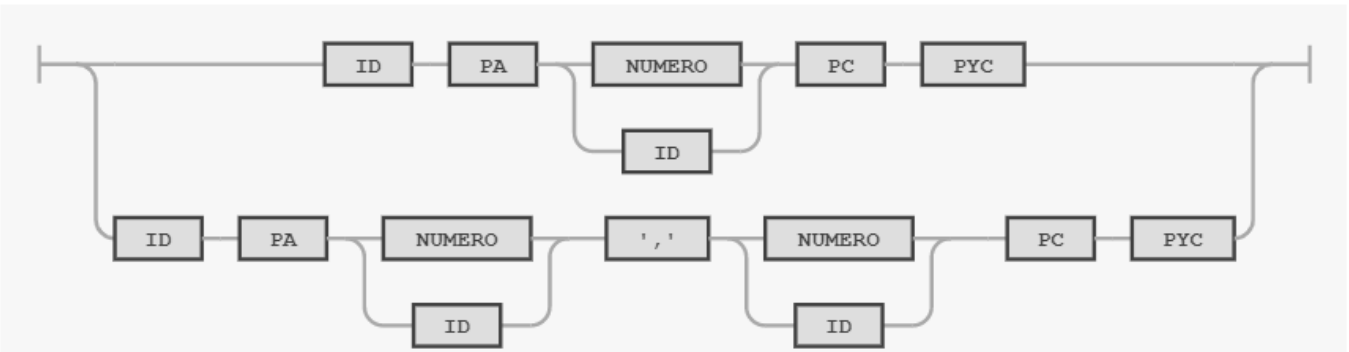


ireturn



inst_while

llamada_func



Parser

```
1  int main() {
2    int i=3;
3    int c = i-2*9;
4  }
```

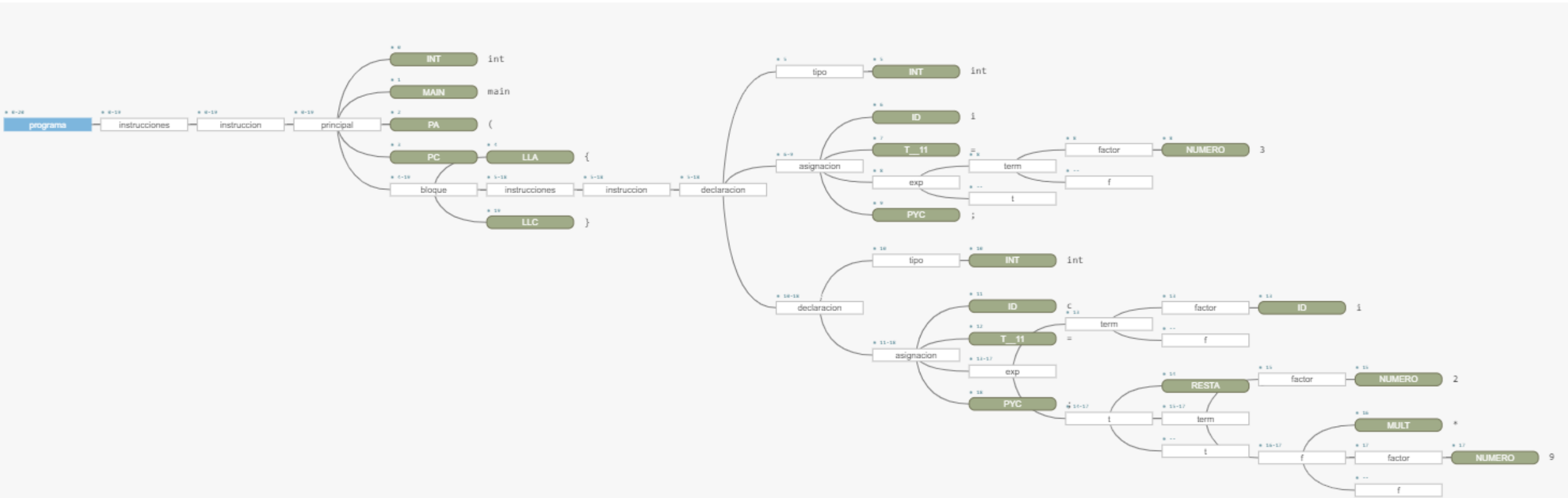
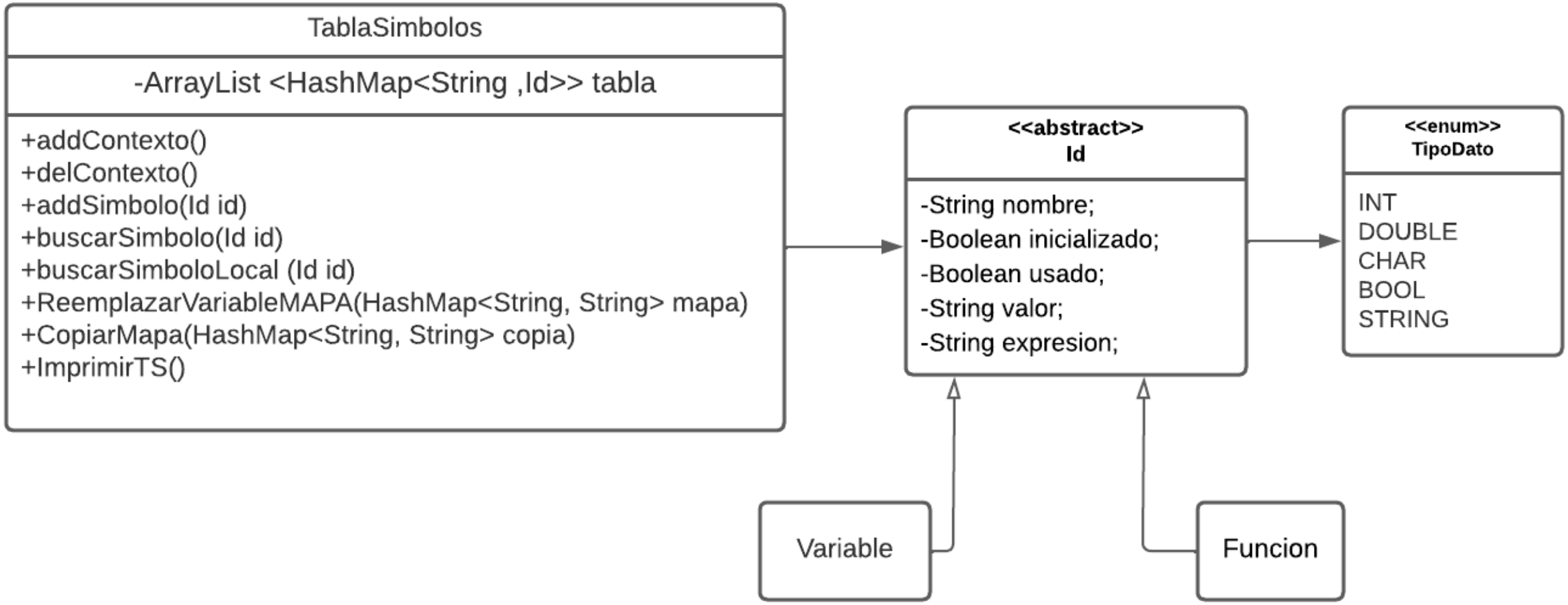


Diagrama UML de TablaSimbolos



Ejemplo:

Se presenta la siguiente entrada:

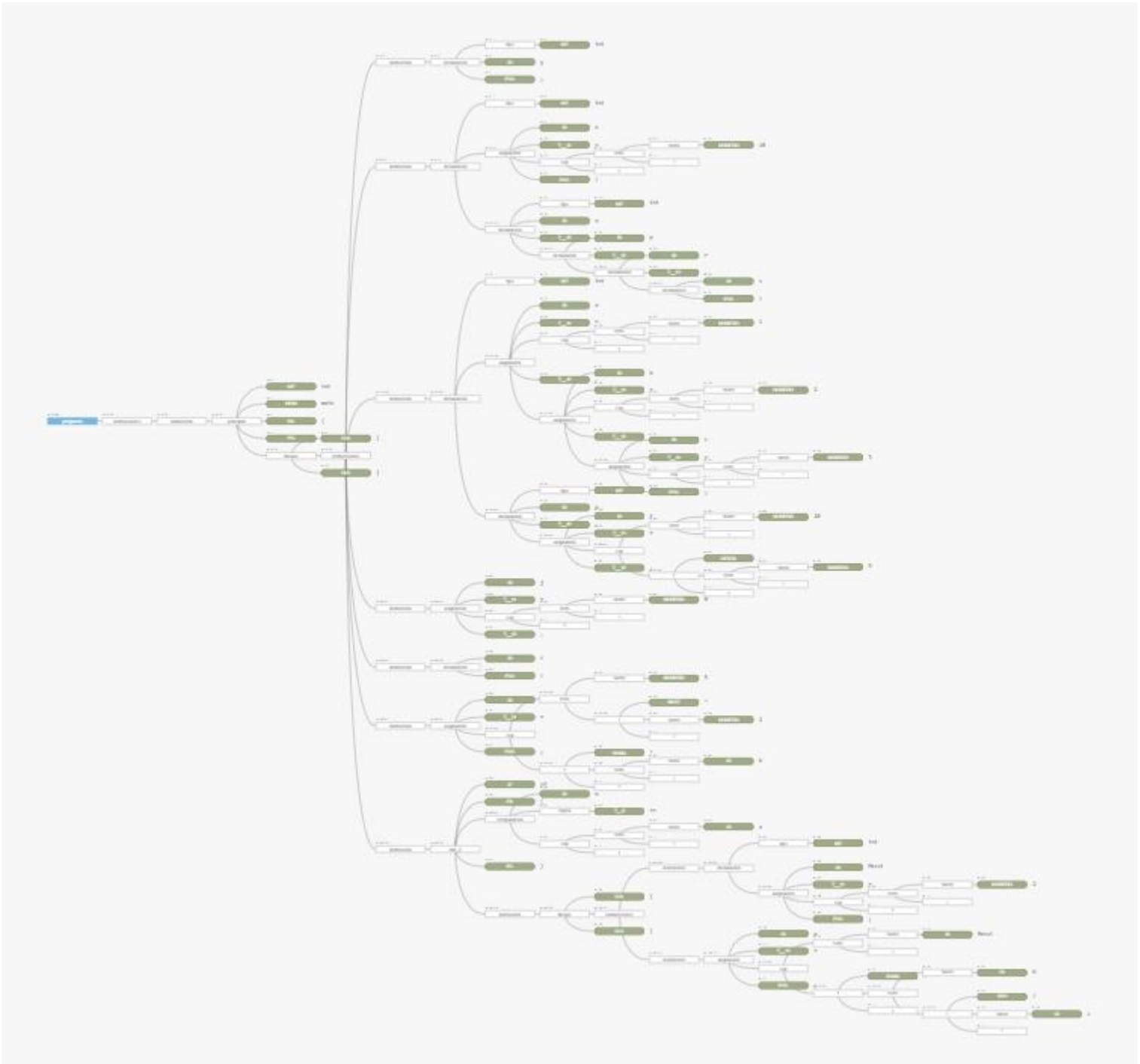
"miPrograma.c"

```c
int main() {
    int y;
    int x = 10;
    int o, p, r, s;
    int a = 1, b = 2, c = 3;
    int w, t = 10-9, j = 0, i;
     i = 5 * 2 + b;

     if(w==a)
     {
        int Messi= 2;
        w=  Messi +b/c;

     }
}
```

Generando la Tabla de Símbolos

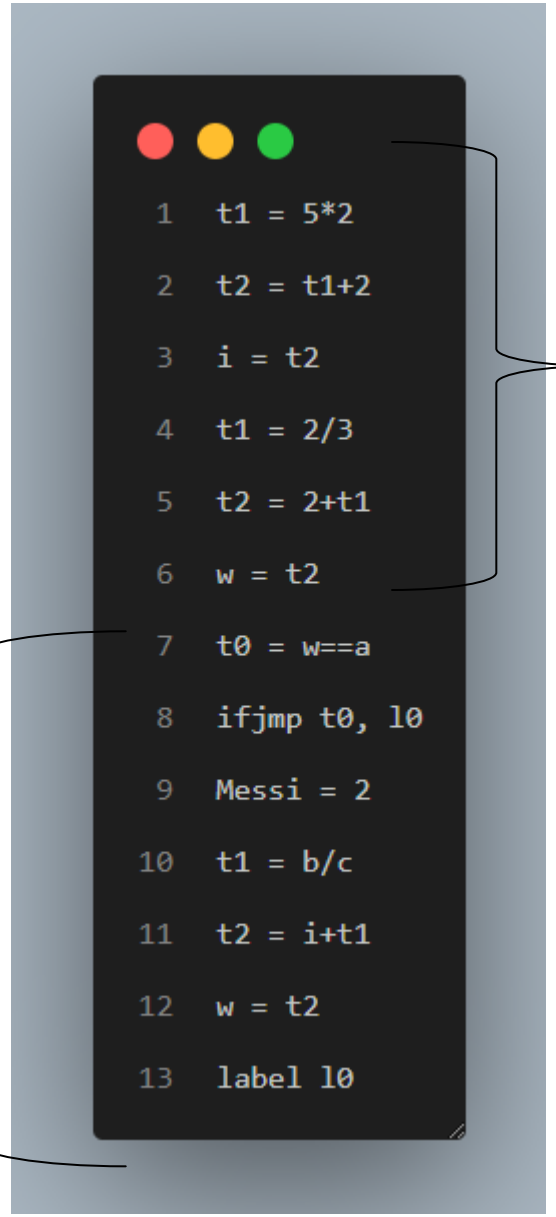
"TablaSimbolos.PNG"

	String	Nombre	TipoDato	Inicializado	Expresion	Valor	Usado
0	a	a	INT	True	Null	1	False
1	b	b	INT	True	Null	2	True
2	c	c	INT	True	Null	3	True
3	i	i	INT	True	5*2+b	12	True
4	j	j	INT	True	Null	0	False
5	o	o	INT	False	Null	Null	False
6	p	p	INT	False	Null	Null	False
7	r	r	INT	False	Null	Null	False
8	s	s	INT	False	Null	Null	True
9	t	t	INT	True	Null	10-9	False
10	w	w	INT	True	Messi+b/c	12	False
11	x	x	INT	True	Null	10	False
12	y	y	INT	False	Null	Null	False
13	Messi	Messi	INT	True	Null	2	False



Generando este Tree.Parser()

Y por ultimo, generando el siguiente Código intermedio:



```
1   t1 = 5*2
2   t2 = t1+2
3   i = t2
4   t1 = 2/3
5   t2 = 2+t1
6   w = t2
7   t0 = w==a
8   ifjmp t0, l0
9   Messi = 2
10  t1 = b/c
11  t2 = i+t1
12  w = t2
13  label l0
```

Declaración / Asignación

Instrucción IF