UNIVERSIDAD TECNOLOGICA NACIONAL

FACULTAD REGIONAL BUENOS AIRES

Trabajo Práctico

“Tesis Final”

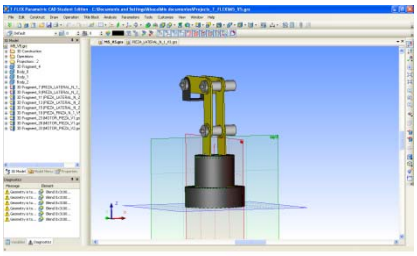
Materia: Robotica

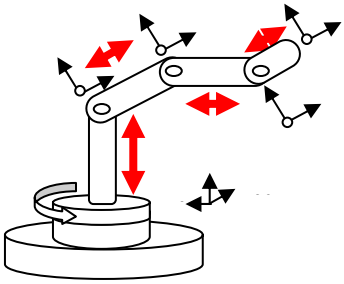
Integrantes: Abaca, Daniel Alberto

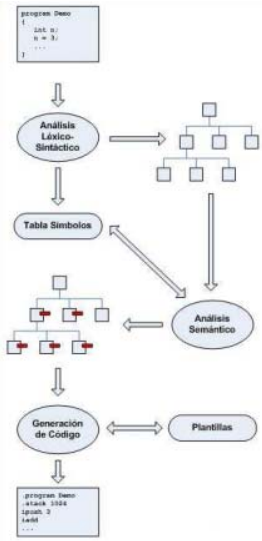
Tunez, Diego

Profesor: Mas. Ing. Giannetta Hernan

JTP: Ing. Granzella Damian







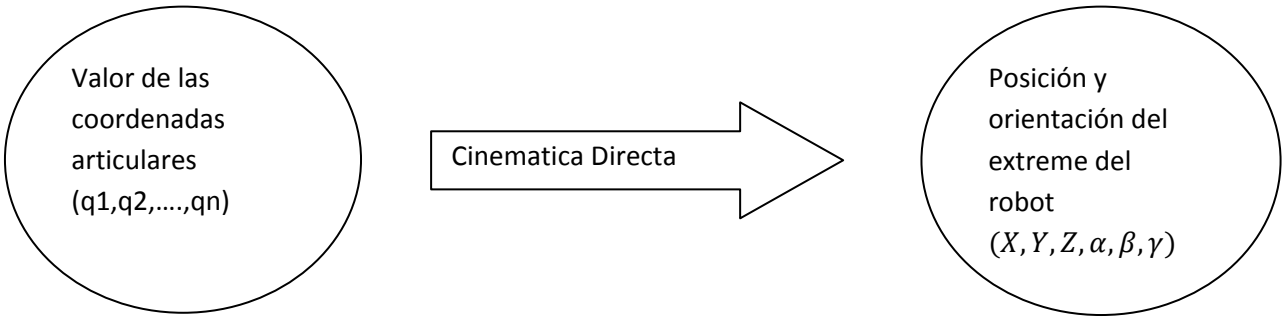
Introducción a la cinemática del robot:

A continuación se dará un resumen sobre la teoría necesaria para el desarrollo de la práctica posterior.

Definición de Cinemática: Es la parte de la Física que estudia el movimiento sin tener en cuenta las causas que lo producen, limitándose esencialmente al estudio de la trayectoria en función del tiempo.

La cinemática del robot se puede resolver mediante cinemática directa o cinemática inversa para lo que es el desarrollo de este trabajo práctico nos volcaremos al estudio de la cinemática directa del robot.

La cinemática directa consiste básicamente en determinar la posición y orientación del extremo del robot $(X, Y, Z, \alpha, \beta, \gamma)$ en función del valor de las coordenadas articulares.



El método que permite resolver la cinemática directa del robot es a través de un método matricial que propusieron Denavit y Hartenberg. Este permite establecer de manera sistemática un sistema de coordenadas {Si} ligado a cada eslabón i de una cadena articulada, pudiéndose determinar a continuación las ecuaciones cinemáticas de las ecuaciones de la cadena completa.

Según la representación de D.H, escogiendo adecuadamente los sistemas de coordenadas asociados a cada eslabón, será posible pasar de uno al siguiente mediante cuatro transformaciones básicas que dependen exclusivamente de las características geométricas de cada eslabón.

Estas transformaciones básicas consisten en una sucesión de rotaciones y traslaciones que permiten relacionar el sistema de referencia del elemento i con el sistema del elemento i-1.

Las transformaciones en cuestión son las siguientes (es importante recordar que el paso del sistema $\{S_{i-1}\}$ al $\{S_i\}$ mediante estas cuatro transformaciones está garantizado solo si los sistemas $\{S_{i-1}\}$ al $\{S_i\}$ han sido definidos de acuerdo a unas normas determinadas que se expondrán posteriormente):

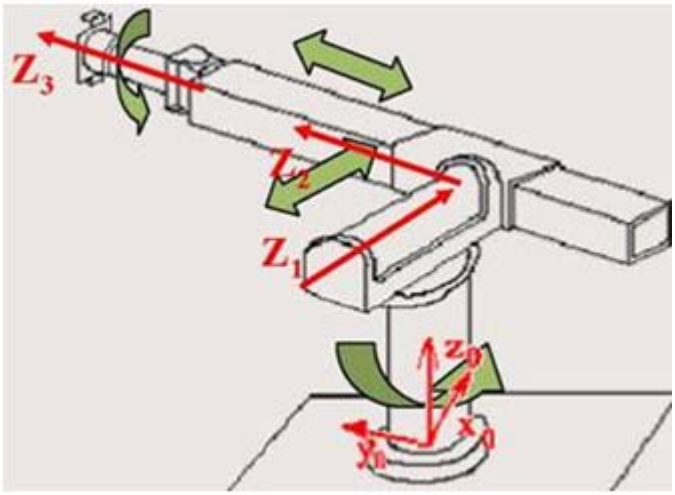
- Rotación alrededor del eje z_{i-1}un ángulo θ_i.
- Traslación a lo largo de z_{i-1}una distancia d_i; vector $d_i(0,0,d_i)$.
- Traslación a lo largo de x_i una distancia a_i; vector $a_i(0,0, a_i)$.
- Rotación alrededor del eje x_i un ángulo α_i.

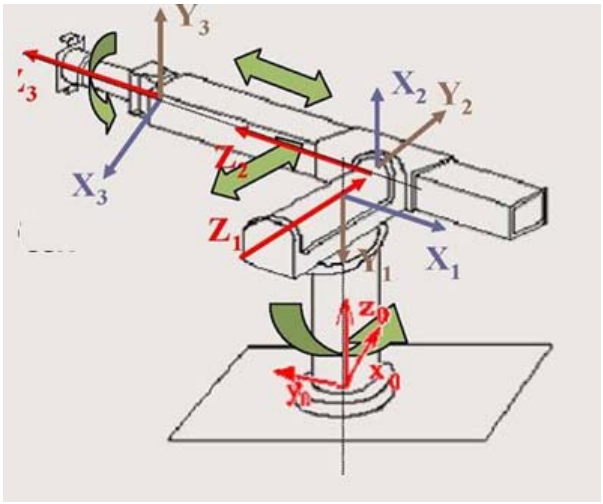
Como se dijo anteriormente por medio del método matricial Denavit y Hartenberg se obtiene el modelo cinematico del robot que es nada más y menos que ecuaciones que determinan la posición del extremo final del robot a través de los valores de las articulaciones y dimensiones del robot. Estas ecuaciones se obtienen considerando previamente las normas anteriormente mencionadas por medio del cálculo de la matriz de transformación homogénea total [T] que es el producto de matrices homogéneas que son en función de cada articulación. Para obtener dichas matrices se aplica el algoritmo de D.H.

Algoritmo de D.H:

- Elegir un sistema de coordenadas fijo (X0, Y0, Z0) asociado a la base del robot.
- Localizar el eje de cada articulación Z: Si la articulación es rotativa, el eje será el propio eje de giro. Si es prismática, el eje lleva a dirección de deslizamiento.

Ejemplo:





- Situar los ejes X a la línea normal común a Z_{i-1} y $Z_{i.}$
- Si estos son paralelos, se elige la línea normal que corta ambos ejes.
- El eje Yi debe completar el tríedro dextrógiro.

Parámetros D.H:

α_i: Angulo entre el eje Z_{i-1} y Z_i sobre el plano perpendicular X_i. El signo lo da la regla de la mano derecha.

a_i: Distancia entre los ejes Z_{i-1} y Z_i, a lo largo de X_i. El signo lo define el sentido X_i.

θ_i: Angulo que forman los ejes X_{i-1} y X_i, sobre el plano perpendicular a Z_i. El signo lo determina la regla de la mano derecha.

d_i:Distancia a lo largo del eje Z_{i-1} desde el origen del sistema S_{i-1} hasta la intersección del eje Z_i, con el eje X_i. En el caso de articulaciones prismáticas será la variable de desplazamiento.

Obtención de la Matriz homogénea Total [T]:

Una vez obtenidos los parámetros D.H para cada articulación en lo que se tendrá cuatro valores correspondiente a $(\alpha_{i,}a_{i,}\theta_i, d_i)$ para cada articulación q1, q2, q3,……, qn; estamos en condiciones de calcular la matriz homogénea para cada articulación respecto de otr articulación y finalmente la matriz de transformación homogénea total [T] como se muestra a continuación.

$${}^{i-1}A_i = Rot(Z_{i-1},\theta).Tras(0,0,d_i),Tras(a_i,0,0).Rot(X_i,\alpha_i)$$

Rot: Matriz homogénea de rotación; Tras: Matriz homogénea de traslación

$${}^{i-1}A_i = \begin{bmatrix} Cos(\theta i) & -Sen(\theta i) & 0 & 0 \\ Sen(\theta i) & Cos(\theta i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & di \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & ai \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & Cos(\alpha i) & -Sen(\alpha i) & 0 \\ 0 & Sen(\alpha i) & Cos(\alpha i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^{i-1}A_i = \begin{bmatrix} Cos(\theta i) & -Cos(\alpha i).Sen(\theta i) & Sen(\alpha i).Sen(\theta i) & ai.Cos(\theta i) \\ Sen(\theta i) & Cos(\alpha i)Cos(\theta i) & -Sen(\alpha i).Cos(\theta i) & aiSen(\theta i) \\ 0 & Sen(\alpha i) & Cos(\alpha i) & di \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

${}^{i-1}A_i$: $Matriz\ de\ transformacion\ homogenea\ desde\ el\ sistema\ i-1\ hasta\ el\ i(una\ por\ cada\ articulacion)$

Finalmente:

$$[T] = {}^{0}A_n = {}^{0}A_1.\ {}^{1}A_2 \dots\dots {}^{i-1}A_i \dots\dots {}^{n-1}A_n$$

[T]: Matriz de transformación homogénea total

Desarrollo de la práctica:

Antes de implementar el código en lenguaje C en el CodeWarrior para el DSP56800/E de la cadena cinemática directa del robot **M5 sin considerar el gripper** se procederá a calcular la matriz de transformación homogénea llamada matriz **[T]** de toda la cadena cinemática, aplicando el método matricial de **Denavit Hartenberg(D.H)**.

Primeramente se procede esquematizar el robot M5 para la aplicación del método de D.H indicando principalmente las articulaciones y los sistemas de referencias convenientes como se muestra a continuación:

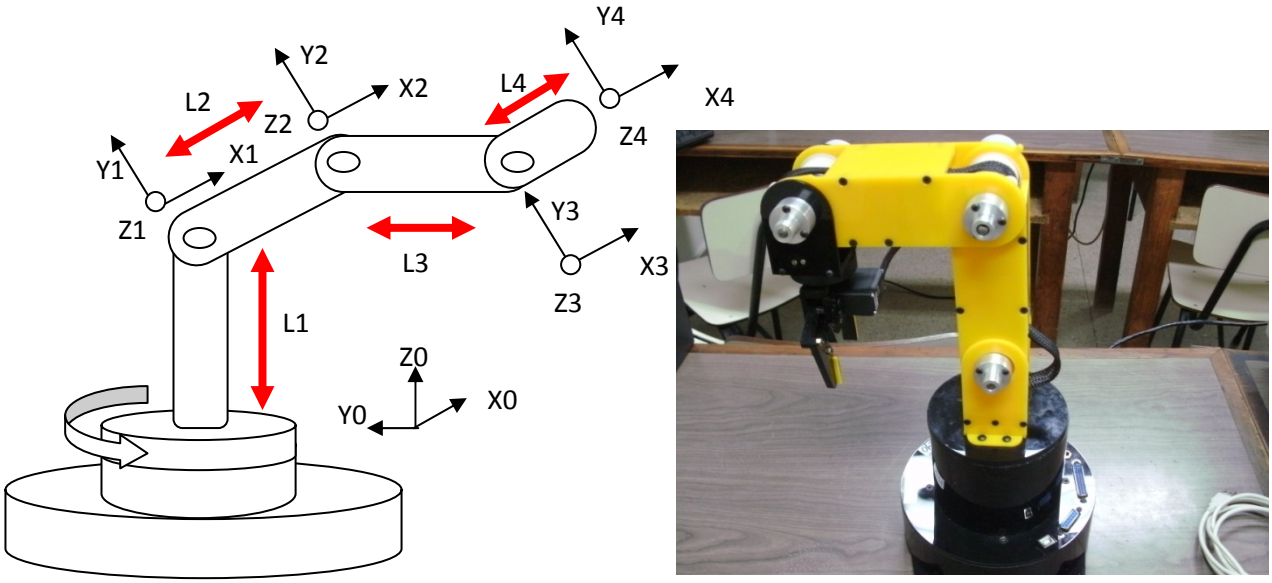


Figura_1: A la izquierda modelo del robot M5 sin considerar el gripper y a la derecha el robot M5.

En base a la figura_1 se aplica el método D.H y se determina para cada articulación los parámetros $\boldsymbol{\theta}, \mathbf{d}, \mathbf{a}, \boldsymbol{\alpha}$ que se detallan a continuación en una tabla:

	θ	d	a	α
Articulación 1	q1	L1	0	90 grados
Articulación 2	q2	0	L2	0
Articulación 3	q3	0	L3	0
Articulación 4	q4	0	L4	0

Nota: En realidad el robot M5 tiene cinco articulaciones (cinco grados de libertad) pero a fin de desarrollar este TP se nos pidió que lo consideremos solo con cuatro articulaciones (cuatro grados de libertad).

En función de la tabla anterior; se tiene una matriz homogénea particular para dos eslabones o articulaciones consecutivas **de acuerdo a una norma establecida** donde los valores de dicha matriz dependen de los parámetros $\boldsymbol{\theta}, \mathbf{d}, \mathbf{a}, \boldsymbol{\alpha}$ de la tabla y está dada por:

$$ {}^{i-1}A_i = \begin{bmatrix} Cos(\theta i) & -Cos(\alpha i).Sen(\theta i) & Sen(\alpha i).Sen(\theta i) & ai.Cos(\theta i) \\ Sen(\theta i) & Cos(\alpha i)Cos(\theta i) & -Sen(\alpha i).Cos(\theta i) & aiSen(\theta i) \\ 0 & Sen(\alpha i) & Cos(\alpha i) & di \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

${}^{i-1}A_i$: Es una matriz de transformación homogénea que representa la orientación y posición relativa entre los sistemas asociados a dos eslabones consecutivos.

Dado que solo consideraremos el robot M5 sin gripper tendremos **solo cuatro grados de libertad** por lo tanto se tendrán solo cuatro matrices de transformación homogéneas ${}^{i-1}A_i$ las mismas serán las que se detallan a continuación:

<u>Matrices de transformación homogénea referida a todos los sistemas asociados a dos eslabones consecutivos</u>

$$ {}^{0}A_1 = \begin{bmatrix} Cos(q1) & 0 & Sen(q1) & 0 \\ Sen(q1) & 0 & -Cos(q1) & 0 \\ 0 & 1 & 0 & L1 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

${}^{0}A_1$:Matriz de transformación homogénea que describe la posición y orientación del sistema de referencia solidario al primer eslabón con respecto al sistema de referencia solidario a la base.

q1: articulación numero 1(es un ángulo)

L1: distancia del sistema de referencia a la articulación 1

$$ {}^{1}A_2 = \begin{bmatrix} Cos(q2) & -Sen(q2) & 0 & L2.Cos(q2) \\ Sen(q2) & Cos(q2) & 0 & L2.Sen(q2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

${}^{1}A_2$:Matriz de transformación homogénea que describe la posición y orientación del sistema de referencia solidario al segundo eslabón con respecto al sistema de referencia solidario al primer eslabón.

q2: articulación numero 2(es un ángulo).

L2: distancia del sistema de referencia 1 a la articulación 2.

$$ {}^{2}A_3 = \begin{bmatrix} Cos(q3) & -Sen(q3) & 0 & L3.Cos(q3) \\ Sen(q3) & Cos(q3) & 0 & L3.Sen(q3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

${}^{2}A_3$:Matriz de transformación homogénea que describe la posición y orientación del sistema de referencia solidario al tercer eslabón con respecto al sistema de referencia solidario al segundo eslabón.

q3: articulación numero 3(es un ángulo).

L3: distancia del sistema de referencia 2 a la articulación 3.

$$ {}^{3}A_4 = \begin{bmatrix} Cos(q4) & -Sen(q4) & 0 & L4.Cos(q4) \\ Sen(q4) & Cos(q4) & 0 & L4.Sen(q4) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

${}^{3}A_4$:Matriz de transformación homogénea que describe la posición y orientación del sistema de referencia solidario al cuarto eslabón con respecto al sistema de referencia solidario al tercer eslabón.

q4: articulación numero 4(es un ángulo).

L4: distancia del sistema de referencia 3 al extremo final del robot (X4, Y4, Z4).

<u>Obtención de la matriz de transformación homogénea **[T]** de toda la cadena cinemática</u>

Finalmente obtenidas anteriormente estas cuatro matrices de transformación homogénea se procede a determinar la matriz de transformación homogénea **[T]**, simplemente multiplicando las cuatro matrices.

$$ [T]=\left[{}^{0}A_4 \right] = \left[{}^{0}A_1 \right]\left[{}^{1}A_2 \right]\left[{}^{2}A_3 \right]\left[{}^{3}A_4 \right] $$

$$ \left[{}^{0}A_1 \right].\left[{}^{1}A_2 \right] = \begin{bmatrix} C1 & 0 & S1 & 0 \\ S1 & 0 & -C1 & 0 \\ 0 & 1 & 0 & L1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} C2 & -S2 & 0 & L2.C2 \\ S2 & C2 & 0 & L2.S2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = $$

$$ =\left[{}^{0}A_1 \right].\left[{}^{1}A_2 \right] = \begin{bmatrix} C1.C2 & -C1.S2 & S1 & C1.L2.C2 \\ S1.C2 & -S1.S2 & -C1 & S1.L2.C2 \\ S2 & C2 & 0 & L2.S2 + L1 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Nota: $\boldsymbol{C1 = Cos(q1);\ S1 = Sen(q1);\ C2 = Cos(q2); S2 = Sen(q2)}$

$$ \left[{}^{0}A_1 \right].\left[{}^{1}A_2 \right].\left[{}^{2}A_3 \right] = \begin{bmatrix} C1.C2 & -C1.S2 & S1 & C1.L2.C2 \\ S1.C2 & -S1.S2 & -C1 & S1.L2.C2 \\ S2 & C2 & 0 & L2.S2 + L1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C3 & -S3 & 0 & L3.C3 \\ S3 & C3 & 0 & L3.S3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Nota: $\left[{}^{0}\boldsymbol{A_1} \right].\left[{}^{1}\boldsymbol{A_2} \right].\left[{}^{2}\boldsymbol{A_3} \right] = \left[{}^{0}\boldsymbol{A_3} \right]$

$$ \left[{}^{0}A_3 \right] = \begin{bmatrix} C1.C2.C3 - C1.S2.S3 & -C1.C2.S3 - C1.S2.C3 & S1 & C1.C2.L3.C3 - C1.S2.L3.S3 + C1.L2.C2 \\ S1.C2.C3 - S1.S2.S3 & -S1.C2.S3 - S1.S2.C3 & -C1 & S1.C2.L3.C3 - S1.S2L3.S3 + S1.L2.C2 \\ S2.C3 + C2.S3 & -S2.S3 + C2.C3 & 0 & S2.L3.C3 + C2.L3.S3 + L2.S2 + L1 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$[T]=\left[{}^{0}A_{4}\right]=[{}^{0}A_{3}][{}^{3}A_{4}]$

$$[T]=\begin{bmatrix} C1.C2.C3-C1.S2.S3 & -C1.C2.S3-C1.S2.C3 & S1 & C1.C2.L3.C3-C1.S2.L3.S3+C1.L2.C2 \\ S1.C2.C3-S1.S2.S3 & -S1.C2.S3-S1.S2.C3 & -C1 & S1.C2.L3.C3-S1.S2L3.S3+S1.L2.C2 \\ S2.C3+C2.S3 & -S2.S3+C2.C3 & 0 & S2.L3.C3+C2.L3.S3+L2.S2+L1 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} C4 & -S4 & 0 & L4.C4 \\ S4 & C4 & 0 & L4.S4 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}=$$

$$[T]=\begin{bmatrix} T11 & T12 & T13 & T14 \\ T21 & T22 & T23 & T24 \\ T31 & T32 & T33 & T34 \\ T41 & T42 & T43 & T44 \end{bmatrix}$$

$T11=C1.C2.C3.C4-C1.S2.S3.C4-C1.C2.S3.S4-C1.S2.C3.S4$

$T12=-C1.C2.C3.S4+C1.S2.S3.S4-C1.C2.S3.C4-C1.S2.C3.C4$

$T13=S1$

$T14=C1.C2.C3.L4.C4-C1.S2.S3.L4.C4-C1.C2.S3.L4.S4-C1.S2.C3.L4.S4+C1.C2.L3.S3-C1.S2.L3.S3+C1.L2.C2$

$T21=S1.C2.C3.C4-S1.S2.S3.C4-S1.C2.S3.S4-S1.S2.C3.S4$

$T22=-S1.C2.C3.S4+S1.S2.S3.S4-S1.C2.S3.C4-S1.S2.C3.C4$

$T23=-C1$

$T24=S1.C2.C3.L4.C4-S1.S2.S3.L4.C4-S1.C2.S3.L4.S4-S1.S2.C3.L4.S4+S1.C2.L3.S3-S1.S2.L3.S3+S1.L2.C2$

$T31=S2.C3.C4+C2.S3.C4-S2.S3.S4+C2.C3.S4$

$T32=-S2.C3.S4-C2.S3.S4-S2.S3.C4+C2.C3.C4$

$T33=0$

$T34=S2.C3.L4.C4+C2.S3.L4.C4-S2.S3.L4.S4+C2.C3.L4.S4+S2.L3.S3+C2.L3.S3+L2.S2+L1$

$T41=0$

$T42=0$

$T43=0$

$T44=1$

Obtenida la Matriz T realizamos la siguiente operación:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}=[T].\begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix}$$

Y hacemos u=0, v=0, w=0 (en el origen) tendremos las coordenadas **(x, y, z) en función de las coordenadas articulares q1, q2, q3, q4 y las dimensiones L1, L2, L3, L4 de nuestro robot M5**.

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}=\begin{bmatrix} T11 & T12 & T13 & T14 \\ T21 & T22 & T23 & T24 \\ T31 & T32 & T33 & T34 \\ T41 & T42 & T43 & T44 \end{bmatrix}.\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}=\begin{bmatrix} T14 \\ T24 \\ T34 \\ T44 \end{bmatrix}$$

Por lo tanto:

$$X = C1.C2.C3.L4.C4 - C1.S2.S3.L4.C4 - C1.C2.S3.L4.S4 - C1.S2.C3.L4.S4 + C1.C2.L3.S3 - C1.S2.L3.S3 + C1.L2.C2$$

$$Y = S1.C2.C3.L4.C4 - S1.S2.S3.L4.C4 - S1.C2.S3.L4.S4 - S1.S2.C3.L4.S4 + S1.C2.L3.S3 - S1.S2.L3.S3 + S1.L2.C2$$

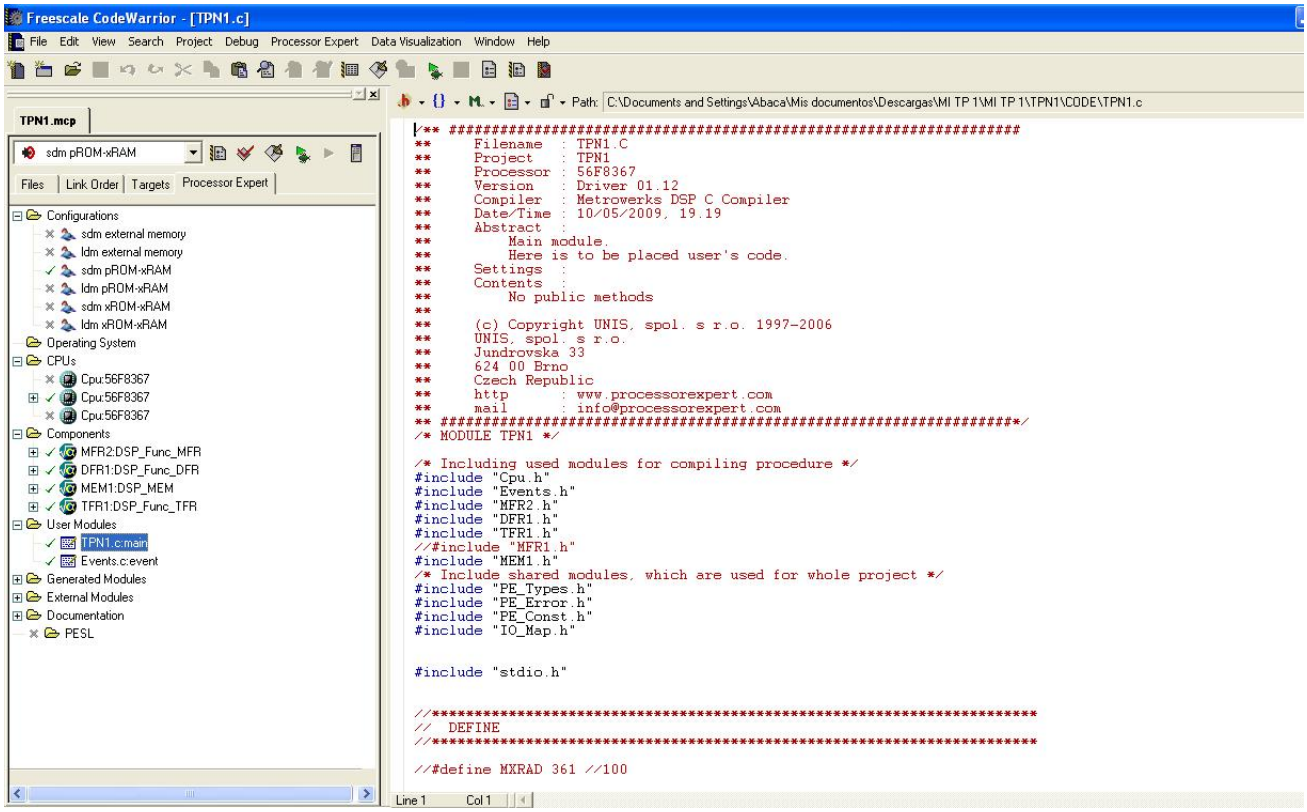
$$Z = S2.C3.L4.C4 + C2.S3.L4.C4 - S2.S3.L4.S4 + C2.C3.L4.S4 + S2.L3.S3 + C2.L3.S3 + L2.S2 + L1$$

Nota: X = X4; Y = Y4; Z = Z4

Implementación del modelo cinematico directo del Robot M5 a través del CodeWarrior

Dado el modelo cinematico directo obtenido en la hoja anterior a través de operaciones complejas entre matrices este puede ser realizado por medio de un procesador digital de señales (DSP). Para ello se utilizo el procesador 56F8367 de la familia 56800E sobre un entorno de desarrollo llamado Freescale CodeWarrior en este se implementara un código en lenguaje C para que desarrolle las operaciones entre matrices para llegar al modelo cinematico directo del RobotM5 y calcule así las coordenadas (X, Y, Z) del extremo final del mismo.

A continuación se muestra una imagen de cómo se ve el entorno de desarrollo del CodeWarrior



El Codewarrior puede descargarse gratis a través de internet en la página http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CW-56800E-DSC&fpsp=1&tab=Design_Tools_Tab

La misma es una versión estudiantil que está limitada en código hasta 64kb.

Luego de esta pequeña introducción se mostrara el código en C para nuestro RobotM5 considerando solo la solución para cuatro grados de libertad.

```
/** ###################################################################
**     Filename  : TPN1.C
**     Project   : TPN1
**     Processor : 56F8367
**     Version   : Driver 01.12
**     Compiler  : Metrowerks DSP C Compiler
**     Date/Time : 10/05/2009, 19.19
**     Abstract  :
**         Main module.
**         Here is to be placed user's code.
**     Settings  :
**     Contents  :
**         No public methods
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2006
**     UNIS, spol. s r.o.
**     Jundrovska 33
**     624 00 Brno
**     Czech Republic
**     http      : www.processorexpert.com
**     mail      : info@processorexpert.com
** ###################################################################*/


/* MODULE TPN1 */
```

```
/* Including used modules for compiling procedure */
#include "Cpu.h"
#include "Events.h"
#include "MFR2.h"
#include "DFR1.h"
#include "TFR1.h"
//#include "MFR1.h"
#include "MEM1.h"
/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
#include "stdio.h"
//*********************************************************************
//        DEFINE
//*********************************************************************
//#define MXRAD 361 //100
//#define PULSE2RAD 32767/MXRAD  // 32767/100 impulsos // #define PULSE2RAD 450
//*********************************************************************
//        GLOBAL VARIABLE
//*********************************************************************
//ac16 pulse2rad=PULSE2RAD ,testResult[MXRAD],testResult2[MXRAD];

//Word16 c16[MXRAD];
//Word32 c32[MXRAD];
```

```
Frac16 Homogenea[4][4];

Frac16 C1, S1, C2, S2, C3, S3, C4, S4, L1, L2, L3, L4, q1, q2, q3, q4;

Frac16 x, y, z, x0, y0, z0;

int j,k;

void main(void)

{

 /* Write your local variable definition here */

 /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/

 PE_low_level_init();

 /*** End of Processor Expert internal initialization.                    ***/

 /* Write your code here */

                    /* Inicializo */

                    q1 = 0;

                    q2 = 0x4000;

                    q3 = 0xC000;

                    q4 = 0xC000;

                    /*Para L1,L2, L3 y L4, normalizo respecto a 13*/

                    L1 = 0x175c;

                    L2 = 0x2876;

                    L3 = 0x2876;

                    L4 = 0xf0a;

                    x0 = 0;

                    y0 = 0;

                    z0 = 0;


printf ("\n\n!!!!!!!!!! VARIANDO q1 SOLAMENTE!!!!!!!!!!!!\n\n");
```

```
for(q1=-21915; q1<22511; q1 = q1+500)
                    {
                                C1 = tfr16CosPIx (q1);
                                C2 = tfr16CosPIx (q2);
                                C3 = tfr16CosPIx (q3);
                                C4 = tfr16CosPIx (q4);
                                S1 = tfr16SinPIx (q1);
                                S2 = tfr16SinPIx (q2);
                                S3 = tfr16SinPIx (q3);
                                S4 = tfr16SinPIx (q4);
//X=
 x        = add(
/*A*/  sub (sub ( mult(mult(mult(C1,C2),C3), mult(L4,C4)),
/*B*/           mult(mult(mult(C1,S2),S3), mult(L4,C4))) ,
/*C*/       add ( mult(mult(mult(C1,C2),S3), mult(L4,S4)),
/*D*/          mult(mult(mult(C1,S2),C3), mult(L4,S4)) ) ) ,
/*E*/  add (sub (mult(mult(C1,C2), mult(L3,S3)) ,
/*F*/          mult(mult(C1,S2), mult(L3,S3)) ),
/*G*/      mult(mult(C1,L2),C2)         )  );
 //Y=
 y = add(
/*A*/  sub (sub ( mult(mult(mult(S1,C2),C3), mult(L4,C4)),
/*B*/           mult(mult(mult(S1,S2),S3), mult(L4,C4))) ,
/*C*/       add ( mult(mult(mult(S1,C2),S3), mult(L4,S4)),
/*D*/          mult(mult(mult(S1,S2),C3), mult(L4,S4)) ) ) ,
/*E*/  add (sub (mult(mult(S1,C2), mult(L3,S3)) ,
```

```
/*F*/         mult(mult(S1,S2), mult(L3,S3)) ),

/*G*/      mult(mult(S1,L2),C2)         )  );

//Z=

  z = add(

/*A*/   add (add ( mult(mult(S2,C3), mult(L4,C4)),

/*B*/           mult(mult(C2,S3), mult(L4,C4))) ,

/*C*/       add ( mult(mult(C3,C2), mult(L4,S4)),

/*D*/            mult(mult(L3,S2),S3)  ) ),

/*E*/   add (add (mult(C2, mult(L3,S3)) ,

/*F*/           mult(L2,S2)) ,

/*G*/       sub (          L1  ,

/*H*/           mult(mult(S2,S3),mult (S4,L4)))));

                        printf ("\n%d\t%d\t%d", x, y, z);

                    }

printf ("\n\n!!!!!!!!!! VARIANDO q2 SOLAMENTE!!!!!!!!!!!!\n\n");

 for(q2=16384; q2<21594; q2 = q2+500)

                    {

                        C1 = tfr16CosPIx (q1);

                        C2 = tfr16CosPIx (q2);

                        C3 = tfr16CosPIx (q3);

                        C4 = tfr16CosPIx (q4);

                        S1 = tfr16SinPIx (q1);

                        S2 = tfr16SinPIx (q2);

                        S3 = tfr16SinPIx (q3);

                        S4 = tfr16SinPIx (q4);

//X=
```

```
 x        = add(
/*A*/  sub (sub ( mult(mult(mult(C1,C2),C3), mult(L4,C4)),
/*B*/           mult(mult(mult(C1,S2),S3), mult(L4,C4))) ,
/*C*/       add ( mult(mult(mult(C1,C2),S3), mult(L4,S4)),
/*D*/          mult(mult(mult(C1,S2),C3), mult(L4,S4)) ) ) ,
/*E*/  add (sub (mult(mult(C1,C2), mult(L3,S3)) ,
/*F*/          mult(mult(C1,S2), mult(L3,S3)) ),
/*G*/      mult(mult(C1,L2),C2)         )  );
//Y=
 y = add(
/*A*/  sub (sub ( mult(mult(mult(S1,C2),C3), mult(L4,C4)),
/*B*/           mult(mult(mult(S1,S2),S3), mult(L4,C4))) ,
/*C*/       add ( mult(mult(mult(S1,C2),S3), mult(L4,S4)),
/*D*/          mult(mult(mult(S1,S2),C3), mult(L4,S4)) ) ) ,
/*E*/  add (sub (mult(mult(S1,C2), mult(L3,S3)) ,
/*F*/          mult(mult(S1,S2), mult(L3,S3)) ),
/*G*/      mult(mult(S1,L2),C2)         )  );
//Z=
 z = add(
/*A*/  add (add ( mult(mult(S2,C3), mult(L4,C4)),
/*B*/           mult(mult(C2,S3), mult(L4,C4))) ,
/*C*/       add ( mult(mult(C3,C2), mult(L4,S4)),
/*D*/           mult(mult(L3,S2),S3)  ) ),
/*E*/  add (add (mult(C2, mult(L3,S3)) ,
/*F*/           mult(L2,S2)) ,
/*G*/       sub (          L1  ,
```

```
/*H*/           mult(mult(S2,S3),mult (S4,L4)))));
                  printf ("\n%d\t%d\t%d", x, y, z);
                  }
printf ("\n\n!!!!!!!!!! VARIANDO q3 SOLAMENTE!!!!!!!!!!!!\n\n");
 for(q3=-16384; q3<4915; q3 = q3+500)
                  {
                        C1 = tfr16CosPIx (q1);
                        C2 = tfr16CosPIx (q2);
                        C3 = tfr16CosPIx (q3);
                        C4 = tfr16CosPIx (q4);
                        S1 = tfr16SinPIx (q1);
                        S2 = tfr16SinPIx (q2);
                        S3 = tfr16SinPIx (q3);
                        S4 = tfr16SinPIx (q4);
//X=
  x         = add(
/*A*/   sub (sub ( mult(mult(mult(C1,C2),C3), mult(L4,C4)),
/*B*/           mult(mult(mult(C1,S2),S3), mult(L4,C4))) ,
/*C*/       add ( mult(mult(mult(C1,C2),S3), mult(L4,S4)),
/*D*/          mult(mult(mult(C1,S2),C3), mult(L4,S4)) ) ) ,
/*E*/  add (sub (mult(mult(C1,C2), mult(L3,S3)) ,
/*F*/          mult(mult(C1,S2), mult(L3,S3)) ),
/*G*/      mult(mult(C1,L2),C2)         )  );



//Y=
```

```
 y = add(
/*A*/  sub (sub ( mult(mult(mult(S1,C2),C3), mult(L4,C4)),
/*B*/           mult(mult(mult(S1,S2),S3), mult(L4,C4))) ,
/*C*/       add ( mult(mult(mult(S1,C2),S3), mult(L4,S4)),
/*D*/           mult(mult(mult(S1,S2),C3), mult(L4,S4)) ) ) ,
/*E*/  add (sub (mult(mult(S1,C2), mult(L3,S3)) ,
/*F*/         mult(mult(S1,S2), mult(L3,S3)) ),
/*G*/      mult(mult(S1,L2),C2)         )  );
//Z=
 z = add(
/*A*/  add (add ( mult(mult(S2,C3), mult(L4,C4)),
/*B*/           mult(mult(C2,S3), mult(L4,C4))) ,
/*C*/       add ( mult(mult(C3,C2), mult(L4,S4)),
/*D*/            mult(mult(L3,S2),S3)  ) ),
/*E*/  add (add (mult(C2, mult(L3,S3)) ,
/*F*/           mult(L2,S2)) ,
/*G*/       sub (          L1  ,
/*H*/           mult(mult(S2,S3),mult (S4,L4)))));
                  printf ("\n%d\t%d\t%d", x, y, z);
                  }
printf ("\n\n!!!!!!!!!! VARIANDO q4 SOLAMENTE!!!!!!!!!!!!\n\n");
```

```
for(q4=-16384; q4<18877; q4 = q4+500)
```

```
                {
                        C1 = tfr16CosPIx (q1);
                        C2 = tfr16CosPIx (q2);
                        C3 = tfr16CosPIx (q3);
                        C4 = tfr16CosPIx (q4);
                        S1 = tfr16SinPIx (q1);
                        S2 = tfr16SinPIx (q2);
                        S3 = tfr16SinPIx (q3);
                        S4 = tfr16SinPIx (q4);
//X=
 x        = add(
/*A*/  sub (sub ( mult(mult(mult(C1,C2),C3), mult(L4,C4)),
/*B*/          mult(mult(mult(C1,S2),S3), mult(L4,C4))) ,
/*C*/      add ( mult(mult(mult(C1,C2),S3), mult(L4,S4)),
/*D*/          mult(mult(mult(C1,S2),C3), mult(L4,S4)) ) ) ,
/*E*/  add (sub (mult(mult(C1,C2), mult(L3,S3)) ,
/*F*/         mult(mult(C1,S2), mult(L3,S3)) ),
/*G*/     mult(mult(C1,L2),C2)        )  );
 //Y=
 y = add(
/*A*/  sub (sub ( mult(mult(mult(S1,C2),C3), mult(L4,C4)),
/*B*/          mult(mult(mult(S1,S2),S3), mult(L4,C4))) ,
/*C*/      add ( mult(mult(mult(S1,C2),S3), mult(L4,S4)),
/*D*/          mult(mult(mult(S1,S2),C3), mult(L4,S4)) ) ) ,
/*E*/  add (sub (mult(mult(S1,C2), mult(L3,S3)) ,
/*F*/         mult(mult(S1,S2), mult(L3,S3)) ),
```

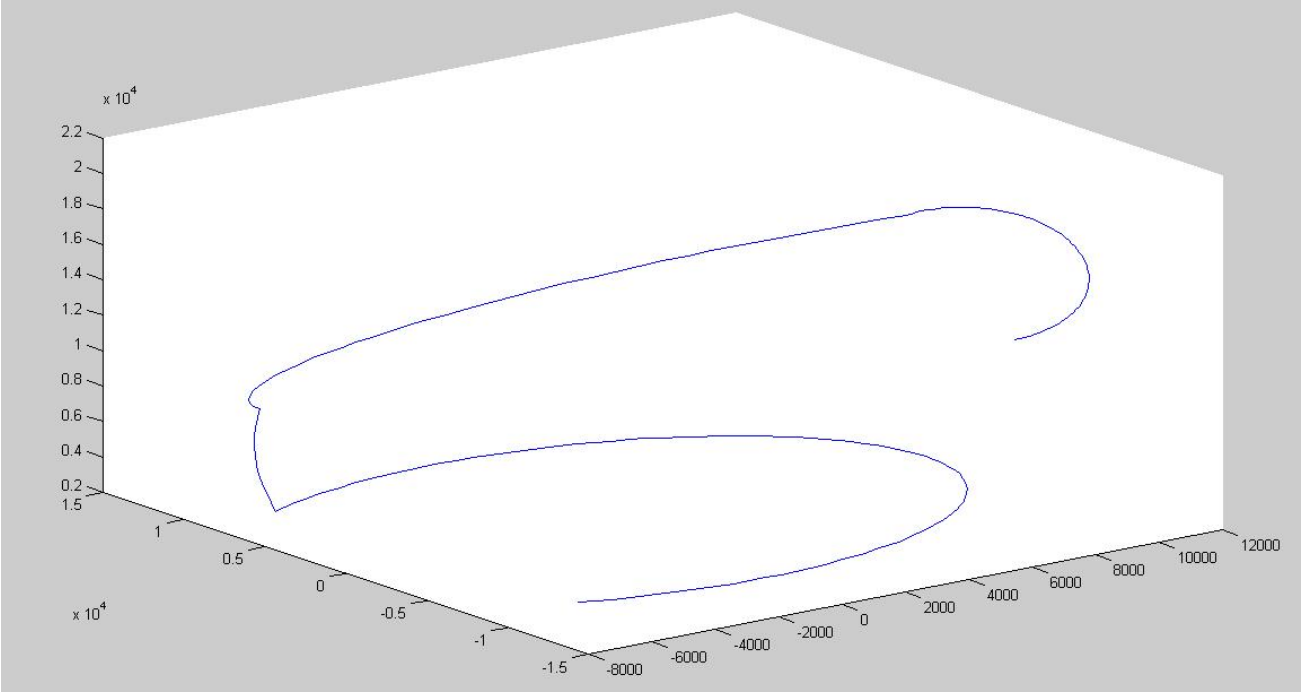
```
/*G*/      mult(mult(S1,L2),C2)         ) );

//Z=

 z = add(

/*A*/  add (add ( mult(mult(S2,C3), mult(L4,C4)),

/*B*/           mult(mult(C2,S3), mult(L4,C4))) ,

/*C*/       add ( mult(mult(C3,C2), mult(L4,S4)),

/*D*/            mult(mult(L3,S2),S3)  ) ),

/*E*/  add (add (mult(C2, mult(L3,S3)) ,

/*F*/           mult(L2,S2)) ,

/*G*/       sub (          L1  ,

/*H*/           mult(mult(S2,S3),mult (S4,L4)))));

                  printf ("\n%d\t%d\t%d", x, y, z);

                  }

}

/* END TPN1 */
```

En el código se puede observar una instrucción **printf** la cual muestra valores diferentes de la posición final del extremo del robotM5 (X, Y, Z) para distintos valores de las coordenadas articulares (q1, q2, q3, q4). Esto se realiza gracias a que el entorno de desarrollo CodeWarrior es un simulador y a través de la acción de **DEBUG** nos va mostrando los resultados de las coordenadas finales (X, Y, Z) producto de las operaciones matemáticas que realiza nuestro **DSP**.

Grafica de trayectoria del robot M5 en Matlab obtenidas por los resultados del DSP

A través de los resultados de las coordenadas (X, Y, Z) en función de distintos valores de las articulaciones (q1, q2, q3, q4) por medio del DSP. Estos pueden ser ingresados en una grafica tridimensional para visualizar la trayectoria que realiza el extremo final de nuestro robot(X, Y, Z).

Utilizando el programa Matlab a través de la función **plot3(x, y, z)** se puede determinar la trayectoria para distintos valores de (X, Y, Z). A continuación se muestra una trayectoria como ejemplo.



Simulación del robot M5 en Matlab a través del Robotics Toolbox

Para llevar a cabo la siguiente simulación es necesario disponer de la herramienta llamada **Robotics Toolbox for Matlab (release 8)** que se debe agregar a nuestro programa Matlab; este paquete puede descargarse de la página **www.petercorke.com**.

A continuación se detalla el script para lograr la simulación de nuestro robot M5 en base a nuestra tabla de los parámetros D.H que se vuelve a mostrar a continuación:

	θ	d	a	α
Articulación 1	q1	L1	0	90 grados
Articulación 2	q2	0	L2	0
Articulación 3	q3	0	L3	0
Articulación 4	q4	0	L4	0

En base a esta tabla se realiza el siguiente script:

```
clear%borra todo el workspace

clc%limpia pantalla

l1=link([90 0 0 7.3])%articulacion 1

l2=link([0 11.95 0 0])%articulacion 2

l3=link([0 11.95 0 0])%articulacion 3

l4=link([0 4.7 0 0])%articulacion 4

robo1=robot({l1 l2 l3 l4})

syms q1 q2 q3 q4

%tr=fkine(robo1,[q1 q2 q3 q4])

%tr=simple(tr)

plot(robo1,[0 0 0 0])

drivebot(robo1)
```

Nota: L1 = 7.3cm; L2 = 11.95cm; L3 = 11.95cm; L4 = 4.7cm. Son las dimensiones de nuestro robot M5.

Finalmente se obtiene la siguiente simulación:

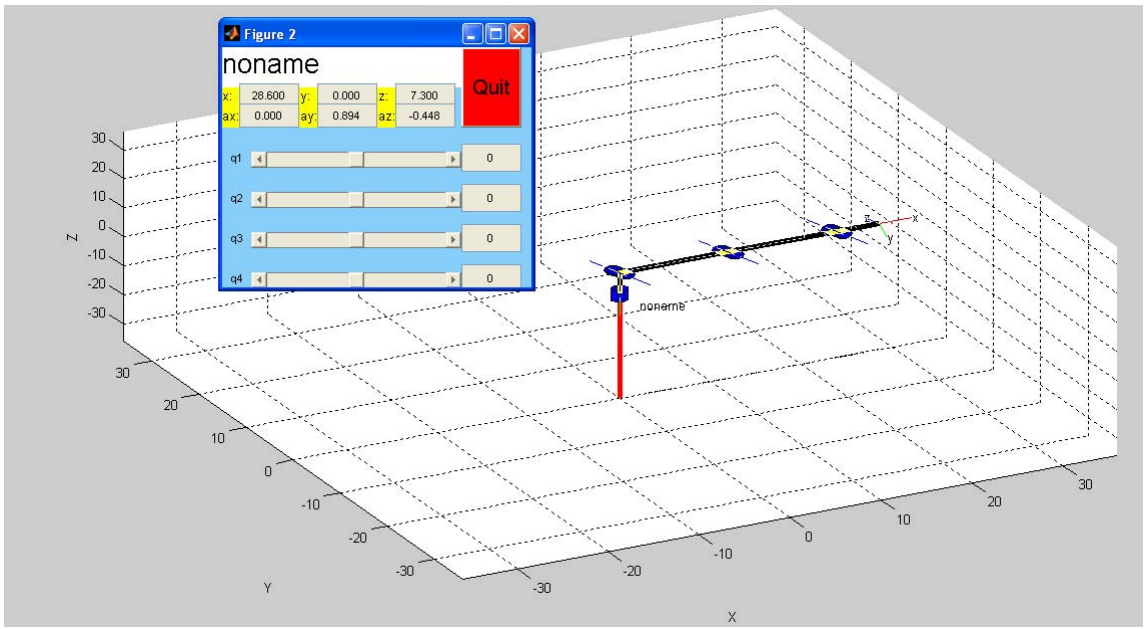


Figura_2: Robot M5 en posición inicial

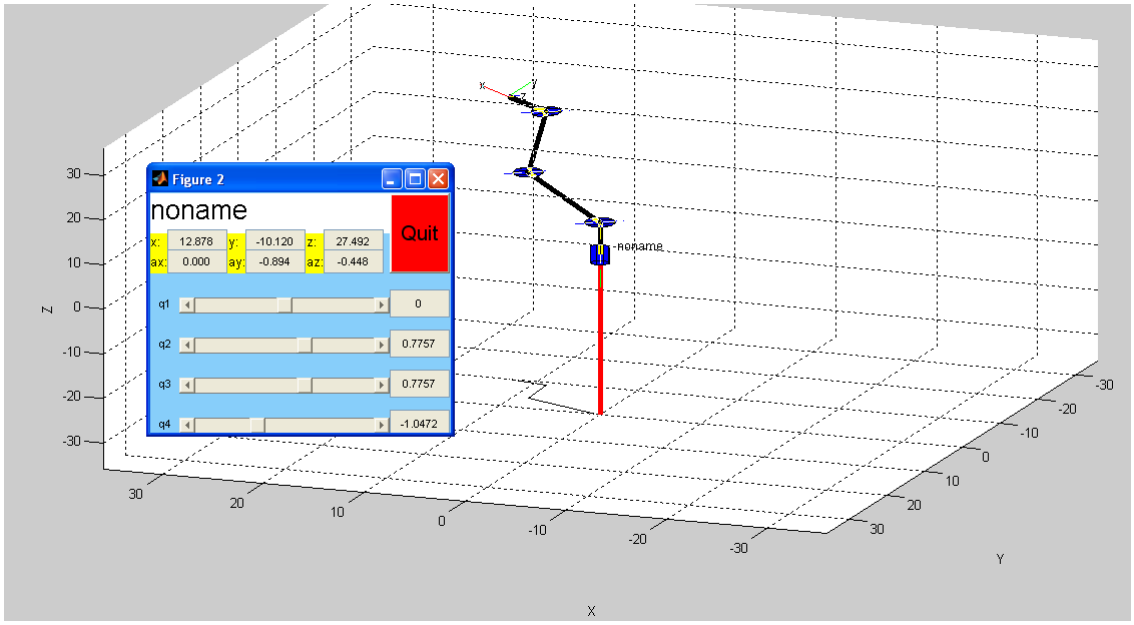


Figura_3: Robot M5 con variación en las articulaciones q2, q3, q4

Obtención del Modelo dinámico:

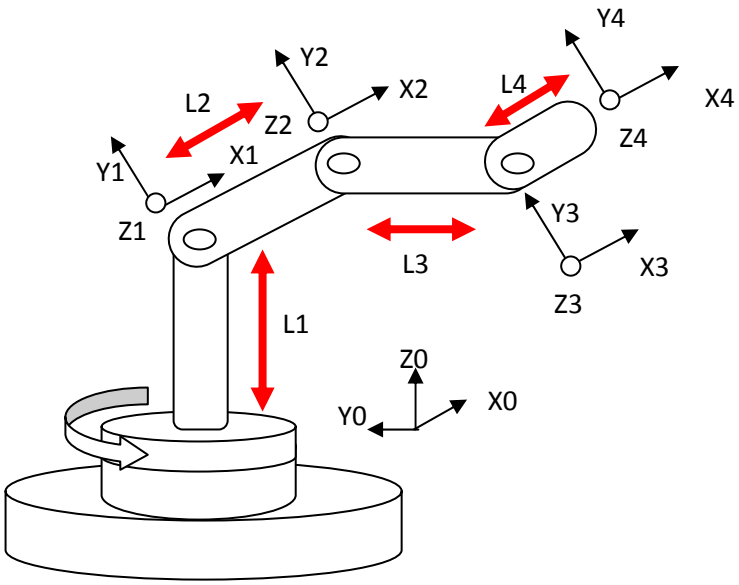
Se obtendrá el modelo dinámico a través del método Newton Euler utilizando el **toolbox Hemero** desarrollado por el señor Aníbal Ollero Baturone, quien explica su funcionamiento en el libro *Robótica: "Manipuladores y robots móviles"*.
Utilizando Matlab a través de la función **rne** permite obtener el modelo dinámico. Para ello simplemente hay que pasarle una matriz de parámetros **dyn (dinámica).**

Para hallar la matriz **dyn** necesitamos obtener algunos parámetros como ser las masas de las articulaciones y los parámetros de D.H que se calcularon en el trabajo práctico número 1 "Implementación de una Matriz Cinemática en DSP".

Parámetros Denavit-Hartenberg(D.H):

Estos ya fueron calculados en el trabajo práctico número 1 que ahora pasamos solo a presentarlos.

En base al siguiente diagrama se obtiene los siguientes parámetros D.H:



	θ	d	a	α
Articulación 1	q1	L1	0	90 grados
Articulación 2	q2	0	L2	0
Articulación 3	q3	0	L3	0
Articulación 4	q4	0	L4	0

Se necesita saber también otro parámetro que no está en la tabla anterior que se denomina **sigma(i)** este indicará el tipo de articulación (será **0** si es de rotación y **1** si por el contrario es prismática(traslacional)) el subíndice **i** indicara el numero de articulación para nuestro caso el valor de i estará **entre 1 a 4** (por ser cuatro grados de libertad) .

	$sigma$
Articulación 1	0
Articulación 2	0
Articulación 3	0
Articulación 4	0

Obtención de Parámetros utilizando el programa T-FLEX:

Se obtendrán parámetros como ser por ejemplo la masa de las articulaciones a través del programa llamado **T-FLEX** Parametric CAD en este caso se descargo una versión estudiantil con algunas limitaciones, permitiendo a estudiantes la utilización del producto, para conocer sus beneficios y características. Este software une funcionalidades de modelado paramétrico 3D con el conjunto de herramientas de producción y dibujo paramétrico. Es interoperable con otros programas 3D y 2D, a través de los siguientes formatos: Parasolid, IGES, PASO, Rhino, DWG, DXF, SolidWorks, Solid Edge, Autodesk Inventor, etc.

Link de descarga (Versión estudiantil):

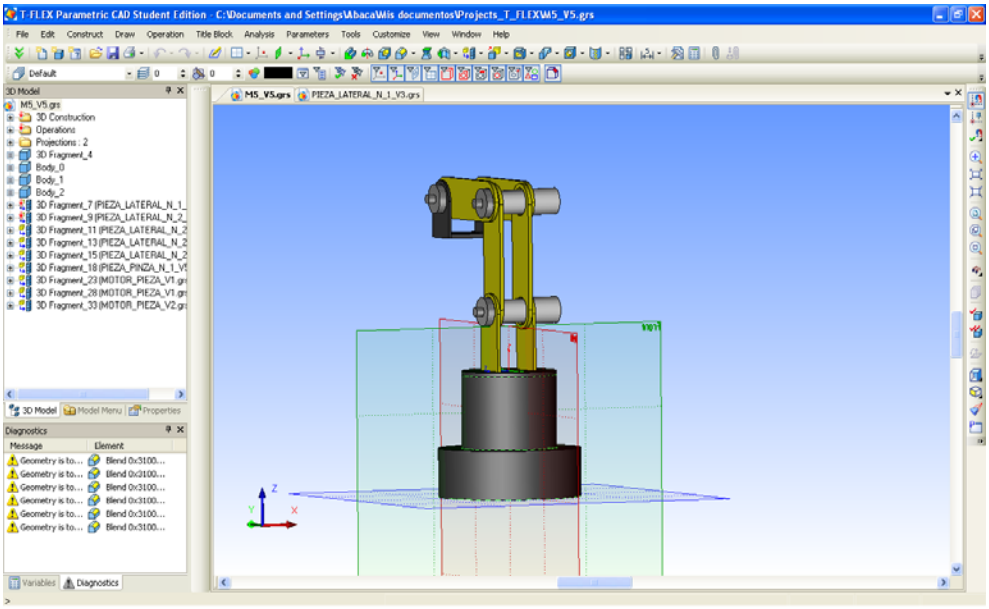
http://tflex.com/student/download.htm

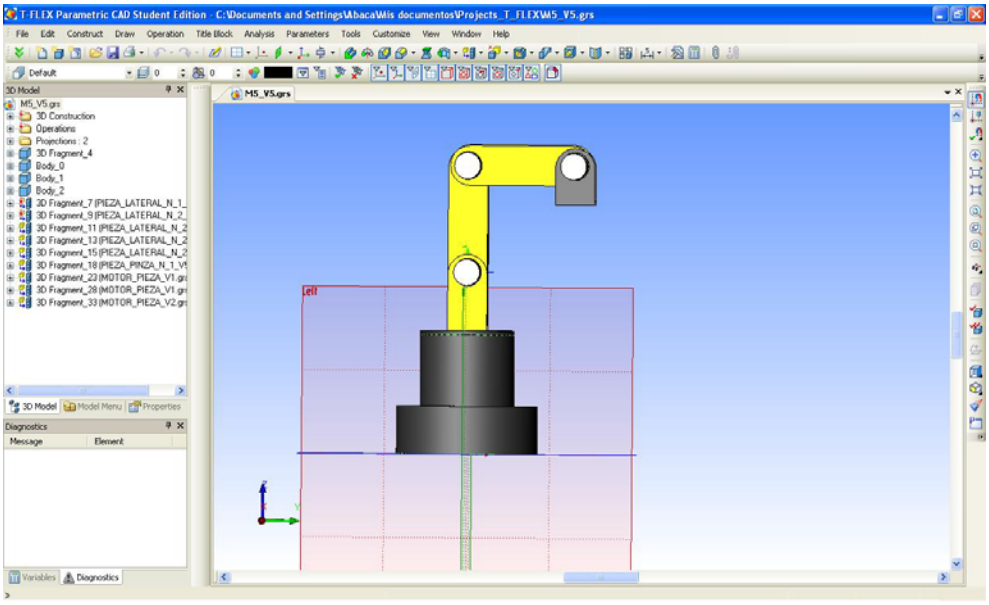
Volviendo al tema que nos interesa que es obtener **parámetros** como ser la masa de cada articulación. A continuación se indicara los pasos que se fueron realizando para cada articulación a fin de obtener los mismos.

En total son **5 pasos** que se detallan en la siguiente página:

Pasos:

1. Como primer paso se realizo el diseño del robot M5 muy aproximado y simplificado a modo de tener solamente un concepto general de todas las partes del robot M5 **sin tener en cuenta el gripper** dado que el estudio se baso solo en cuatro grados de libertad. A continuación se muestra dos imágenes generales.





2. Como primer cuerpo a hallar los parámetros partimos de la **articulación número 1**

Asignando adecuadamente las propiedades de nuestro cuerpo como ser el tipo de material que es plástico como se observa en la ventana **Body Properties**. Se obtienen los parámetros de nuestro interés como se muestra en la ventana de **Measure Element**.

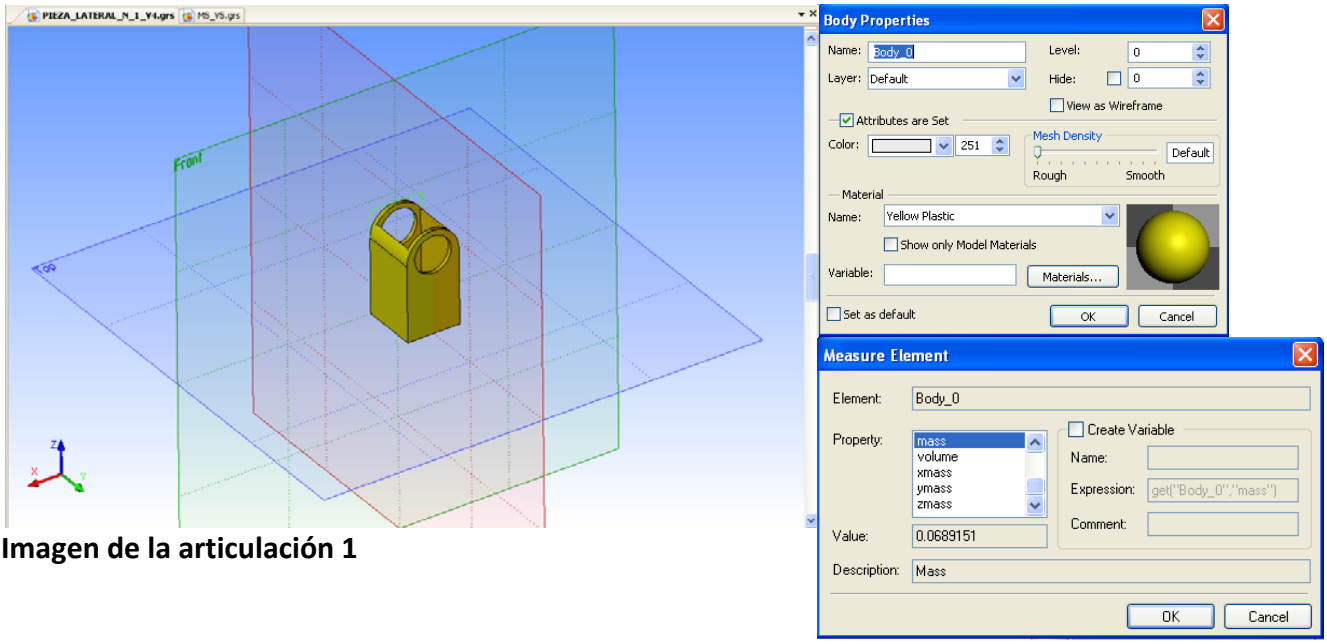


Imagen de la articulación 1

De la ventana de **Measure Element** se obtiene que la masa $\boldsymbol{M1 \cong 0.07Kg}$.

3. Parámetros de la **articulación numero 2**
Idem Paso 2

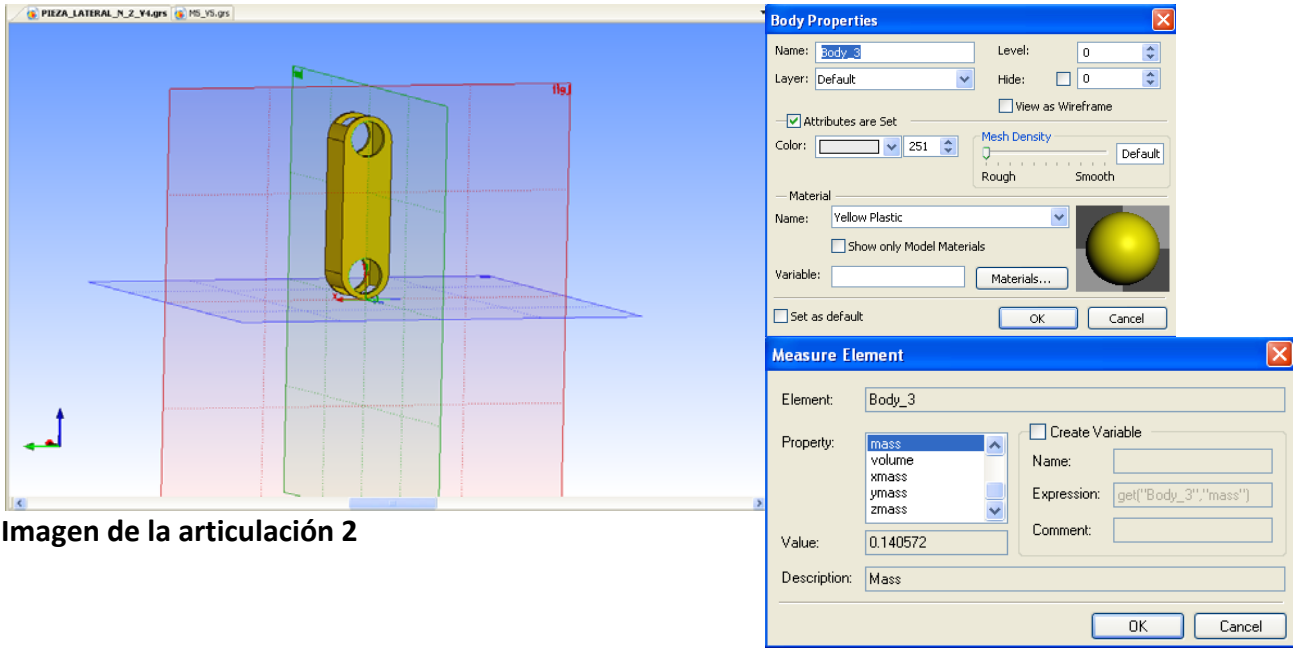


Imagen de la articulación 2

De la ventana de **Measure Element** se obtiene que la masa $\boldsymbol{M2 \cong 0.14Kg}$.

4. Parámetros de la **articulación numero 3**
   Idem Paso 2

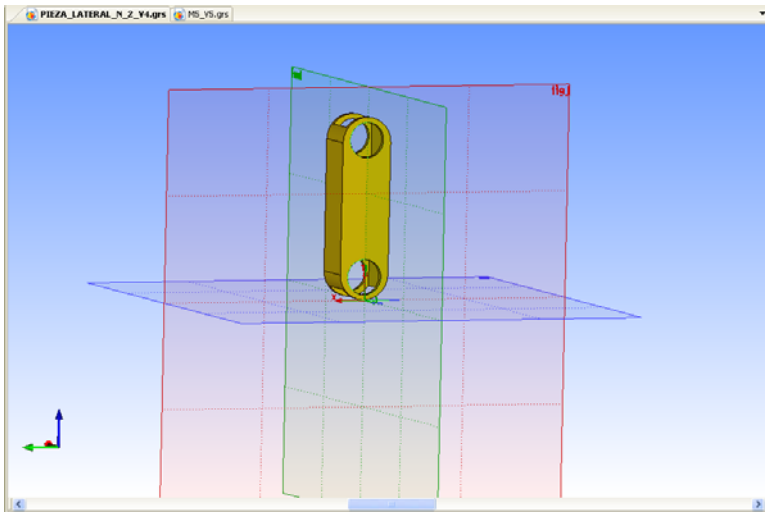
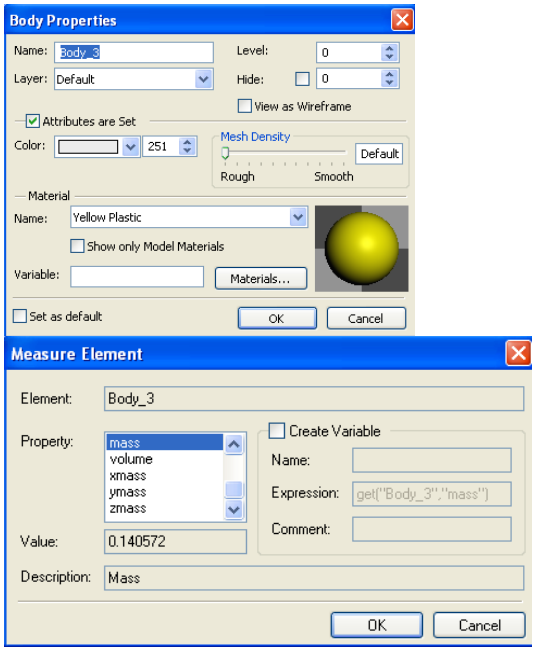


Imagen de la articulación 3



De la ventana de **Measure Element** se obtiene que la masa $\boldsymbol{M3 \cong 0.14Kg}$.

5. Parámetros de la **articulación numero 4**
   Idem Paso 2

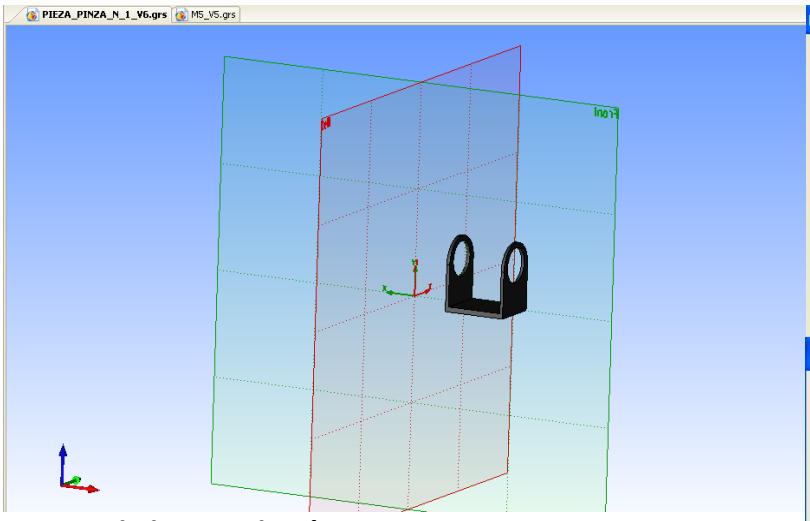
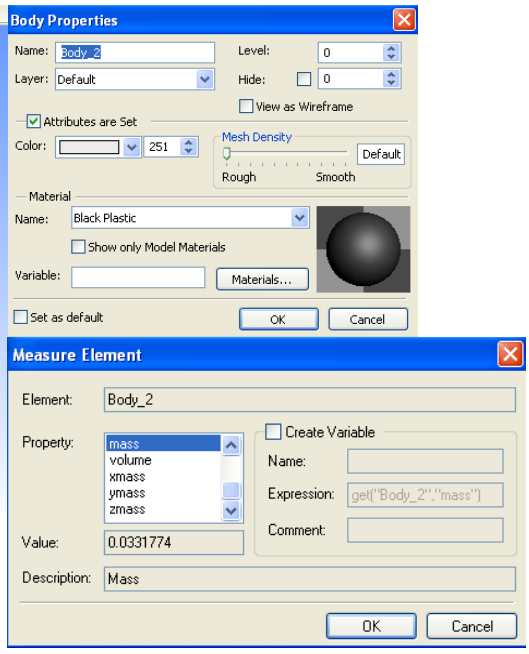


Imagen de la articulación 4



De la ventana de **Measure Element** se obtiene que la masa $\boldsymbol{M4 \cong 0.033Kg}$.

Aplicación de Matlab para la obtención del modelo dinámico mediante el método Newton-Euler:

Obtenidos los parámetros necesarios con el método de D.H y con el programa T-FLEX estamos en condiciones de calcular el modelo dinámico a través de la herramienta matemática Matlab utilizado el **toolbox Hemero** nombrado anteriormente.

A continuación se detallan los parámetros del robot M5, necesarios para el Toolbox Hemero:

	$Masas$
Articulación 1	M1 = 0.07kg
Articulación 2	M2 = 0.14kg
Articulación 3	M3 = 0.14kg
Articulación 4	M4 = 0.033Kg

	$Longitudes$
Articulación 1	l1 = 0.07m
Articulación 2	l2 = 0.12m
Articulación 3	l3 = 0.12m
Articulación 4	l4 = 0.05m

Centros de masas:

Para facilitar la resolución del problema se supondrá que las masas M1, M2, M3 y M4 están concentradas en los extremos de dichas articulaciones de nuestro robot M5.

$$lcm_1 = \begin{bmatrix} 0 \\ 0 \\ 0.08 \end{bmatrix} \; lcm_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \; lcm_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \; lcm_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Determinación de la matriz dyn:

Para resolver las ecuaciones se utiliza una matriz llamada **dyn**. Dicha matriz posee nx20 elementos, en donde n es la cantidad de articulaciones del robot a estudiar.

Las columnas de dicha matriz **dyn** se conforman con los siguientes datos:

Columna 1: alpha (i-1): Parámetros de Denavit-Hartenberg

Columna 2: a (i-1)

Columna 3: theta (i)

Columna 4: d (i)

Columna 5: sigma (i): Tipo de articulación; 0 si es de rotación y 1 si es prismática

Columna 6: masa: Masa del enlace i

Columna 7: rx: Centro de masas del enlace respecto al cuadro de referencia de dicho enlace
Columna 8: ry
Columna 9: rz

Columna 10: Ixx: Elementos del tensor de inercia referido al centro de masas del enlace
Columna 11: Iyy
Columna 12: Izz
Columna 13: Ixy
Columna 14: Iyz
Columna 15: Ixz

Columna 16: Jm: Inercia de la armadura

Columna 17: G: Velocidad de la articulación / velocidad del enlace

Columna 18: B: Fricción viscosa, referida al motor

Columna 19: Tc+: Fricción de Coulomb (rotación positiva), referida al motor
Columna 20: Tc-: Fricción de Coulomb (rotación negativa), referida al motor

En nuestro caso, la matriz **dyn** de **4x20 (no considerar la primera fila)** correspondiente sería la siguiente:

$$\begin{pmatrix} \boldsymbol{\alpha} & \boldsymbol{a} & \boldsymbol{\theta} & \boldsymbol{d} & \boldsymbol{sigma} & \boldsymbol{masa} & \boldsymbol{rx} & \boldsymbol{ry} & \boldsymbol{rz} & \boldsymbol{lxx} & \boldsymbol{lyy} & \boldsymbol{lzz} & \boldsymbol{lxy} & \boldsymbol{lyz} & \boldsymbol{lxz} & \boldsymbol{Jm} & \boldsymbol{G} & \boldsymbol{B} & \boldsymbol{Tc+} & \boldsymbol{Tc-} \\ \mathbf{90} & \mathbf{0} & \boldsymbol{t1} & \mathbf{0.07} & \mathbf{0} & \mathbf{0.07} & \mathbf{0} & \mathbf{0} & \mathbf{0.08} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0.12} & \boldsymbol{t2} & \mathbf{0} & \mathbf{0} & \mathbf{0.14} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0.12} & \boldsymbol{t3} & \mathbf{0} & \mathbf{0} & \mathbf{0.14} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0.05} & \boldsymbol{t4} & \mathbf{0} & \mathbf{0} & \mathbf{0.033} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix}$$

Matriz dyn

Simulación en Matlab utilizando el toolbox Hemero:

El objetivo es obtener las ecuaciones de **cupla, torque o par** necesarios para los **cuatro motores brushless** que operan las cuatro articulaciones de nuestro robot M5 (por considerar solo cuatro grados de libertad) a fin de saber que cupla debo aplicarle a cada motor a fin de obtener una respuesta satisfactoria al requerimiento solicitado a nuestro robot M5.

Luego, para calcular las correspondientes cuplas para cada articulación, se realizo el siguiente script en Matlab:

```
%Considerese ahora el empleo de la herramienta MATLAB. La funcion rne
%devuelve como resultado los pares ejercidos en cada articulacion.

clear
clc

%Para obtener el modelo dinamico simbolico bastan las siguientes lineas:

syms t1 t2 t3 t4 real; %Variables articulares tita 1, tita 2, tita 3, tita 4
syms td1 td2 td3 td4 real; %Velocidades articulares tita' 1, tita' 2, tita' 3, tita' 4
syms tdd1 tdd2 tdd3 tdd4 real; %Aceleraciones articulares tita''1, tita''2, tita''3 y tita''4
syms g real; %Aceleracion de la gravedad

%Escribimos la matriz con los parametros dinamicos del manipulador

dyn=[90 0 t1 0.07 0 0.07 0 0 0.08 0 0 0 0 0 0 0 1 0 0 0;
     0 0.12 t2 0 0 0.14 0 0 0 0 0 0 0 0 0 0 1 0 0 0;
     0 0.12 t3 0 0 0.14 0 0 0 0 0 0 0 0 0 0 1 0 0 0;
     0 0.05 t4 0 0 0.033 0 0 0 0 0 0 0 0 0 0 1 0 0 0];

q=[t1 t2 t3 t4]; %Vector de variables articulares
qd = [td1 td2 td3 td4]; %Vector de velocidades articulares
qdd = [tdd1 tdd2 tdd3 tdd4]; %Vector de aceleraciones articulares
grav = [0 -g 0]; %Vector de aceleracion de la gravedad
tau = rne(dyn, q, qd, qdd, grav)
simple (tau)

%De esta forma se obtiene una expresion simbolica de los pares.
%Tambien es posible evaluar cada uno de los terminos que intervienen en la
%expresion del par por separado. Para ello se escribe:

%M = inertia (dyn, q) %Matriz de masas M(tita)
%G = gravedad (dyn, q, grav) %Termino gravitatorio G(tita)
%V = coriolis (dyn, q, qd) %Terminos centrifugos y de coriolis V(tita, tita')
```

Con esto obtenemos un vector de 1 fila con 4 columnas, en donde están las cuplas de los motores brushless de cada articulación.
Al ejecutar estas instrucciones en Matlab, obtuvimos los siguientes pares motores, en forma simbólica. O sea, que de acuerdo a los valores de **posición, velocidad y aceleración** que le necesitemos obtener del robot, podemos calcular los pares motores que se deberán aplicar a las articulaciones del robot.

Resultados obtenidos con Matlab:

Expresión del torque (T1) para la articulación numero 1:

T1 = (70809*tdd1)/10000000 + (25737*tdd2)/10000000 + (33*tdd3)/400000 - (1557*td2^2*sin(t2))/625000 - (99*td3^2*sin(t3))/500000 + (519*g*cos(t1 + t2 - 90))/50000 + (519*g*cos(t1 + t2 + 90))/50000 + (99*tdd1*cos(t2 + t3))/250000 + (99*tdd2*cos(t2 + t3))/500000 + (99*tdd3*cos(t2 + t3))/500000 + (1557*tdd1*cos(t2))/312500 + (1557*tdd2*cos(t2))/625000 + (99*tdd1*cos(t3))/250000 + (99*tdd2*cos(t3))/250000 + (99*tdd3*cos(t3))/500000 + (33*g*cos(t1 + t2 + t3 - 90))/40000 + (33*g*cos(t1 + t2 + t3 + 90))/40000 - (99*td2^2*sin(t2 + t3))/500000 - (99*td3^2*sin(t2 + t3))/500000 + (939*g*cos(t1 - 90))/50000 + (939*g*cos(t1 + 90))/50000 - (99*td1*td2*sin(t2 + t3))/250000 - (99*td1*td3*sin(t2 + t3))/250000 - (99*td2*td3*sin(t2 + t3))/250000 - (1557*td1*td2*sin(t2))/312500 - (99*td1*td3*sin(t3))/250000 - (99*td2*td3*sin(t3))/250000

Expresión del torque (T2) para la articulación numero 2:

T2 = (25737*tdd1)/10000000 + (25737*tdd2)/10000000 + (33*tdd3)/400000 + (1557*td1^2*sin(t2))/625000 - (99*td3^2*sin(t3))/500000 + (519*g*cos(t1 + t2 - 90))/50000 + (519*g*cos(t1 + t2 + 90))/50000 + (99*tdd1*cos(t2 + t3))/500000 + (1557*tdd1*cos(t2))/625000 + (99*tdd1*cos(t3))/250000 + (99*tdd2*cos(t3))/250000 + (99*tdd3*cos(t3))/500000 + (33*g*cos(t1 + t2 + t3 - 90))/40000 + (33*g*cos(t1 + t2 + t3 + 90))/40000 + (99*td1^2*sin(t2 + t3))/500000 - (99*td1*td3*sin(t3))/250000 - (99*td2*td3*sin(t3))/250000

Expresión del torque (T3) para la articulación numero 3:

T3 = (33*tdd1)/400000 + (33*tdd2)/400000 + (33*tdd3)/400000 + (99*td1^2*sin(t3))/500000 + (99*td2^2*sin(t3))/500000 + (99*tdd1*cos(t2 + t3))/500000 + (99*tdd1*cos(t3))/500000 + (99*tdd2*cos(t3))/500000 + (33*g*cos(t1 + t2 + t3 - 90))/40000 + (33*g*cos(t1 + t2 + t3 + 90))/40000 + (99*td1^2*sin(t2 + t3))/500000 + (99*td1*td2*sin(t3))/250000

Expresión del torque (T4) para la articulación numero 4:

T4 = 0

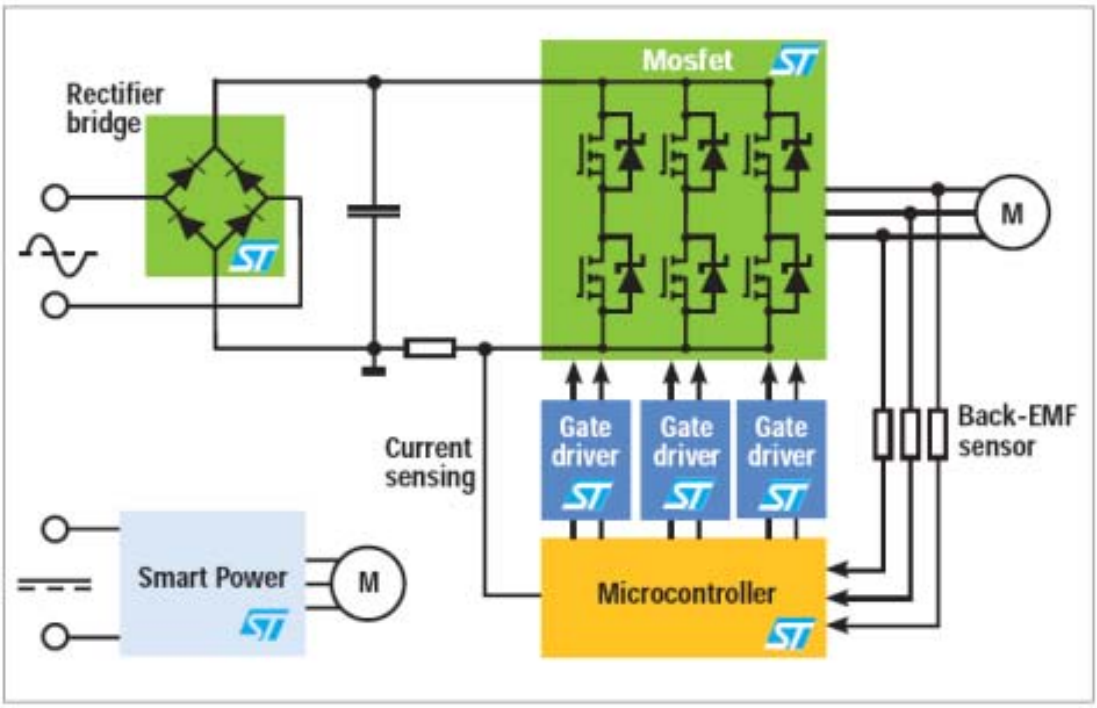
Finalmente se observa que las expresiones de los torque quedan en función de los siguientes parámetros:

t1, t2, t3, t4: Variables articulares tita1, tita2, tita3, tita4
td1, td2, td3, td4: Velocidades articulares tita'1, tita'2, tita'3, tita'4
tdd1, tdd2, tdd3, tdd4: Aceleraciones articulares tita''1, tita''2, tita''3 y tita''4
g=$9,8\frac{m}{s^2}$: Aceleración de la gravedad

Aclaración: La última articulación tiene el par motor igual a cero, ya que el extremo del robot está sin carga.

Implementación en código VHDL:

En esta etapa del trabajo, debemos implementar un sistema de control para un motor Brushless mediante PWM. Esto, se realizará con el uso de FPGA y un lenguaje descriptor de hardware (VHDL).



High-frequency three-phase brushless DC motor drive

Para realizar este control, hay que:

- Generar el PWM: Debe haber un módulo encargado de generar una señal de PWM. Este módulo recibe un dato (generalmente, de un microcontrolador) y entrega una señal rectangular, cuyo ciclo de actividad, depende del dato ingresado.

- Controlar el puente H: Hay que elegir que transistores deben activarse siguiendo la secuencia adecuada. Los sensores de efecto Hall permiten conocer la posición del rotor, y dependiendo de los cambios de estado de los mismos, se sabe cuando hay que activar cada transistor.

Para generar la señal de PWM, se utilizará el código del ejemplo dado por Atmel. Dicho código, se encuentra en:

http://www.atmel.com/dyn/resources/prod_documents/DOC2324.PDF

Dicho código es:

---- pwm_fpga.vhd

```
library IEEE;
USE     ieee.std_logic_1164.all;
USE    ieee.std_logic_arith.all ;
USE     work.user_pkg.all;


ENTITY pwm_fpga IS
PORT (  clock,reset     :in  STD_LOGIC;
      Data_value        :in  std_logic_vector(7 downto 0);
      pwm               :out STD_LOGIC
   );
END pwm_fpga;


ARCHITECTURE arch_pwm OF pwm_fpga IS

SIGNAL reg_out                : std_logic_vector(7 downto 0);
SIGNAL cnt_out_int            : std_logic_vector(7 downto 0);
SIGNAL pwm_int, rco_int       : STD_LOGIC;

BEGIN

 -- 8 BIT DATA REGISTER TO STORE THE MARKING VALUES .
 -- THE MARKING VALUES WILL DETERMINE THE DUTY CYCLE OF PWM OUTPUT
```

```
PROCESS(clock,reg_out,reset)
        BEGIN
            IF (reset ='1') THEN
            reg_out <="00000000";
            ELSIF (rising_edge(clock)) THEN
                reg_out <= data_value;
            END IF;
        END PROCESS;

-- 8 BIT UPDN COUNTER. COUNTS UP OR DOWN BASED ON THE PWM_INT SIGNAL AND GENERATES
-- TERMINAL COUNT WHENEVER COUNTER REACHES THE MAXIMUM VALUE OR WHEN IT TRANSISTS
-- THROUGH ZERO. THE TERMINAL COUNT WILL BE USED AS INTERRUPT TO AVR FOR GENERATING
-- THE LOAD SIGNAL.
-- INC and DEC are the two functions which are used for up and down counting. They are defined in sepearate user_pakge library

PROCESS (clock,cnt_out_int,rco_int,reg_out)
    BEGIN
     IF (rco_int = '1') THEN
        cnt_out_int <= reg_out;
     ELSIF  rising_edge(clock) THEN
    IF (rco_int = '0' and pwm_int ='1' and cnt_out_int <"11111111") THEN
          cnt_out_int <= INC(cnt_out_int);
    ELSE
       IF (rco_int ='0' and pwm_int ='0' and cnt_out_int > "00000000") THEN
```

```
                    cnt_out_int <= DEC(cnt_out_int);
                END IF;
            END IF;
        END IF;
END PROCESS;

-- Logic to generate RCO signal

PROCESS(cnt_out_int, rco_int, clock,reset)
        BEGIN

      IF (reset ='1') THEN
            rco_int <='1';
          ELSIF rising_edge(clock) THEN
       IF ((cnt_out_int = "11111111") or (cnt_out_int ="00000000")) THEN
              rco_int <= '1';
       ELSE
                rco_int <='0';
            END IF;
          END IF;

END PROCESS;

-- TOGGLE FLIP FLOP TO GENERATE THE PWM OUTPUT.

PROCESS (clock,rco_int,reset)
```

```
        BEGIN
                IF (reset = '1') THEN
                  pwm_int <='0';
                 ELSIF rising_edge(rco_int) THEN
                  pwm_int <= NOT(pwm_int);
                 ELSE
                  pwm_int <= pwm_int;
                END IF;
        END PROCESS;
        pwm <= pwm_int;

END arch_pwm;
```

Donde INC y DEC, se implementan en el siguiente código (también, de Atmel):

```
--user_pkg.vhd
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

PACKAGE user_pkg IS
   function INC(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
   function DEC(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
END user_pkg ;

PACKAGE BODY user_pkg IS
```

```
function INC(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
   variable XV: STD_LOGIC_VECTOR(X'LENGTH - 1 downto 0);
   begin
 XV := X;
 for I in 0 to XV'HIGH LOOP
    if XV(I) = '0' then
       XV(I) := '1';
       exit;
    else XV(I) := '0';
    end if;
 end loop;

return XV;
end INC;

function DEC(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
   variable XV: STD_LOGIC_VECTOR(X'LENGTH - 1 downto 0);
   begin
 XV := X;
 for I in 0 to XV'HIGH LOOP
    if XV(I) = '1' then
       XV(I) := '0';
       exit;
    else XV(I) := '1';
    end if;
```

```
 end loop;


return XV;

end DEC;


END user_pkg;
```

Una vez obtenida la generación de PWM en función de los datos de entrada, hay que controlar en puente H.

Dependiendo del valor de las entradas de los sensores Hall, hay que elegir que transistores se van a activar, y que transistor va a recibir los pulsos de PWM. La activación/desactivación de cada transistor, se realiza según el siguiente diagrama de estados:

Estado	Q1	Q2	Q3	Q4	Q5	Q6
0	1	0	0	PwmSignal	0	0
1	1	0	0	0	0	PwmSignal
2	0	0	1	0	0	PwmSignal
3	0	PwmSignal	1	0	0	0
4	0	PwmSignal	0	0	1	0
5	0	0	0	PwmSignal	1	0

El código es el siguiente:

```
--Driver motor fpga

library IEEE;

USE ieee.std_logic_1164.all;

USE ieee.std_logic_arith.all;


ENTITY driver_motor_fpga IS

PORT (  ClockDriver    :in  STD_LOGIC;
```

```
        ResetDriver    :in  STD_LOGIC;
        SentidoGiro    :in  STD_LOGIC;
      DataDriver       :in  std_logic_vector(7 downto 0);
      SensoresHall     :in  std_logic_vector(2 downto 0);
      Q5Q3Q1           :out std_logic_vector(2 downto 0);
      Q2               :out STD_LOGIC;
      Q4               :out STD_LOGIC;
      Q6               :out STD_LOGIC
    );
END driver_motor_fpga;

ARCHITECTURE Arch_Driver_motor_fpga OF driver_motor_fpga IS

 --Este componente, es el definido en pwm_fpga.vhd del ejemplo hecho por ATMEL
 COMPONENT pwm_fpga
   PORT (clock: IN STD_LOGIC;
         reset: IN STD_LOGIC;
         data_value: IN std_logic_vector(7 downto 0);
         pwm: OUT STD_LOGIC);
 END COMPONENT;

SIGNAL PwmSignal: STD_LOGIC;

BEGIN
  -- Conexion del modulo pwm
  ModuloPWM: pwm_fpga
```

```
        PORT MAP(
                        clock => ClockDriver,
                        reset => ResetDriver,
                        data_value => DataDriver,
                        pwm => PwmSignal
                        );

PROCESS(SensoresHall,SentidoGiro, PwmSignal)
 BEGIN
   IF(SentidoGiro = '0') THEN --En caso de que gire en sentido horario

     CASE SensoresHall IS
       WHEN "001" =>
        Q5Q3Q1 <= "001";
        Q6 <= PwmSignal;
        Q4 <= '0';
        Q2 <= '0';
       WHEN "000" =>
        Q5Q3Q1 <= "001";
        Q6 <= '0';
        Q4 <= PwmSignal;
        Q2 <= '0';
       WHEN "100" =>
        Q5Q3Q1 <= "100";
        Q6 <= '0';
        Q4 <= PwmSignal;
```

```
        Q2 <= '0';
      WHEN "110" =>
        Q5Q3Q1 <= "100";
        Q6 <= '0';
        Q4 <= '0';
        Q2 <= PwmSignal;
      WHEN "111" =>
        Q5Q3Q1 <= "010";
        Q6 <= '0';
        Q4 <= '0';
        Q2 <= PwmSignal;
      WHEN "011" =>
        Q5Q3Q1 <= "010";
        Q6 <= PwmSignal;
        Q4 <= '0';
        Q2 <= '0';
      WHEN OTHERS => NULL;
  END CASE;
END IF;

IF(SentidoGiro = '1') THEN --En caso de que gire en sentido antihorario

  CASE SensoresHall IS
       WHEN "011" =>
         Q5Q3Q1 <= "100";
         Q6 <= '0';
```

```
    Q4 <= PwmSignal;
    Q2 <= '0';
WHEN "111" =>
    Q5Q3Q1 <= "001";
    Q6 <= '0';
    Q4 <= PwmSignal;
    Q2 <= '0';
WHEN "110" =>
    Q5Q3Q1 <= "001";
    Q6 <= PwmSignal;
    Q4 <= '0';
    Q2 <= '0';
WHEN "100" =>
    Q5Q3Q1 <= "010";
    Q6 <= PwmSignal;
    Q4 <= '0';
    Q2 <= '0';
WHEN "000" =>
    Q5Q3Q1 <= "010";
    Q6 <= '0';
    Q4 <= '0';
    Q2 <= PwmSignal;
WHEN "001" =>
    Q5Q3Q1 <= "100";
    Q6 <= '0';
    Q4 <= '0';
```

```
            Q2 <= PwmSignal;
          WHEN OTHERS => NULL;
        END CASE;
    END IF;
 END PROCESS;


 END ARCHITECTURE Arch_Driver_motor_fpga;
```

Ahora, para poder simularlo, se crea otra entidad, que use un componente driver_motor_fpga, y se generen las señales necesarias para probar su funcionamiento.

Para esto, se le va a generar una secuencia a los sensores Hall, para simular que el motor está girando.

El código es:

-- test_control.vhd

```
LIBRARY ieee;

use ieee.std_logic_1164.all;

ENTITY test_control IS

END test_control;


ARCHITECTURE Arch_Test_control OF test_control IS


COMPONENT driver_motor_fpga

PORT (  ClockDriver    :in  STD_LOGIC;

        ResetDriver    :in  STD_LOGIC;

        SentidoGiro    :in  STD_LOGIC;

        DataDriver     :in  std_logic_vector(7 downto 0);
```

```
        SensoresHall  :in  std_logic_vector(2 downto 0);
        Q5Q3Q1        :out std_logic_vector(2 downto 0);
        Q2            :out STD_LOGIC;
        Q4            :out STD_LOGIC;
        Q6            :out STD_LOGIC
    );
END COMPONENT;

-- Internal signal declaration
SIGNAL sig_clock       : std_logic;
SIGNAL sig_reset       : std_logic;
SIGNAL sig_sentido     : std_logic;
SIGNAL sig_data_value          : std_logic_vector(7 downto 0);
SIGNAL sig_HALLs       : std_logic_vector(2 downto 0);
SIGNAL Signal_Q2, Signal_Q4, Signal_Q6      : std_logic;
SIGNAL Sig_Q5Q3Q1 : std_logic_vector(2 downto 0);
shared variable ENDSIM: boolean:=false;
constant clk_period:TIME:=200 ns;

BEGIN
-- Genera el clock
clk_gen: process
   BEGIN
   If ENDSIM = false THEN
     sig_clock <= '1';
     wait for clk_period/2;
```

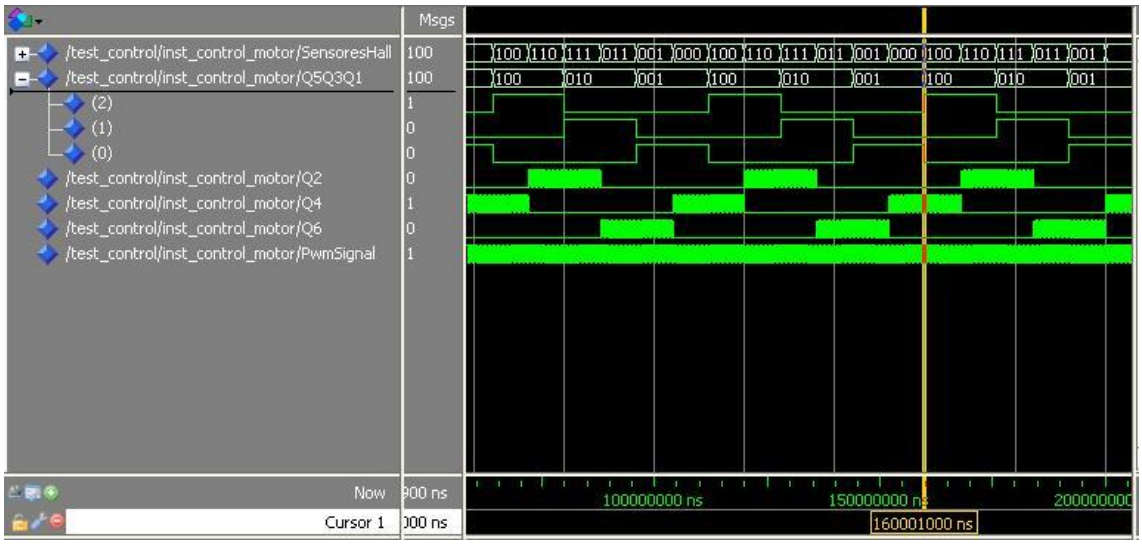
```
    sig_clock <= '0';
    wait for clk_period/2;
  else
    wait;
  end if;
 end process;

inst_control_motor : driver_motor_fpga
PORT MAP(
  ClockDriver => sig_clock,
  ResetDriver => sig_reset,
  SentidoGiro => sig_sentido,
  DataDriver  => sig_data_value,
  SensoresHall => sig_HALLs,
  Q2 => Signal_Q2,
  Q4 => Signal_Q4,
  Q6 => Signal_Q6,
  Q5Q3Q1 => Sig_Q5Q3Q1
 );

stimulus_process: PROCESS
-- Se dejan tiempos de 8mseg entre cada cambio de estado, para que gire a, aprox. 1200 RPM
BEGIN
 sig_sentido <= '0'; -- primero giro en sentido horario
 sig_reset <= '1';
 wait for 500 ns;
```

```
 sig_reset <= '0';
 sig_data_value <= "10000000";  -- Para el duty
 wait for 500 ns;
 for i in 1 to 5 loop   --simulo 5 vueltas del motor(sentido horario)aprox 1200RPM
   sig_HALLs <= "001";
   wait for 8 ms;
   sig_HALLs <= "000";
   wait for 8 ms;
   sig_HALLs <= "100";
   wait for 8 ms;
   sig_HALLs <= "110";
   wait for 8 ms;
   sig_HALLs <= "111";
   wait for 8 ms;
   sig_HALLs <= "011";
   wait for 8 ms;
end loop;
wait for 1000 ns;
sig_sentido <= '1'; -- Ahora, va a girar en sentido antihorario
for i in 1 to 5 loop   --simulo 5 vueltas del motor(sentido antihorario)
    sig_HALLs <= "011";
    wait for 7 ms;
    sig_HALLs <= "111";
    wait for 7 ms;
    sig_HALLs <= "110";
    wait for 7 ms;
```

```
    sig_HALLs <= "100";
    wait for 7 ms;
    sig_HALLs <= "000";
    wait for 7 ms;
    sig_HALLs <= "001";
    wait for 7 ms;
  end loop;
wait;
 END PROCESS stimulus_process;
END Arch_Test_control;
```

Resultados de la simulación:

Al realizar la simulación, se hizo el gráfico de: los sensores Hall, los transistores del puente H, y la señal a la salida del módulo PWM. Los resultados, son:



Compilador:

Introducción:

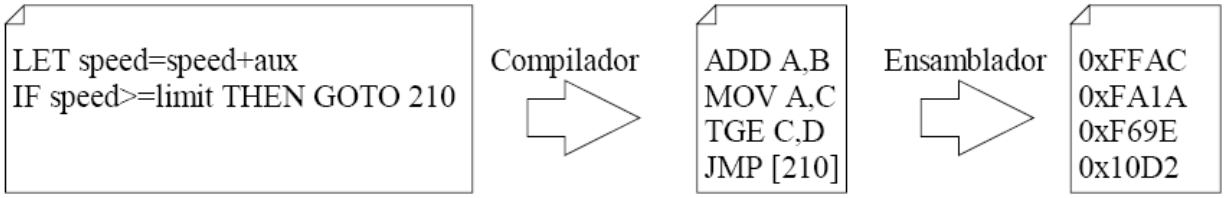
Se pretende dar una breve introducción a la herramienta **antlr**, una herramienta escrita en java y dedicada al desarrollo de intérpretes.

Un compilador es, a muy a groso modo "un programa que sirve para hacer otros programas, mediante el uso de un lenguaje de programación".

Los compiladores, por regla general, se limitan a traducir un lenguaje de programación de alto nivel a ensamblador, y después un ensamblador se encarga de generar el código de máquina. Un ensamblador es a su vez un programa. Dicho programa se encarga de traducir automáticamente el lenguaje ensamblador a código de máquina.

Al lenguaje de alto nivel que define un programa de software se lo suele llamar "código fuente".

Funcionamiento de un ensamblador

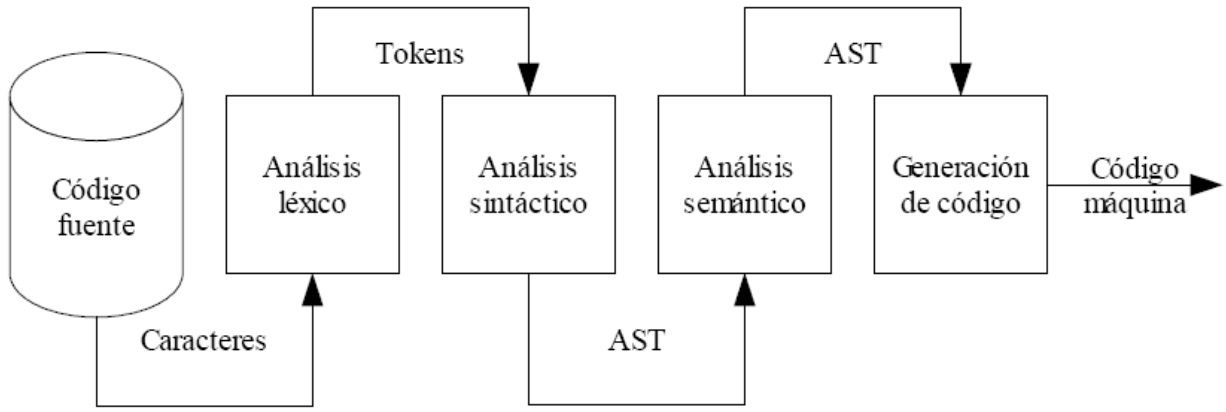


Funcionamiento de un compilador

Fases del proceso de compilación:

Lo primero que debe hacer un compilador es comprobar que la información que se le suministra pertenece a su lenguaje (no hay errores léxicos, sintácticos ni semánticos). Si es así, el intérprete debe representar de alguna manera la información que se le suministro para poder trabajar con ella, y finalmente traducir dicha información a código maquina.

Un esquema de dicho funcionamiento es el que se muestra en la siguiente figura:



Estructura básica de un compilador

Definiciones de las distintas etapas de compilación:

Código Fuente: Es información almacenada en la memoria de un ordenador. Suele tratarse de uno o varios ficheros de texto normalmente en el disco duro de la maquina. En estos ficheros hay cierta información cuyo fin es provocar ciertas acciones en una maquina objetivo (que puede no ser la que esta interpretándolos"). Para ello, los ficheros son leídos del disco y pasados a la memoria, conformando el flujo denominado "Caracteres".

Análisis léxico: Esta fase tiene que ver con el "vocabulario" del que hablábamos más arriba. El proceso de análisis léxico agrupa los diferentes caracteres de su flujo de entrada en tokens. Los tokens son los símbolos léxicos del lenguaje; se asemejan mucho a las palabras del lenguaje natural. Los tokens están identificados con símbolos (tienen "nombres") y suelen contener información adicional (como la cadena de caracteres que los originó, el fichero en el que están y la línea donde comienzan, etc). Una vez son identificados, son transmitidos al siguiente nivel de análisis. El programa que permite realizar el análisis léxico es un analizador léxico. En inglés se le suele llamar scanner o lexer.

Para ejemplificar cómo funciona un lexer vamos a usar un ejemplo: Dado el fragmento de código imaginario siguiente, que podría servir para controlar un robot en una fábrica de coches (por simplicidad, suponemos que es el ordenador que realiza el análisis es el del propio robot, es decir, no hay compilación cruzada):

```
Apretar (tuercas);

Pintar (chasis+ruedas);
```

El analizador léxico del robot produciría la siguiente serie de tokens:

```
RES_APRETAR PARENT_AB NOMBRE PARENT_CE PUNTO_COMA

RES_PINTAR PARENT_AB NOMBRE SIM_MAS NOMBRE PARENT_CE PUNTO_COMA
```

Análisis sintáctico: En la fase de análisis sintáctico se aplican las reglas sintácticas del lenguaje analizado al flujo de tokens. En caso de no haberse detectado errores, el intérprete representará la información codificada en el código fuente en un Árbol de Sintaxis Abstracta, que no es más que una representación arbórea de los diferentes patrones sintácticos que se han encontrado al realizar el análisis, salvo que los elementos innecesarios (signos de puntuación, paréntesis) son eliminados. En adelante llamaremos AST a los Árboles de Sintaxis Abstracta. El código que permite realizar el análisis sintáctico se llama "analizador sintáctico". En inglés se le llama *parser*, que significa "iterador" o directamente *analyzer* ("analizador"). Continuando con el ejemplo anterior, el análisis léxico del robot produciría un AST como el siguiente:

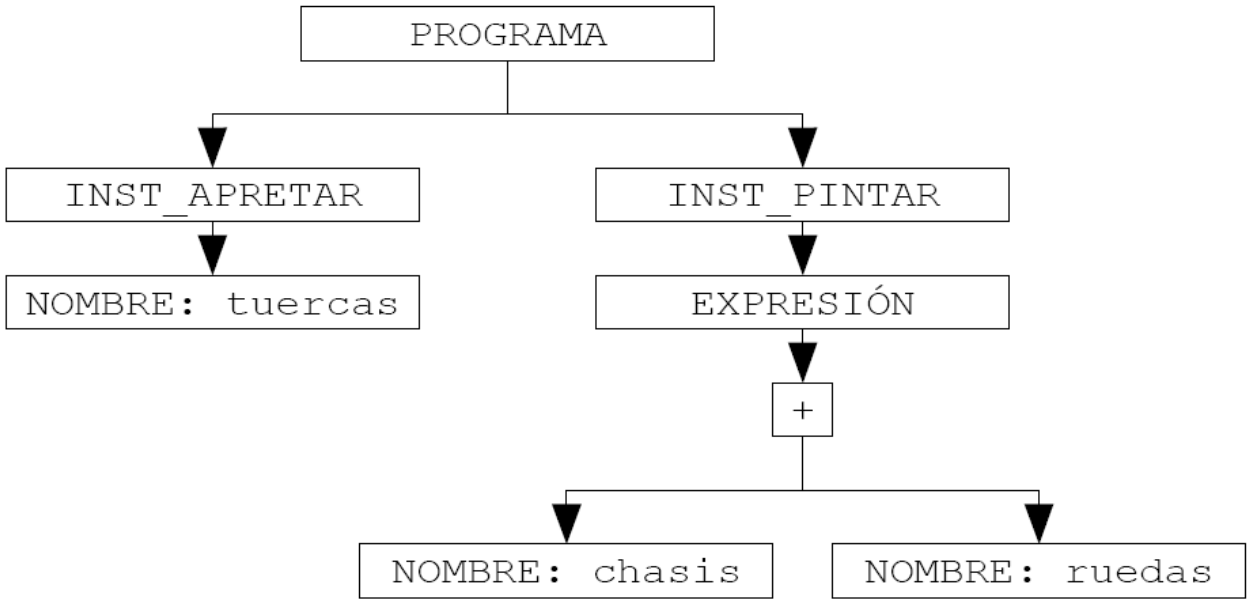


Ilustración 1.6 Estructura de un AST sencillo

El robot ha identificado dos instrucciones, una de “apretar” y otra de “pintar”. Los tokens del tipo “NOMBRE” tienen información adicional. Para operar sobre varios objetos a la vez (como sobre el chasis y las ruedas) se pueden escribir varios nombres unidos por el símbolo '+'.

Análisis semántico: El análisis semántico del árbol AST empieza por detectar incoherencias a nivel sintáctico en el AST (Árboles de Sintaxis Abstracta). Si el AST supera esta fase, es corriente enriquecerlo para realizar un nuevo análisis semántico. Es decir, es corriente efectuar varios análisis semánticos, cada uno centrado en aspectos diferentes. Durante éstos análisis el árbol es enriquecido y modificado.

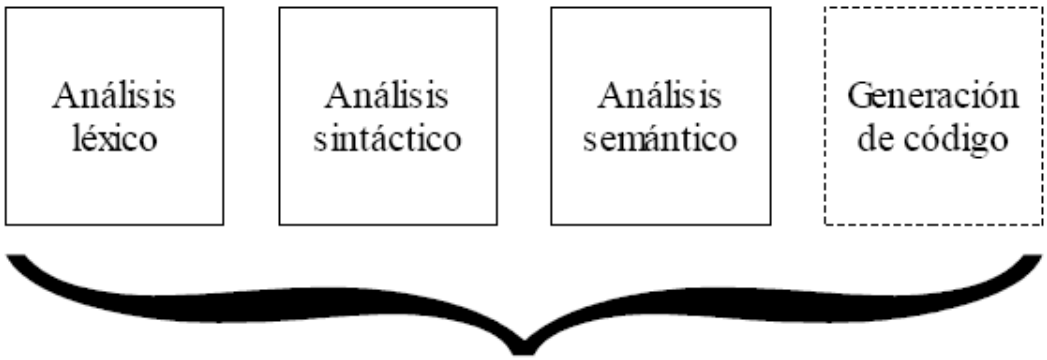
Cualquier herramienta que realice un análisis semántico será llamada “analizador semántico” en este texto. En la bibliografía inglesa suelen referirse a los analizadores semánticos como *tree parsers* (o “iteradores de árboles”). En el caso de nuestro robot, podríamos considerar que aunque el chasis de un coche se pueda pintar, las ruedas no son “pintables”. Por lo tanto se emitiría un mensaje de error (a pesar de que el código suministrado fuese gramaticalmente válido) y no continuaría el análisis. Imaginemos ahora que el operario del robot se da cuenta de su error y sustituye “ruedas” por “llantas”, que sí es reconocido por el robot como algo que se puede pintar. En tal caso, dada la simplicidad del AST, no es necesario enriquecerlo de ninguna forma, por lo que se pasaría a la siguiente fase.

Generación de código: En esta fase se utiliza el AST enriquecido, producto del proceso de análisis semántico, para generar código máquina. Nuestro robot, una vez eliminado el error sintáctico detectado, generaría el código binario necesario para apretar y pintar los elementos adecuados, pudiendo ser utilizado en el futuro. Dicho código (en ensamblador-para-robot) sería parecido al siguiente:

```
mientras queden tuercas sin apretar

busca tuerca sin apretar.

aprieta tuerca.

sumergir chasis en pozo de pintura.

colocar llanta1 bajo ducha de sulfato. Esperar. Despejar ducha.

colocar llanta2 bajo ducha de sulfato. Esperar. Despejar ducha.

colocar llanta3 bajo ducha de sulfato. Esperar. Despejar ducha.

colocar llanta4 bajo ducha de sulfato. Esperar. Despejar ducha.

Desconectar.
```

¡Recordemos que el código anterior está codificado en binario, todo generado a partir de dos líneas de código fuente! El compilador es un "puente" entre el operario y el robot, que genera por el ser humano todo el código repetitivo en binario insertando cuando es necesario elementos auxiliares (como la orden de desconexión, que el operario podría olvidarse de introducir). Como vemos el uso de los compiladores aumenta la productividad. Nos vamos a centrar en las tres primeras fases del análisis, que son las cubiertas por ANTLR de una manera novedosa con respecto a los enfoques anteriores. Una vez realizado el análisis semántico, la generación de código no se diferenciará mucho de cómo se haría antiguamente.

Finalmente ANTLR es capaz de actuar a *tres* niveles a la vez (cuatro si tenemos en cuenta la generación de código):



Comparación de las herramientas

El uso de una sola herramienta para todos los niveles tiene varias ventajas. La más importante es la "estandarización": con ANTLR basta con comprender el paradigma de análisis una vez para poder implementar todas las fases de análisis.

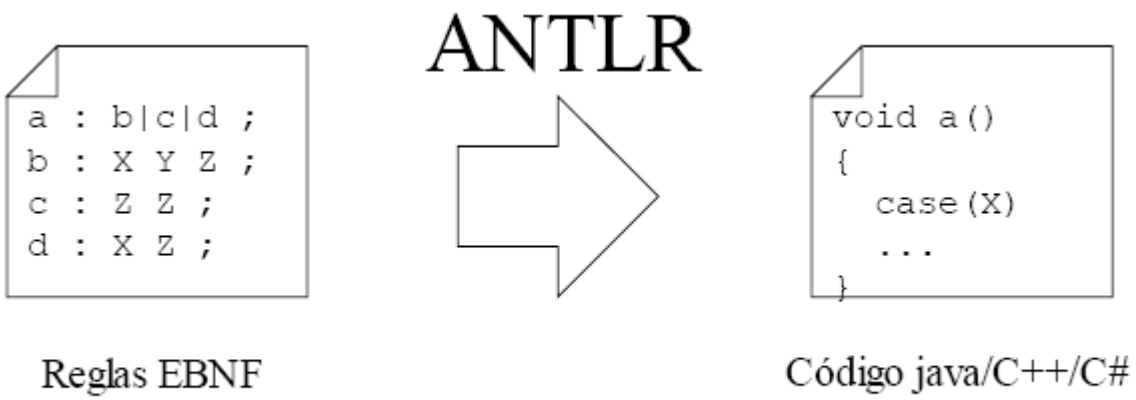
Presentación del ANTLR:

Definición y como funciona: ANTLR es un programa está escrito en java, por lo que se necesita alguna máquina virtual de java para poder ejecutarlo. Es software libre, lo que quiere decir que al descargarlo de la página oficial (**http://www.antlr.org**) obtendremos tanto los ficheros compilados *.class como el código fuente en forma de ficheros *.java.

ANTLR es un generador de analizadores. Mucha gente llama a estas herramientas compiladores de compiladores, dado que ayudar a implementar compiladores es su uso más popular. Sin embargo tienen otros usos. ANTLR, por ejemplo, podría servir para implementar el intérprete de un fichero de configuración.

ANTLR es capaz de generar un analizador léxico, sintáctico o semántico en varios lenguajes (java, C++ y C# en su versión 2.7.2) a partir de unos ficheros escritos en un lenguaje propio.

Dicho lenguaje es básicamente una serie de reglas EBNF y un conjunto de construcciones auxiliares.



Funcionamiento de ANTLR

Ejemplos de uso del ANTLR:

Calculadora simbólica(Versión 1):

El propósito de este ejemplo es ilustrar el funcionamiento del ANTLR implementando un intérprete para un lenguaje muy sencillo.

La calculadora será capaz de realizar operaciones aritméticas simples (suma y resta) con números enteros o con decimales, e imprimirá el resultado por pantalla. Los cálculos los introducirá el usuario por la consola, y la calculadora los presentara por pantalla.

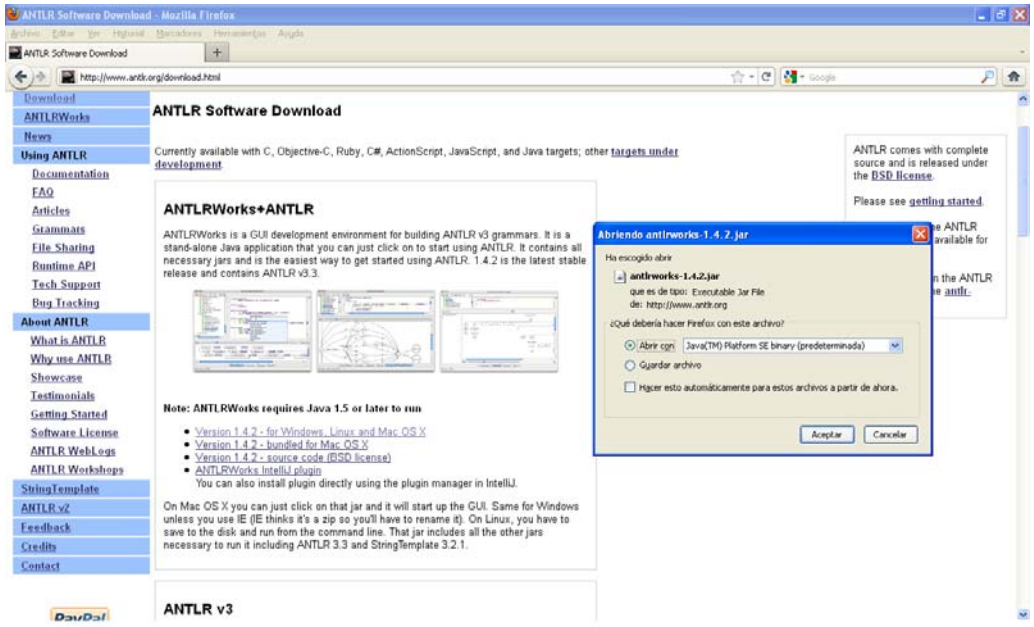
Para llevarlo a cabo es necesario construir los analizadores que son el analizador **léxico**, **sintáctico** y **semántico** donde el funcionamiento de cada analizador estará basado en el uso de reglas **EBNF**.

Las reglas **EBNF** no es ni más ni menos que un lenguaje especial para definir las gramáticas. Una gramática es un conjunto de *reglas*. Toda regla comienza con un *nombre* (o "parte izquierda") seguido por el carácter de los dos puntos ":". A continuación aparece el *cuerpo* (o "parte derecha") de la regla. Todas las reglas terminan con el carácter de punto y coma ";".

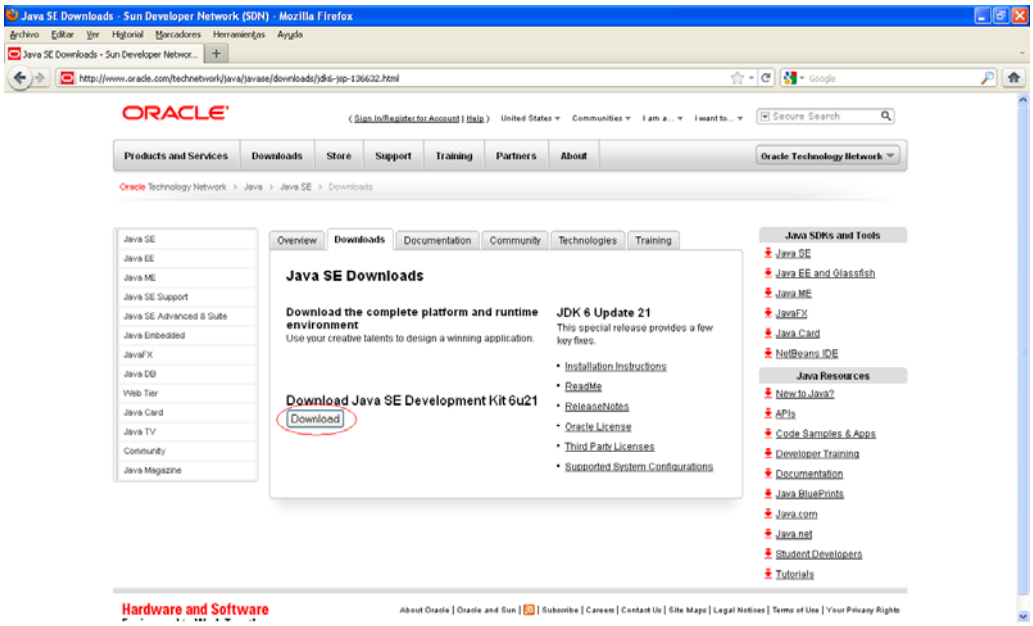
Volviendo con los analizadores estos pueden ser construidos en lenguaje **java** o en otros lenguajes como ser **C#** (Csharp) (dependiendo de ciertas opciones que se especifican en la edición del archivo de gramática) en forma de **clases** (ver programación orientada a objetos). Es decir por cada analizador descripto en el archivo de gramática se generara una clase.

Para la construcción de los analizadores deberemos ingresar al programa **ANTLR** y crear un archivo con la extensión ***.g** este tipo de archivo se denomina **archivo de gramática**; allí contendrá la definición de todos los analizadores (léxico, sintáctico y semántico).

Este ejemplo y los siguientes están basados en la versión **3.0** del ANTLR por consiguiente antes de comenzar con el desarrollo en sí de la calculadora se debe bajar de la siguiente pagina www.antlr.org/download.html un archivo ejecutable realizado en java llamado **antlrworks-1.4.2.jar** haciendo clic en el siguiente link (**Versión 1.4.2 – for Windows Linux and Mac Os X)** como se muestra en la imagen siguiente:

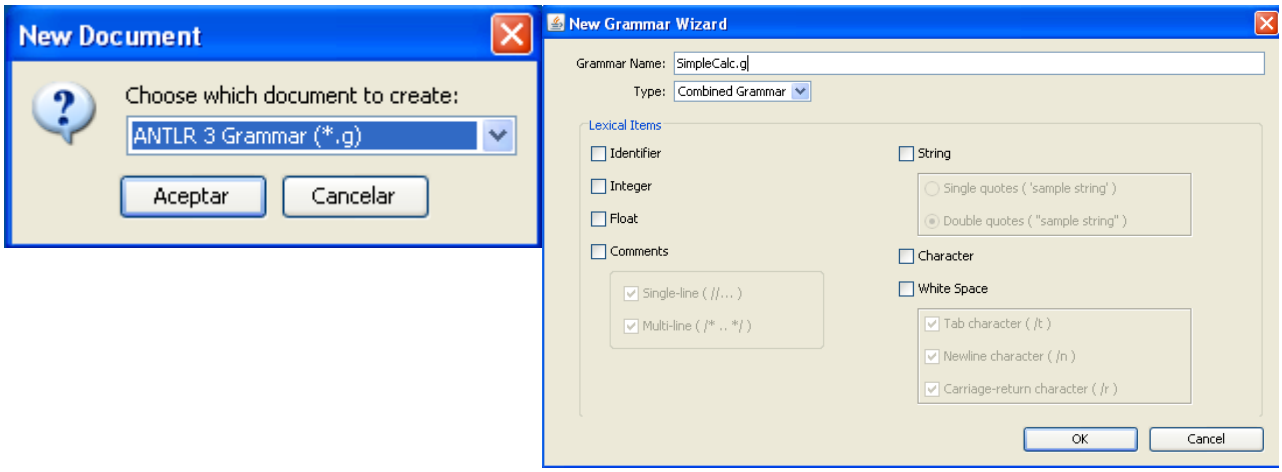


Luego del paso anterior además se deberá descargar un paquete java **necesario para la compilación de nuestro archivo de gramática escritos en lenguaje Java** de la siguiente dirección: http://www.oracle.com/technetwork/java/javase/downloads/jdk6-jsp-136632.html como muestra la siguiente imagen; luego se deberá instalar en nuestra computadora.

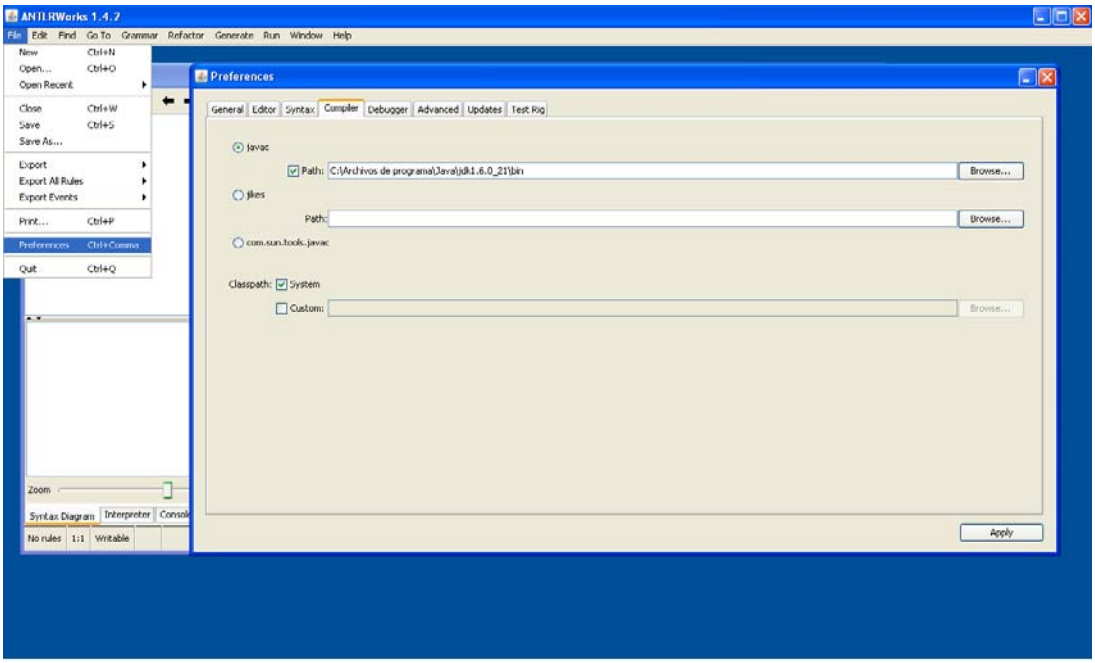


Realizados los pasos previos mencionados estamos en condiciones de empezar con nuestra calculadora.

Comenzamos primero creando una carpeta llamada SimpleCalc allí copiaremos el ejecutable de java que descargamos de la web llamado **antlrworks-1.4.2.jar.** Luego lo ejecutamos y nos aparecerá una ventana para crear nuestro archivo gramática; a continuación ingresamos el nombre SimpleCalc.g como se detallan en las imágenes siguientes:



Otro paso importante es configurar el ANTLR con el paquete Java descargado de la pagina oracle; dentro del entorno del ANTLRWorks 1.4.2 se deberá ir a la **solapa File → Preferences** allí se deberá seleccionar la opción javac y indicarle la ruta donde se instalo el paquete java como muestra la imagen siguiente:



Comenzamos ahora con la estructura del archivo gramática SimpleCalc.g

Se comienza primero con la cabecera del fichero. Aquí se debe prestar atención de darle el mismo nombre que tiene el archivo. Dado que la clase debe llamarse igual que el nombre del fichero, sino, no compilará.

```
grammar SimpleCalc;
```

El siguiente paso es la declaración de tokens. Aquí debemos añadir todos los elementos que queremos que reconozca nuestra gramática. En nuestro caso el signo + y el signo –.

```
tokens {

PLUS = '+';

MINUS = '- ';

}
```

La siguiente parte es @ members contiene el código java de nuestra aplicación. En nuestro caso hemos colocado un main. De hecho el programa funcionará igual sin este código el problema es que no podremos llegar a ver nada si el texto de entrada es aceptado. Lo que hace este código es leer un fichero de texto especificado en la línea de comandos cuando arrancamos nuestro compilador, eso es args[0]. Dentro del bloque try, llamamos a la primera regla del programa (parser.expr()), esto desencadenará el inicio del programa.

```
@members {

public static void main(String[] args) throws Exception {

  SimpleCalcLexer lex = new SimpleCalcLexer(new ANTLRFileStream(args[0]));

    CommonTokenStream tokens = new CommonTokenStream(lex);

     SimpleCalcParser parser = new SimpleCalcParser(tokens);

     try {

        parser.expr();

     } catch (RecognitionException e) {

          e.printStackTrace();

         }

 }

}
```

El siguiente bloque es las reglas del parser, es decir puramente la gramática. Aquí podemos ver como una expresión es un ID (identificador, o nombre de variable) el token igual ´=´ (si, ya sé que lo podríamos haber definido en el apartado de tokens, pero para que vierais que también podemos ponerlo directamente aquí) y una operación (la definimos más abajo). Lo que vemos entre { } es el código java que he escrito yo. Básicamente nos imprimirá por pantalla las expresiones que vaya reconociendo.

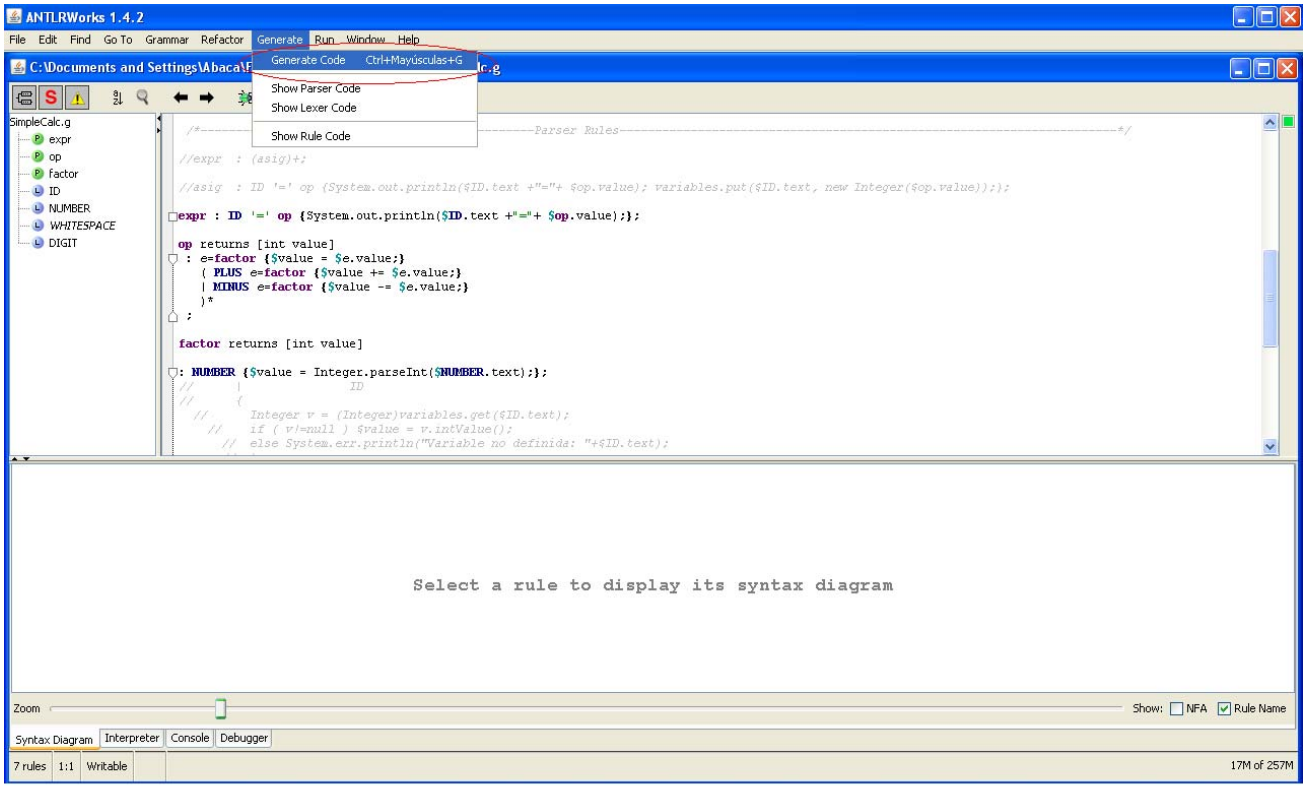
La siguiente regla es la operación. Tenemos un factor OPERANDO factor. El factor en nuestro caso será un entero, lo podemos ver en la tercera regla. Lo que hace es almacenar en una variable valor temporal, el resultado de sumar o restar los dos factores. La tercera regla, almacena en una variable llamada value el resultado de parsear la lectura del fichero a un Entero.

```
/*------------------------------------------------------------------

* PARSER RULES

*------------------------------------------------------------------*/

expr : ID '=' op {System.out.println($ID.text +"="+ $op.value);};

op returns [int value]

: e=factor {$value = $e.value;}

( PLUS e=factor {$value += $e.value;}

| MINUS e=factor {$value -= $e.value;}

)*

;

factor returns [int value]

: NUMBER {$value = Integer.parseInt($NUMBER.text);};
```

La ultima parte de nuestro archivo gramática es la declaración de los tipos en nuestro caso un ID (identificador de variable) es cualquier nombre de la a-Z que contenga al menos una letra. Esto es debido a la cláusala positiva de Klenee (+). El mismo razonamiento para los números, para los espacios en blanco y los dígitos. Los campos disponen de varios valores (como si fueran una struct) estos son .value y .text. Se entiende que value es un entero y text un campo para almacenar strings, como los ID.

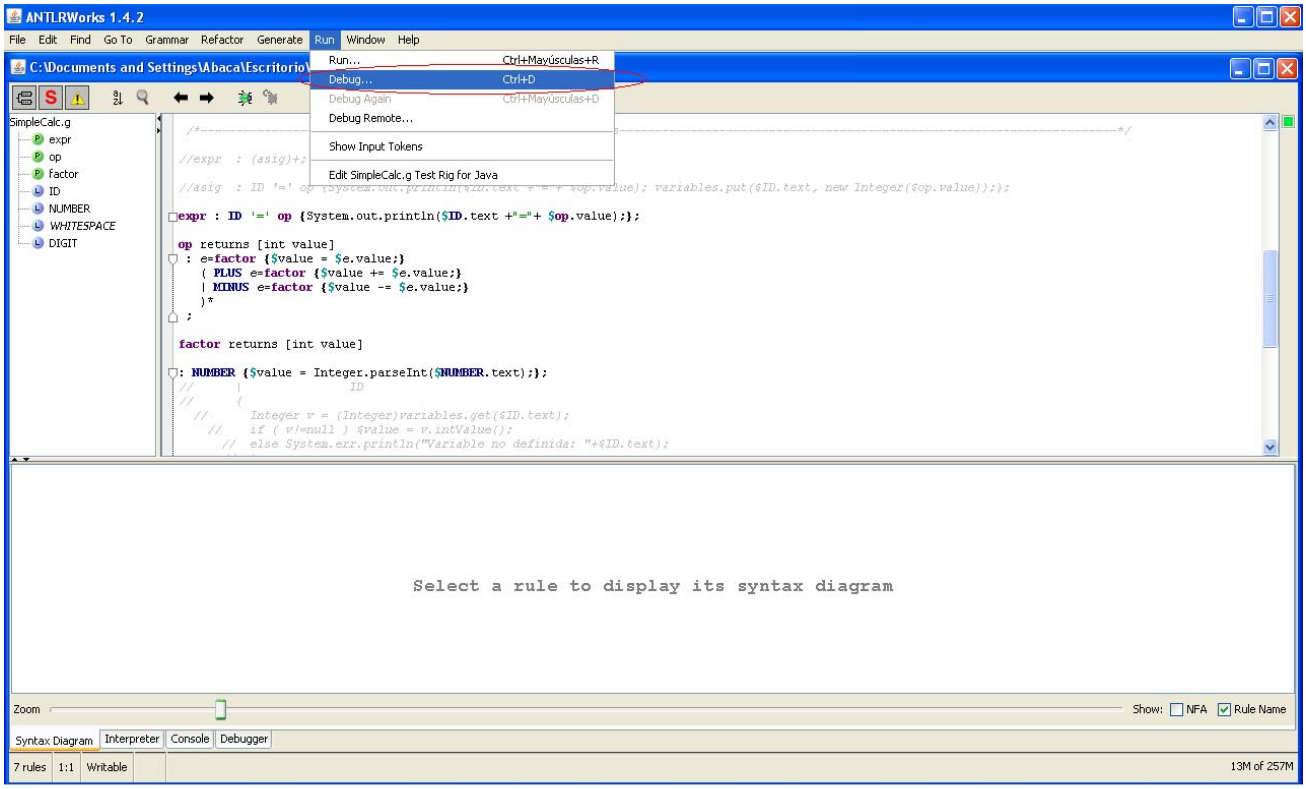
```
/*------------------------------------------------------------------

* LEXER RULES

*------------------------------------------------------------------*/

ID : ('a'..'z'|'A'..'Z')+ ;

NUMBER : (DIGIT)+ ;

WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+ { $channel = HIDDEN; } ;

fragment DIGIT : '0'..'9';
```

Luego de finalizado nuestro archivo gramática se procede a **generar el código y a compilar** para ello se debe ir a la solapa **Generate** se indica en la siguiente imagen:

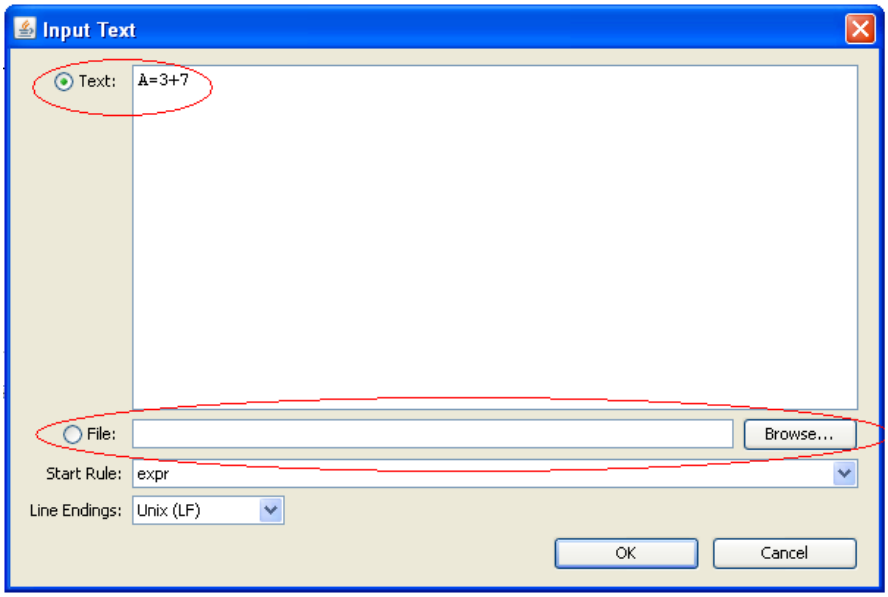


Si todo esta correcto esto nos creara dos archivos java **SimpleCalcLexer.java y SimpleCalcParser.java** dentro de una carpeta llamada output esta misma esta dentro de nuestra carpeta de trabajo **SimpleCalc**.

El siguiente paso es compilar nuestro programa para ello nos dirigimos a la solapa Run →Debug como muestra la siguiente imagen:

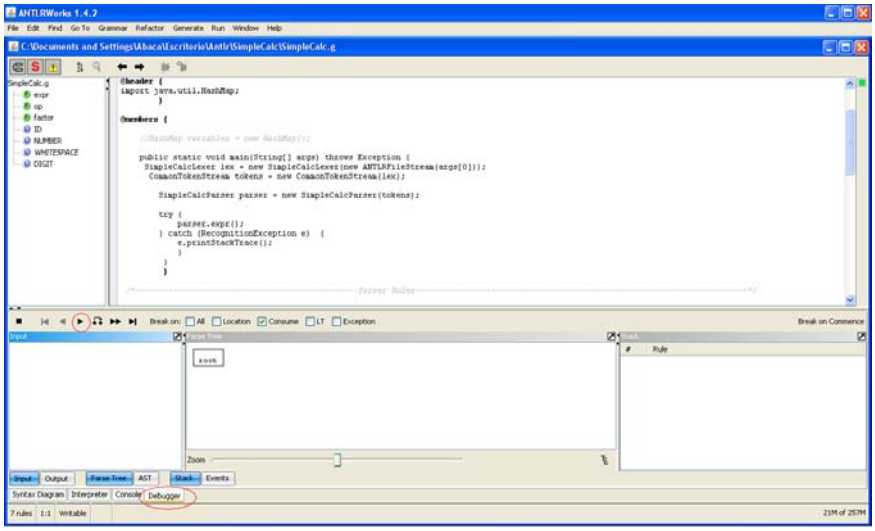


Si todo ello funciona correctamente, como debería, nos generará dos ficheros class con el mismo nombre que los archivos de java dentro de la carpeta SimpleCalc/output/classes y nos debería aparecer la siguiente ventana:

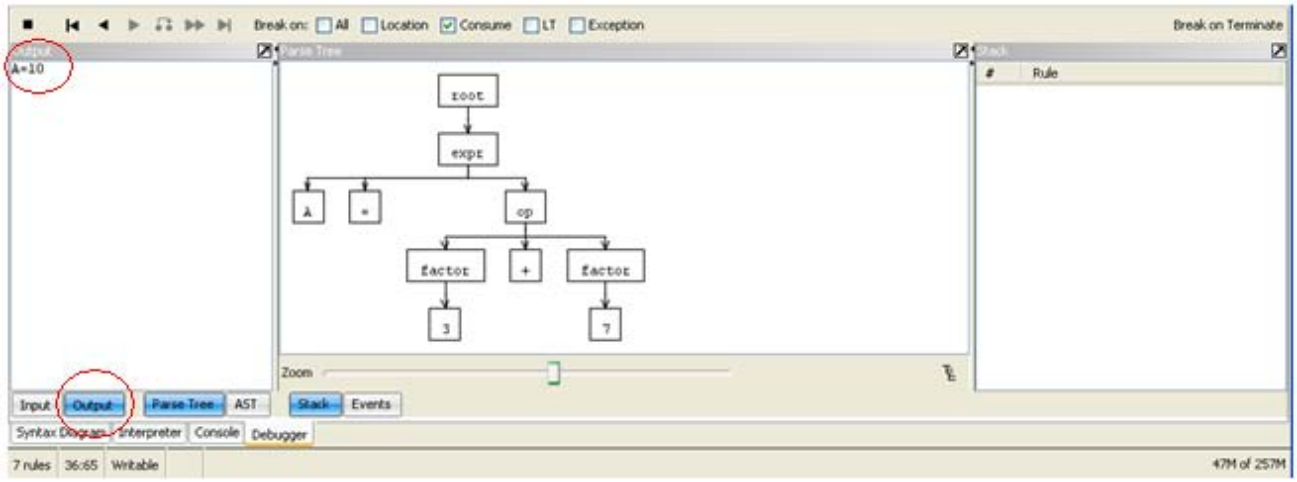


Aquí podremos **ingresar nuestro código fuente** que hemos definido en el archivo gramática a través de la consola seleccionando **Text** o a través de un archivo de texto seleccionando la opción **File**.

Luego de hacer clic en OK en la solapa Debugger podremos comprobar el correcto funcionamiento de nuestro compilador haciendo clic en el símbolo play y ver así paso por paso la ejecución del compilador hasta llegar al resultado.



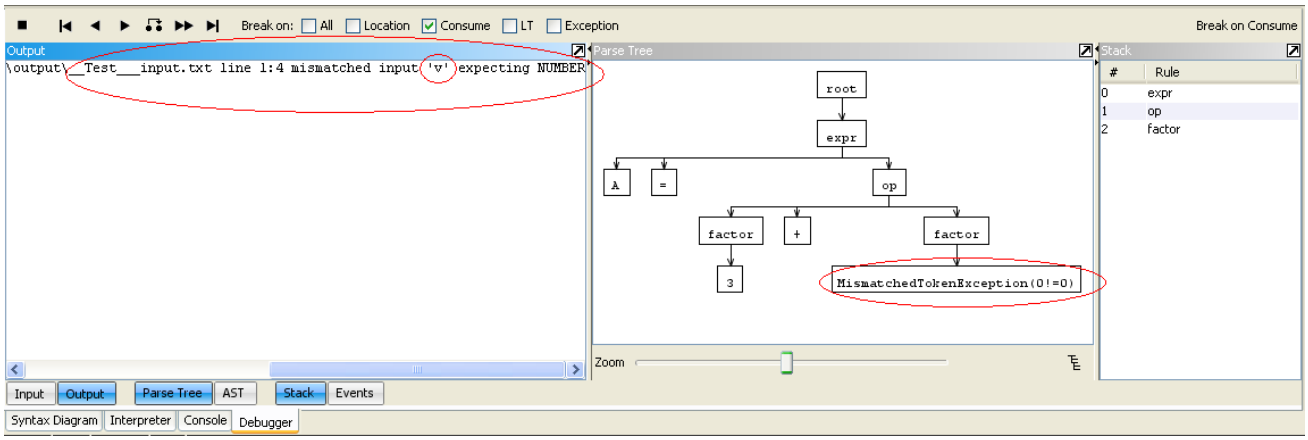
Una vez que se ejecutaron todos los pasos de nuestro compilador si ingresamos A = 3+7 en la solapa llamada Output debería mostrarnos el resultado de la suma en este caso A=10 como se detalla a continuación en la siguiente imagen:



Ahora que vemos que funciona nuestro compilador; vamos a probar si es capaz de detectar errores en el código fuente que le ingresamos por ejemplo si cambiamos nuestra asignación por esta otra:

A=3+**v**

Rápidamente nuestro compilador nos dará un error como se muestra en la siguiente imagen:



Como podemos ver se queja de que no reconoce el token **v** en esa expresión. Es decir un error de missmatch o sea que no coincide lo que espera con lo que le llega. Obvio ya que hemos definido las sumas como sumas de enteros y v no es ningún entero, ya que es un ID (Ver en el código "Lexer Rules".

Calculadora simbólica (Versión 2):

La calculadora simbólica del punto anterior tiene el inconveniente de que no puede realizar suma de variables dado que arroja un error de missmatch. Además solo imprime el valor del resultado pero no lo almacena en ningún registro que es necesario si se quiere realizar suma de variables.

Para ello vamos a modificar la gramática para que nos admita más de una asignación. Si se observa el código anterior la regla expr, nuestra primera regla solo reconoce una expresión. Para ello se cambiara el nombre de expr por asig y antes de esta regla se añadirá una nueva regla llamada expr.

```
expr    : (asig)+;

asig    : ID '=' op {System.out.println($ID.text +"="+ $op.value);};
```

Con esto se logra reconocer un conjunto de varias asignaciones.

Para guardar las variables usaremos una tabla Hash. Java ya nos proporciona una estructura de datos de este tipo. Se llama HashMap.

En la directiva @members debemos añadir nuestra estructura. Quedaría algo así:

```
HashMap variables = new HashMap();
```

Hemos de importar ahora la clase HashMap. Para esto antlr dispone de un nuevo espacio para indicar los imports de clases java. Añadiremos el siguiente código justo antes de la directiva

```
@members
@header {
import java.util.HashMap;
}
```

Con este cambio, solo nos faltará modificar las reglas de la gramática. En concreto, la regla asig, aparte de que nos muestre la asignación aceptada, nos guarda el resultado en la tabla de variables. Dentro del código java entre llaves, añadimos el siguiente:

```
asig : ID '=' op {System.out.println($ID.text +"="+ $op.value);
variables.put($ID.text, new Integer($op.value));};
```

Se ha añadido entonces una instrucción para que nos añada en la tabla la variable, con su valor entero.

El siguiente paso es modificar la regla factor para que nos permita operar también con variables (ID).

```
factor returns [int value]
: NUMBER {$value = Integer.parseInt($NUMBER.text);}
| ID
{
Integer v = (Integer)variables.get($ID.text);
if ( v!=null ) $value = v.intValue();
else System.err.println("Variable no definida: "+$ID.text);
};
```

Se puede observar que hay una nueva regla con una disyunción (|). Esta regla busca la variable ID, en la tabla de variables, la parsea a un entero. Si no la encuentra imprime un mensaje de error "Variable no definida".

Luego de finalizado nuestro archivo gramática se procede como el ejemplo anterior a **generar el código y a compilar** para ello se debe ir a la solapa **Generate.** Si todo esta correcto esto nos creara dos archivos java **SimpleCalcLexer.java y SimpleCalcParser.java** dentro de una carpeta llamada output esta misma esta dentro de nuestra carpeta de trabajo **SimpleCalc**.

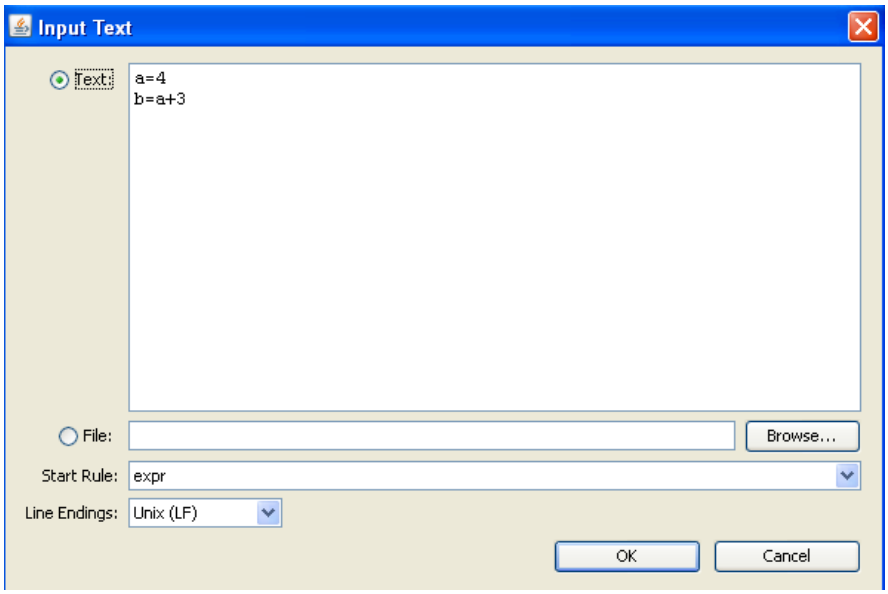
El siguiente paso es compilar nuestro programa para ello nos dirigimos a la solapa Run →Debug. Si todo ello funciona correctamente, como debería, nos generará dos ficheros class con el mismo

nombre que los archivos de java dentro de la carpeta SimpleCalc/output/clases y estaríamos listos para probar nuestro compilador.

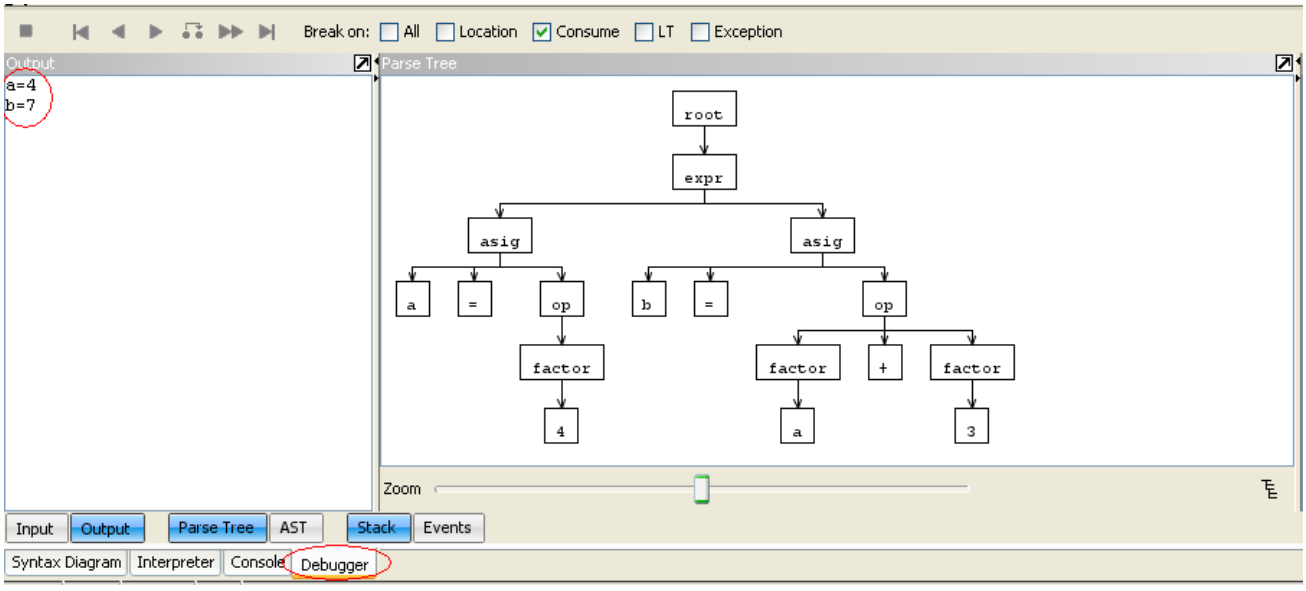
Si se ingresa lo siguiente en Input Text:

a = 4

b = a+3



Luego de hacer clic en OK e ir a la solapa Debugger llegando hasta el paso final se debería obtener lo siguiente:

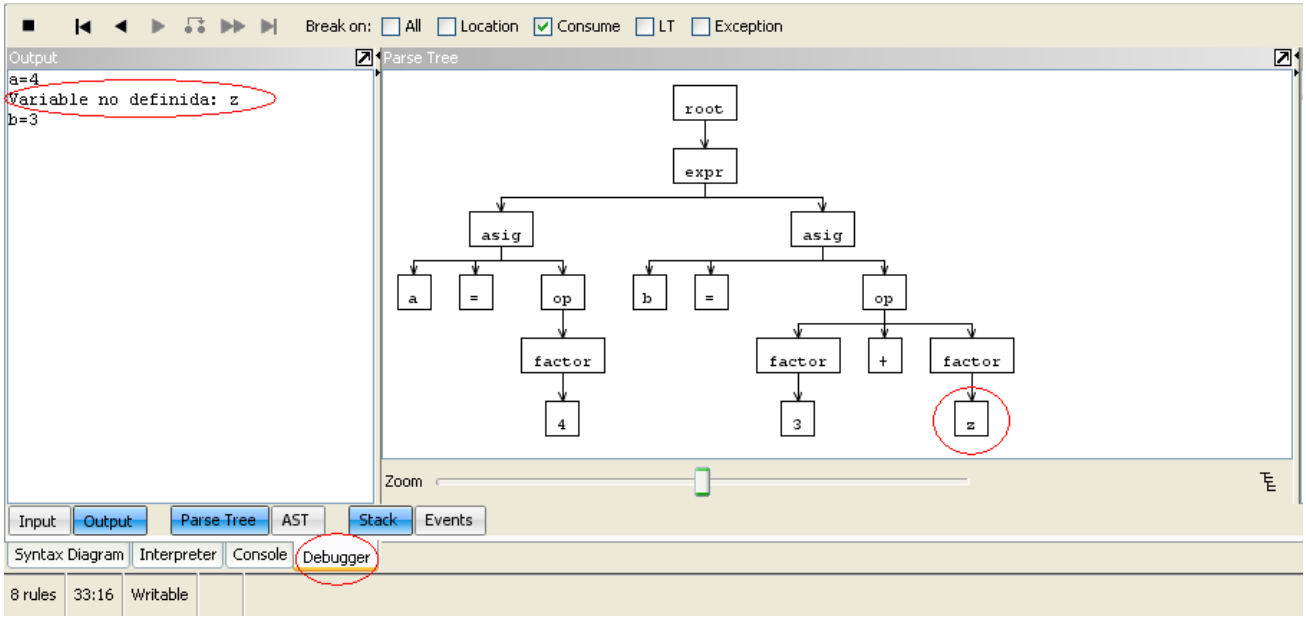


Se obtiene un resultado correcto dado que a = 4 y b = a + 3 entonces b = 4 + 3 = 7. Se comprueba entonces que ahora nuestro compilador es capaz de comprender operaciones con variables.

Acto seguido vamos a probar si es capaz de detectar errores en el código fuente que le ingresamos por ejemplo si cambiamos nuestra asignación por esta otra:

a = 4

b = 3 + **z**

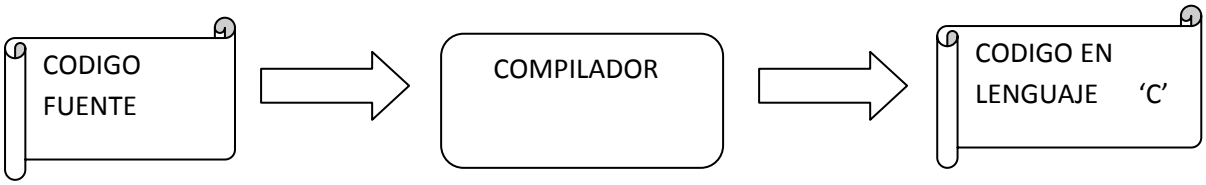


Se puede observar como nuestro compilador es capaz de detectar variables no definidas como en este caso es **z**.

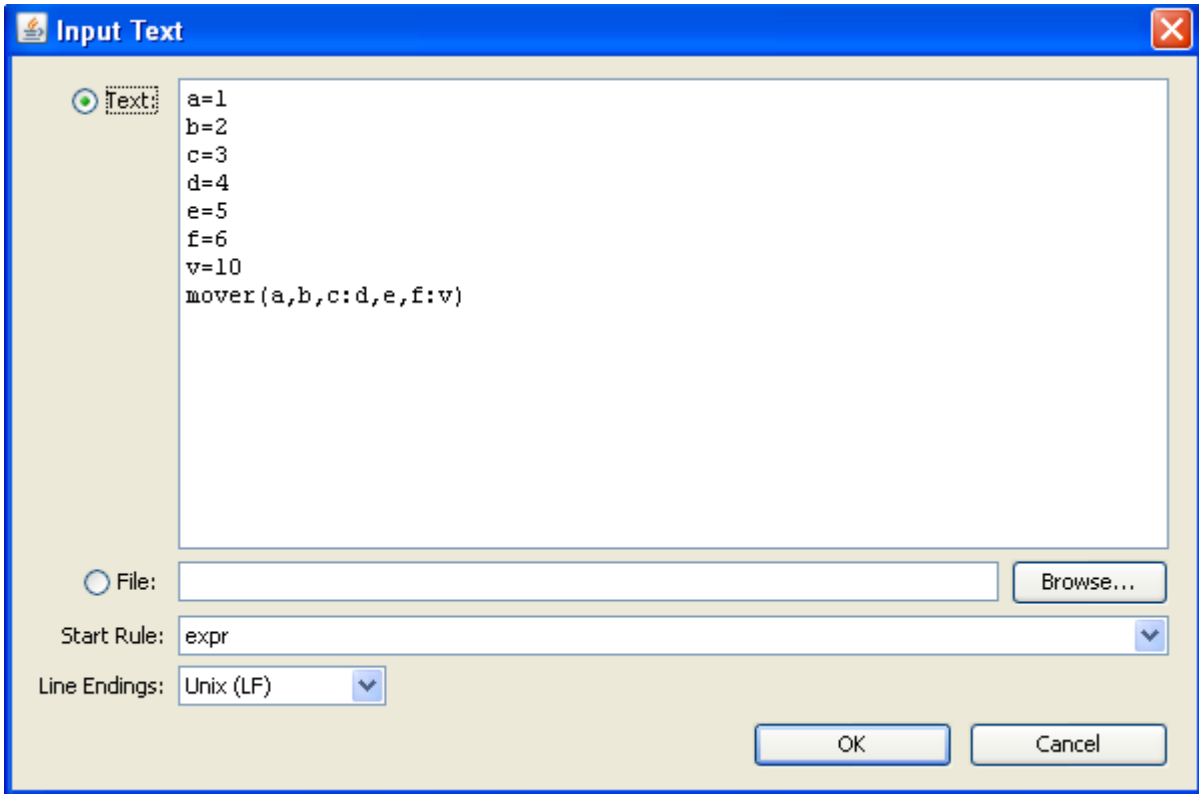
Compilador para nuestro robot M5:

En este ejemplo el objetivo será que a partir de nuestro propio lenguaje (código fuente) sea traducido y o analizado para lograr otro lenguaje en este caso ‘C’ a través de nuestro compilador haciendo uso de sus analizadores (léxico, sintáctico y semántico).

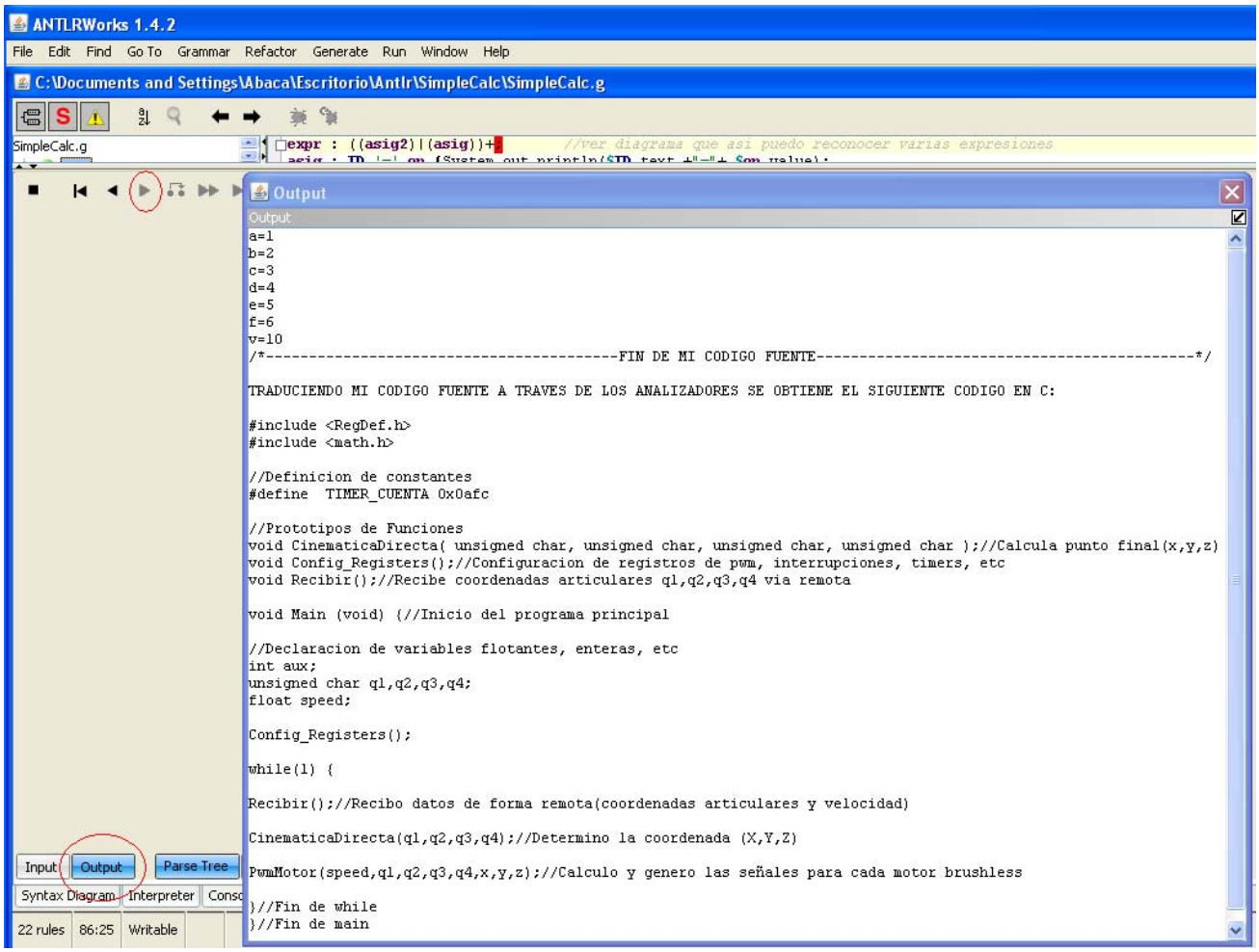
En síntesis nuestro objetivo es el siguiente:



Una vez construido nuestro archivo gramática en el ANTLR y realizados los pasos de generar código y compilar como los ejemplos anteriores; a continuación se procederá a cargar en la consola Input Text mi lenguaje propio (código fuente) a través de un archivo de texto o simplemente escribiendo sobre la consola como se muestra en la imagen siguiente.



Luego de hacer clic en OK como ya sabemos debemos ir a la solapa Debugger y hacer varias veces clic en el simbolito 'play' eso ejecutara nuestro compilador paso a paso que como resultado obtendremos el siguiente código en 'C' para nuestro robot M5. Los resultados se muestran en la siguiente imagen:



Nota: Este código es a modo de ejemplo y esta realizado para 4 grados de libertad.

A continuación se muestra el código del archivo gramática:

```
grammar SimpleCalc;

tokens

{

     //Acá definimos los símbolos que reconoce el compilador

     ALIAS          = 'ALIAS';

     AND            = 'AND';

     ARRAY          = 'ARRAY';

     ASSOCIATIVE    = 'ASSOCIATIVE';

     BEGIN          = 'BEGIN';

     BINDINGS       = 'BINDINGS';
```

```
BY             = 'BY';
CASE           = 'CASE';
CONST          = 'CONST';
DEFINITION     = 'DEFINITION';
DIV            = 'DIV';
DO             = 'DO';
ELSE           = 'ELSE';
ELSIF          = 'ELSIF';
END            = 'END';
EXIT           = 'EXIT';
FOR            = 'FOR';
FROM           = 'FROM';
IF             = 'IF';
IMPLEMENTATION = 'IMPLEMENTATION';
IMPORT         = 'IMPORT';
IN             = 'IN';
LOOP           = 'LOOP';
MINUS          = '-' ;
MOD            = 'MOD';
MODULE         = 'MODULE';
NOT            = 'NOT';
OF             = 'OF';
OPAQUE         = 'OPAQUE';
OR             = 'OR';
PLUS        = '+' ;
POINTER        = 'POINTER';
PROCEDURE      = 'PROCEDURE';
PRODUCT        = '*';
RECORD         = 'RECORD';
```

```
	REPEAT		= 'REPEAT';
	RETURN		= 'RETURN';
	SET		= 'SET';
	THEN		= 'THEN';
	TO		= 'TO';
	TYPE		= 'TYPE';
	UNTIL		= 'UNTIL';
	VAR		= 'VAR';
	VARIADIC	= 'VARIADIC';
	WHILE		= 'WHILE';
	MOVE	= 'mover';
}
@header
{
	import java.util.HashMap;  //Importamos la clase de la tabla Java
para almacenar variables
	//package com.ociweb.math;
}
@members
{
Integer i=0;
HashMap variables = new HashMap();//Tabla de java para almacenar
variables
HashMap valores = new HashMap();
public static void main(String[] args) throws Exception
	{
SimpleCalcLexer lex = new SimpleCalcLexer(new ANTLRFileStream(args[0]));
CommonTokenStream tokens = new CommonTokenStream(lex);
SimpleCalcParser parser = new SimpleCalcParser(tokens);
```

```
try

{

parser.expr();

}

catch (RecognitionException e)

{

e.printStackTrace();

}

     }

}

/*------------------------------------------------------------------

* PARSER RULES

*------------------------------------------------------------------*/

expr : ((asig2)|(asig))+;//ver diagrama que asi puedo reconocer varias
expresiones

asig : ID '=' op {System.out.println($ID.text +"="+ $op.value);

variables.put($ID.text, new Integer($op.value));};  //para debug

op returns [int value]

:

 e=factor {$value = $e.value;}

(PLUS e=factor {$value += $e.value;}

| MINUS e=factor {$value -= $e.value;}

| PRODUCT e=factor {$value *= $e.value;}

)*

;

factor returns [int value]

:

  NUMBER {$value = Integer.parseInt($NUMBER.text);}//Con esto acepta un
entero
```

```
| ID

{

Integer v = (Integer)variables.get($ID.text);//Con esto acepta letras
como enteros

valores.put(i++, new Integer(v));

//System.out.println("valor1="+(Integer)valores.get(i));

if ( v!=null ) $value = v.intValue();

else System.err.println("Variable no definida: "+$ID.text);

};

asig2: MOVE '(' factor COMA factor COMA factor ':' factor COMA factor
COMA factor ':' factor ')'

{

System.out.println("/*----------------------------------------FIN DE MI
CODIGO FUENTE-----------------------------------------*/");

System.out.println("");

System.out.println("TRADUCIENDO MI CODIGO FUENTE A TRAVES DE LOS
ANALIZADORES SE OBTIENE EL SIGUIENTE CODIGO EN C:");

System.out.println("");

System.out.println("#include <RegDef.h>");

System.out.println("#include <math.h>");

System.out.println("");

System.out.println("//Definicion de constantes");

System.out.println("#define  TIMER_CUENTA 0x0afc ");

System.out.println("");

System.out.println("//Prototipos de Funciones ");

System.out.println("void CinematicaDirecta( unsigned char, unsigned char,
unsigned char, unsigned char );//Calcula punto final(x,y,z)");

System.out.println("void Config_Registers();//Configuracion de registros
de pwm, interrupciones, timers, etc");

System.out.println("void Recibir();//Recibe coordenadas articulares
q1,q2,q3,q4 via remota");

System.out.println("");
```

```
System.out.println("void Main (void) {//Inicio del programa principal ");

System.out.println("");

System.out.println("//Declaracion de variables flotantes, enteras, etc");

System.out.println("int aux;");

System.out.println("unsigned char q1,q2,q3,q4;");

System.out.println("float speed;");

System.out.println("");

System.out.println("Config_Registers();");

System.out.println("");

System.out.println("while(1) {");

System.out.println("");

System.out.println("Recibir();//Recibo datos de forma remota(coordenadas
articulares y velocidad)");

System.out.println("");

System.out.println("CinematicaDirecta(q1,q2,q3,q4);//Determino la
coordenada (X,Y,Z)");

System.out.println("");

System.out.println("PwmMotor(speed,q1,q2,q3,q4,x,y,z);//Calculo y genero
las señales para cada motor brushless");

System.out.println("");

System.out.println("}//Fin de while");

System.out.println("}//Fin de main");

};

/*------------------------------------------------------------------

* LEXER RULES

*------------------------------------------------------------------*/

ID : ('a'..'z'|'A'..'Z')+ ;

WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+ { $channel = HIDDEN;
} ;

LEFT_PAREN: '(';
```

```
//LIST: 'list';

//PRINT: 'print';

RIGHT_PAREN: ')';

COMA :    ',';

//VARIABLES: 'variables'; // for list command

//SIGN: '+' | '-';

NUMBER: FLOAT|INTEGER;

fragment FLOAT:INTEGER '.' '0'..'9'+;

fragment INTEGER: '0' | '1'..'9' '0'..'9'*;

//fragment INTEGER: '0' | SIGN? '1'..'9' '0'..'9'*;

NAME: LETTER (LETTER | DIGIT | '_')*;

STRING_LITERAL: '"' NONCONTROL_CHAR* '"';

fragment NONCONTROL_CHAR: LETTER | DIGIT | SYMBOL | SPACE;

fragment LETTER: LOWER | UPPER;

fragment LOWER: 'a'..'z';

fragment UPPER: 'A'..'Z';

fragment DIGIT: '0'..'9';

fragment SPACE: ' ' | '\t';

fragment SYMBOL: '!' | '#'..'/' | ':'..'@' | '['..'`' | '{'..'~';
```

Finalmente así logramos crear nuestro propio lenguaje y que sea analizado y traducido para nuestro robot M5.

Anexo:

A continuación **se desarrolla de forma completa un compilador para los que estén interesados en realizar los analizadores en otro lenguaje que no sea Java y hacerlo en Csharp C#**.

Introducción al compilador:

Para encarar este compilador ayudará mucho conocer al menos unos principios básicos de construcción de compiladores. En cuanto a ANTLR v3, en este manual no se describen sus características generales por lo que se recomienda consultar documentación adicional si aún no se ha tenido ningún contacto con la herramienta. Serán de mucha utilidad conocimientos previos sobre otras herramientas de generación de compiladores (como Flex/Bison, Lex/Yacc, JavaCC...), ya que aunque sus principios no coincidan con los de ANTLR, sí que compartirán muchos conceptos comunes.

¿Qué pretendemos construir? Los requerimientos a grandes rasgos del lenguaje que pretendemos implementar durante esta guía son los siguientes:

· El programa principal se escribirá en un solo fichero y estará formado por una sola función principal, es decir, no será necesario la implementación de llamadas a funciones internas.

· Las variables del lenguaje tendrán un tipo declarado de forma explícita y se deberá comprobar en tiempo de compilación que han sido declaradas y que sus tipos concuerdan dentro de una expresión o una asignación.

· Deberán existir los siguientes tipos de datos: entero, real, lógico y cadena.

· El lenguaje deberá proporcionar las instrucciones clásicas: asignaciones, condicionales y bucles.

· El lenguaje deberá proporcionar las expresiones aritméticas y lógicas básicas.

El sistema se escribirá completamente en C# y estará formado por un compilador que transformará el código script a un lenguaje intermedio.

ANTLR será usado para construir el compilador, que estará formado a su vez por los analizadores léxico y sintáctico, un analizador semántico para el cálculo y comprobación de tipos, y un generador de código a partir del *árbol de sintaxis abstracta* (AST) construido durante las fases anteriores.

Como ventajas adicionales que diferencian a ANTLR de otras herramientas similares podemos citar la posibilidad de generar el código de salida en diferentes lenguajes como Java, C, C++, C# o Python, y el hecho de disponer de un entorno de desarrollo propio llamado ANTLRWorks que nos permitirá construir de una forma bastante amigable las gramáticas de entrada a la herramienta, proporcionando representaciones gráficas de las expresiones y árboles generados, e incluyendo un intérprete y depurador propio.

Como recursos para empezar a conocer esta herramienta recomiendo los siguientes:

Web principal de ANTLR

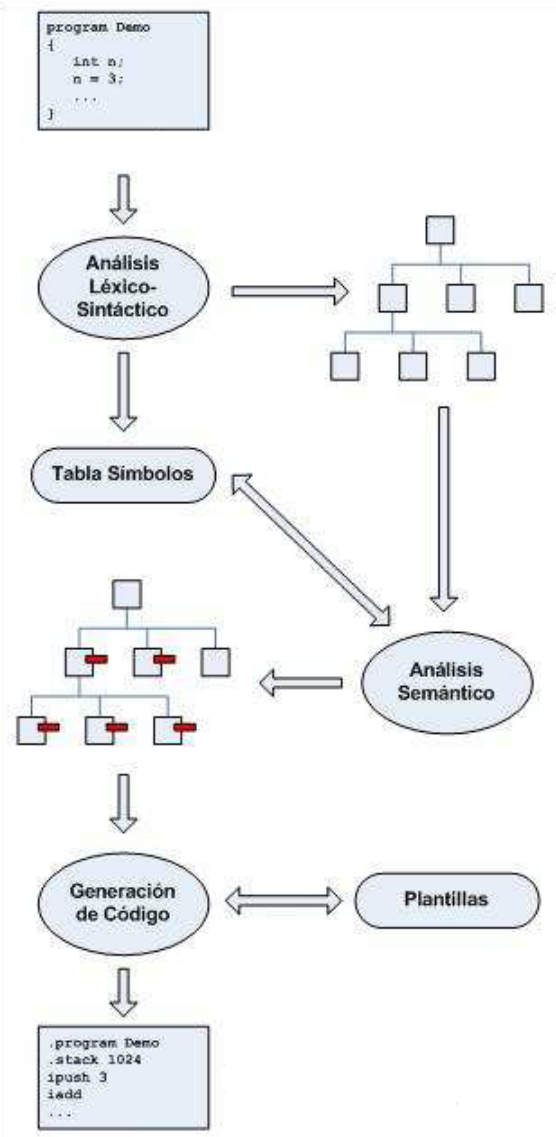
Web principal de ANTLRWorks

Wiki de documentación (Docs, Tutoriales, Ejemplos...)

Libro: The Definitive ANTLR Reference - Building Domain-Specific Languages

Proceso general del compilador:

De cara a tener una visión global del sistema que vamos a construir vamos a describir brevemente el proceso general que se seguirá durante la compilación y cada uno de los módulos que van a intervenir en el proceso. Como se muestra en la siguiente imagen:



Como punto de partida se tomará el archivo fuente con el código escrito, el cual será procesado por el compilador para generar un fichero en lenguaje intermedio. El proceso de compilación se desarrollará íntegramente utilizando ANTLR 3 y se dividirá en las siguientes fases:

1. Análisis léxico-sintáctico.

2. Análisis semántico.

3. Generación de código.

En primer lugar, los **análisis léxico y sintáctico** parsearán todo el código, detectando y reportando posibles errores de sintaxis, y se generará un **árbol de sintaxis abstracta** (**AST**) con todos los elementos relevantes del código del programa y su estructura completa. Además, durante el análisis se creará la tabla de símbolos, que contendrá todas las variables del programa junto a su tipo y su número de orden en el programa. Si los análisis léxico y sintáctico se realizan sin errores se procederá al **análisis semántico**, que a partir de la **tabla de símbolos** y el **AST** generados en el paso anterior se encargará de enriquecer dicho AST con el tipo de cada una de las expresiones referenciadas en el programa y verificará que no existen errores de tipo en el código. Posteriormente, y si el análisis semántico se realiza correctamente, se ejecutará la fase de **generación de código**, que tomará como entrada el **AST enriquecido** y generará a partir de él el **código intermedio** obtenido del programa, utilizando para ello una serie de plantillas construidas a priori.

Análisis Léxico y Sintáctico:

En esta sección comentaremos la construcción mediante ANTLR 3 de los analizadores léxico y sintáctico para nuestro código fuente.

En primer lugar cabe destacar que ANTLR v3 permite definir los analizadores léxico (*lexer*) y sintáctico (*parser*) en ficheros independientes o en un solo fichero donde aparezcan ambos. Dado que en nuestro caso ambos analizadores son relativamente sencillos vamos a optar por la segunda opción.

Lo primero que debemos especificar en el fichero ANTLR será el tipo de gramática a definir (lexer grammar, parser grammar, o grammar para combinar análisis léxico y sintáctico en un mismo archivo) y el nombre que recibirá la gramática, en nuestro caso "FKVM".

En segundo lugar indicaremos las opciones del analizador, donde sólo incluiremos la opción language, que indica el lenguaje en el que se generará el código de los analizadores, utilizaremos C#, y el tipo de salida producida por el analizador sintáctico, en nuestro caso un árbol AST (opción output) de tipo CommonTree (opción ASTLabelType).

```
------------------------------------------------------------------

grammar FKVM;
options {
output=AST;
ASTLabelType=CommonTree;
language=CSharp;
}
------------------------------------------------------------------
```

Analizador Léxico:

El analizador léxico se encargará de reconocer y separar convenientemente los elementos básicos (*tokens*) del lenguaje que estamos construyendo. En la mayoría de los casos deberemos distinguir: literales de los distintos tipos de datos que existan en el lenguaje, identificadores (ya sean palabras clave o no) y deberemos definir qué se entenderá como espacio en blanco entre elementos.

- Literales:

Dentro del Analizador existen 4 tipos de datos: enteros, reales, lógicos y cadenas. Deberemos definir por tanto cómo reconocer cada uno de estos literales en nuestro código. Para ayudar a que las definiciones de estos elementos no sean demasiado complejas y repetitivas definiremos dos reglas auxiliares (para las cuales se utiliza la palabra clave fragment) para reconocer dígitos y letras. Una vez definidas estas reglas auxiliares, el resto son muy sencillas. Por ejemplo, un literal entero estará formado por cualquier combinación de 1 o más dígitos (lo que se indica con el operador '+' de ANTLR). Los literales de tipo cadena estarán formados por una doble comilla seguida de cualquier combinación de caracteres distintos a dobles comillas, saltos de línea o tabuladores y por último otra doble comilla de cierre. Por su parte, un literal lógico tan sólo podrá tomar dos valores: true o false.

```
------------------------------------------------------------------

fragment LETRA : 'a'..'z'|'A'..'Z' ;
fragment DIGITO : '0'..'9' ;
LIT_ENTERO : DIGITO+ ;
LIT_REAL : LIT_ENTERO '.' LIT_ENTERO ;
LIT_CADENA : '"' (~('"'|'\n'|'\r'|'\t'))* '"' ;
LIT_LOGICO : 'true'|'false' ;
------------------------------------------------------------------
```

- Identificadores:

Los identificadores en el Analizador, como ya se indicó en la especificación del lenguaje, estarán formados por cualquier letra o carácter subrayado seguida de cualquier combinación de dígitos, letras o caracteres de subrayado.

```
------------------------------------------------------------------

IDENT: (LETRA|'_')(LETRA|DIGITO|'_')* ;
------------------------------------------------------------------
```

- Comentarios:

Los comentarios permitidos serán de una sola línea y se utilizará la sintaxis de C++ o Java, es decir, precederlos de los caracteres "//". Como puede observarse en la regla se le indicará además a ANTLR que estos *tokens* no deben ser pasados al analizador sintáctico ya que no son de utilidad. Esto lo indicaremos mediante la acción $channel=HIDDEN;

```
------------------------------------------------------------------
COMENTARIO : '//' (~('\n'|'\r'))* '\r'? '\n' {$channel=HIDDEN;};
------------------------------------------------------------------
```

- Espacio en blanco:

Se considerará espacio en blanco entre elementos del lenguaje a toda combinación de caracteres espacio, saltos de línea o tabuladores. Además, estos elementos tampoco serán pasados como tokens al analizador sintáctico.

```
------------------------------------------------------------------

WS : (' '|'\r'|'\n'|'\t')+ {$channel=HIDDEN;} ;

------------------------------------------------------------------
```

Analizador Sintáctico:

A partir de los tokens pasados por el analizador léxico, el analizador sintáctico se encargará de reconocer las combinaciones correctas de tokens que forman instrucciones o expresiones válidas en nuestro lenguaje. Por tanto deberemos definir cuál es la estructura general de un programa en código fuente y la estructura concreta de cada una de las construcciones (instrucciones y expresiones) aceptadas en el lenguaje.

- Programa:

Tal como se indicó en la especificación del lenguaje, un programa en código fuente estará formado por una serie de declaraciones de funciones API (opcionales) seguidas del programa principal que se identifica mediante la palabra clave program seguida de un identificador que indicará el nombre del programa y una lista de instrucciones delimitadas por llaves ('{' y '}'). Las instrucciones a su vez podrán ser: declaraciones, asignaciones, condicionales *if*, bucles *while* o instrucciones de retorno. Abajo podemos ver lo sencillo que resulta definir todo esto en ANTLR v3, y lo que es mejor, utilizando la misma sintaxis usada ya para el analizador léxico.

```
-----------------------------------------------------------------

programa : declaraciones_api principal ;
declaraciones_api : declaracion_api* ;
declaracion_api : 'api' tipo IDENT '(' lista_decl ')' ';' ;
principal : 'program' tipo IDENT '{' lista_instrucciones '}' ;
lista_instrucciones : instruccion* ;
instruccion : inst_decl
| inst_asig
| inst_if
| inst_while
| inst_return
| inst_expr;
-----------------------------------------------------------------
```

- Instrucciones:

La definición de las instrucciones no implica ninguna dificultad. Así, por ejemplo, la instrucción IF de nuestro lenguaje estará formada por la palabra clave if, una expresión (regla que definiremos más tarde) entre paréntesis, una lista de instrucciones para el caso de que se cumpla la condición, y opcionalmente (operador ? de ANTLR) otra lista de instrucciones para el caso de que no se cumpla dicha condición precedida de la palabra clave else. Nótese como las palabras clave y símbolos de puntuación los indicamos directamente entre comillas simples, los tokens devueltos por el analizador léxico con mayúsculas y las reglas del analizador sintáctico en minúsculas.

```
-----------------------------------------------------------------
inst_decl : tipo IDENT ';' ;
inst_asig : IDENT '=' expresion ';' ;
inst_if : 'if' '(' expresion ')' '{' lista_instrucciones '}'
else_otras? ;
else_otras : 'else' '{' lista_instrucciones '}' ;
inst_while : 'while' '(' expresion ')' '{' lista_instrucciones '}'
;
inst_return : 'return' expresion ';' ;
inst_expr : expresion ';' ;
-----------------------------------------------------------------
```

- Expresiones:

La definición de las expresiones en ANTLR sí es más interesante. Dado que ANTLR 3 no provee ningún mecanismo explícito para indicar la precedencia de cada uno de los operadores debemos buscar la forma de que ésta se tenga en cuenta por la propia definición de las reglas. Para ello, el método utilizado será definir cada una de las posibles expresiones (según operadores) de forma que un tipo de expresión quede definido en función del siguiente tipo en el orden de precedencia (siempre de menor a mayor). Así, por ejemplo, como las operaciones de multiplicación y división tienen mayor precedencia que la suma y la resta definiremos ésta última en función de la primera, así nos aseguramos de que las operaciones se asocien siempre de forma correcta.

```
-----------------------------------------------------------------
expMasMenos : expMultDiv (
('+'|'-') expMultDiv)* ;
-----------------------------------------------------------------
```

Para el resto de expresiones se seguiría la misma técnica:

```
-----------------------------------------------------------------
expresion : expOr ;
expOr : expAnd ('||'^ expAnd)* ;
expAnd : expComp ('&&'^ expComp)* ;
expComp : expMasMenos (
('=='|'!='|'>'|'<'|'>='|'<=') expMasMenos)* ;
expMasMenos : expMultDiv (
('+'|'-') expMultDiv)* ;
expMultDiv : expMenos (
('*'|'/') expMenos)* ;
expMenos : '-' expNo
| '+'? expNo ;
expNo : '!'? acceso ;
acceso : IDENT
| literal
| llamada
| '(' expresion ')' ;
-----------------------------------------------------------------
```

- Llamadas a funciones:

Las llamadas a función seguirán la sintáxis clásica compuesta por el nombre de la función seguido de una lista de parámetros (expresiones) entre paréntesis y separados por comas.

```
-----------------------------------------------------------------

llamada : IDENT '(' lista_expr ')' ;

lista_expr : expresion (',' expresion)*

| //Sin parámetros ;

-----------------------------------------------------------------
```

- Otras reglas:

Por último, tan sólo queda definir el resto de reglas auxiliares, como los tipos de datos o los tipos de literales. Estas reglas suelen ser muy sencillas ya que se limitan a enumerar cada uno de los elementos aceptados en el lenguaje.

```
-----------------------------------------------------------------
tipo : 'int'|'float'|'string'|'bool'|'void' ;
literal : LIT_ENTERO
| LIT_REAL
```

```
| LIT_CADENA
| LIT_LOGICO ;
-----------------------------------------------------------------
```

- Programa de prueba:

Hemos finalizado nuestros analizadores básicos. Con esto ya deberíamos ser capaces de comprobar si la sintaxis de un programa escrito en nuestro código fuente es o no válida. Para ello, necesitamos disponer de un programa de prueba, en el que se llame convenientemente a las clases generadas por ANTLR a partir de nuestra gramática.

```
-----------------------------------------------------------------
using System;
using Antlr.Runtime;
using Antlr.Runtime.Tree;
using Antlr.StringTemplate;
namespace PruAntlr
{
class Program
{
static void Main(string[] args)
{
ANTLRFileStream input = new ANTLRFileStream("prueba.fks");
FKVMLexer lexer = new FKVMLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
FKVMParser parser = new FKVMParser(tokens);
FKVMParser.programa_return result = parser.programa();
Console.WriteLine("Analisis finalizado.");
}
}
}
-----------------------------------------------------------------
```

De esta forma, utilizando el programa principal anterior y un fichero de entrada válido como el que se muestra a continuación deberíamos obtener como única salida del programa el mensaje de "Análisis finalizado", ya que si los analizadores no encuentran ningún error no muestran nada a la salida.

```
-----------------------------------------------------------------

program Prueba
{
int a;
a = 1;
}
-----------------------------------------------------------------
```

Sin embargo, si introducimos un error en el fichero de entrada, por ejemplo en este caso hemos eliminado la variable 'a' de la primera declaración, el analizador sintáctico debería mostrarnos el mensaje de error correspondiente.

```
----------------------------------------------------------------------
program Prueba
{
int ;
a = 1;
}
----------------------------------------------------------------------
```

El resultado del análisis debería ser el siguiente:

```
----------------------------------------------------------------------

line 2:5 mismatched input ';' expecting IDENT
Analisis finalizado.
----------------------------------------------------------------------
```

En el mensaje se informa al usuario de que en la posición 5 de la línea 2 del fichero de entrada se ha encontrado un carácter ';' cuando se esperaba un identificador.

- Recuento y reporte de errores:

Vamos a ir ahora un poco más allá y además de mostrar los errores al usuario vamos a mostrar un último mensaje con el número de errores encontrados por los analizadores léxico y sintáctico. Para ello, la técnica que vamos a utilizar será sobescribir los métodos de ANTLR encargados de generar los mensajes de error (GetErrorMessage). La modificación que vamos a realizar a estos métodos será simplemente incrementar una variable propia cada vez que sean llamados y llamar por último al metodo padre para que ANTLR realice el resto de tareas necesarias. El código de estos métodos y la declaración de nuestras variables lo incluiremos en las secciones @lexer::members y @members

```
----------------------------------------------------------------------
@lexer::members {
public int numErrors = 0;
public override string GetErrorMessage(RecognitionException e,
string[]
tokenNames)
{
numErrors++;
return base.GetErrorMessage(e,tokenNames);
}
}
@members {
public int numErrors = 0;
public override string GetErrorMessage(RecognitionException e,
string[]
```

```
tokenNames)
{
numErrors++;
return base.GetErrorMessage(e,tokenNames);
}
}
```

A partir de este momento, desde nuestro programa principal tendremos acceso a nuestra nueva variable que tras llamar a los analizadores contendrán el número de errores encontrados. Así, podríamos modificar la última línea de nuestro programa principal por la siguiente:

```
Console.WriteLine("Análisis finalizado. Errores: " + parser.numErrors);
```

Y ante este archivo de entrada:

```
program Prueba
{
int ;
= 1;
}
```

La salida sería la siguiente:

```
line 2:5 mismatched input ';' expecting IDENT
line 3:2 mismatched input '=' expecting '}'
Análisis finalizado. Errores: 2
```

- Construcción del AST:

Una vez que ya somos capaces de reconocer correctamente ficheros de entrada válidos para nuestro lenguaje y de informar de los posibles errores en caso de producirse el siguiente paso será modificar convenientemente la gramática para construir durante el análisis el árbol de sintaxis abstracta (AST) y la tabla de símbolos que serán utilizados por el analizador semántico y el generador de código.

ANTLR v3 proporciona dos mecanismos básicos de construcción de árboles AST:

1. Mediante operadores.

2. Mediante reglas de reescritura.

Ambos mecanismos pueden mezclarse en una misma gramática y el uso de uno u otro dependerá de la complejidad del árbol que queramos construir. En nuestro caso mezclaremos ambos métodos por lo que veamos en primer lugar un par de ejemplos.
La construcción mediante operadores consiste en incluir en las propias reglas del analizador una serie de operadores que indiquen a ANTLR cómo debe tratarlos para construir el árbol devuelto por la regla. Estos operadores son tan sólo dos: ^ y !. El primero de ellos se utilizará para indicar qué elemento de la regla se utilizará como raíz del árbol (todos los demás pasarán a ser hijos directos de éste) y el segundo operador se añadirá a los elementos que no deben formar parte del árbol. Así, por ejemplo, en la regla inst_while podremos indicar que la palabra clave while se utilizará como raíz del árbol y que los paréntesis y llaves no deben incluirse en el árbol ya que no aportan ningún valor para las fases siguientes.

```
----------------------------------------------------------------------
inst_while : 'while'^ '('! expresion ')'! '{'! lista_instrucciones '}'! ;
----------------------------------------------------------------------
```

En reglas más complejas o donde haya que utilizar *nodos ficticios* (elementos que no aparecen en la regla pero se incluyen en el árbol AST por conveniencia) se pueden utilizar las reglas de reescritura de árboles. Éstas se incluyen a la derecha de la regla precedida por el operador -> y utilizan la sintaxis siguiente:

```
----------------------------------------------------------------------
regla -> ^(raiz hijo1 hijo2 ...)
----------------------------------------------------------------------
```

En nuestro caso podemos poner como ejemplo la regla para definir las asignaciones inst_asig, donde indicaremos mediante una regla de reescritura que queremos construir un árbol con un nodo raíz ficticio llamado ASIGNACION y dos hijos, uno con el identificador y otro con la expresión asignada.

```
----------------------------------------------------------------------
```

```
inst_asig : IDENT '=' expresion ';' -> ^(ASIGNACION IDENT expresion);
-----------------------------------------------------------------
```

Los nodos ficticios deben declararse al principio de la gramática en la sección tokens. En nuestro caso utilizaremos los siguientes:

```
-----------------------------------------------------------------
tokens {
PROGRAMA;
DECLARACION;
DECLARACIONPARAM;
LISTADECLARACIONESAPI;
DECLARACIONAPI;
ASIGNACION;
MENOSUNARIO;
LISTAINSTRUCCIONES;
LLAMADA;
LISTAEXPRESIONES;
LISTAPARAMETROS;
}
-----------------------------------------------------------------
```

- Construcción de la Tabla de Símbolos:

Otra de las tareas a realizar durante el análisis sintáctico será la construcción de la tabla de símbolos. Esta estructura deberá contener al finalizar el análisis una relación fácilmente accesible que contenga todos los identificadores utilizados en el programa junto cualquier información asociada a dicho identificador que sea relevante para las etapas posteriores, como por ejemplo el tipo de dato de la variable.

En nuestro caso, construiremos una tabla de símbolos que contenga cada identificador junto a su tipo y su número de orden dentro del programa. Para ello definiremos en primer lugar una clase para encapsular toda la información asociada a cada identificador:

```
-----------------------------------------------------------------
public class Symbol
{
public int numvar; //Número de orden la variable
public string type; //Tipo de la variable
//Constructor de la clase
public Symbol(string t, int n)
{
type = t;
numvar = n;
}
}
-----------------------------------------------------------------
```

El siguiente paso será declarar la estructura que contendrá la tabla de símbolos. Nosotros utilizaremos una colección genérica de tipo Dictionary con claves de tipo string para el nombre de a variable y valores de tipo Symbol que acabamos de crear. La declaración de esta estructura debe incluirse en la sección @members y los *includes* necesarios en la sección @header de la gramática.

```
----------------------------------------------------------------------
@header {
using System.Collections.Generic;
}
@members {
public Dictionary<string,Symbol> symtable = new
Dictionary<string,Symbol>();
int numVars = 0;
...
}
----------------------------------------------------------------------
```

En nuestro código fuente, el único lugar válido donde pueden definirse variables es dentro de las declaraciones, por lo que la única regla donde debemos ir actualizando la tabla de símbolos será inst_decl. Por tanto, dentro de esta regla, además de la regla de reescritura para construir el árbol AST deberemos incluir una acción (entre llaves) donde se añada el identificador declarado a la tabla de símbolos. Esta acción se limitará a comprobar si el identificador ya existe en la tabla y añadirlo en caso de no existir. Veamos cómo:

```
----------------------------------------------------------------------
inst_decl : tipo IDENT ';' {
if(!symtable.ContainsKey($IDENT.text))
{
symtable.Add($IDENT.text, new Symbol($tipo.text, numVars++));
}
} -> ^(DECLARACION tipo IDENT);
----------------------------------------------------------------------
```

- Finalizando el analizador léxico-sintáctico:

Una vez completada la implementación de nuestra gramática tan sólo nos queda probarla mediante el programa principal que ya comenzamos en apartados anteriores. En esta ocasión vamos a modificarlo para que al finalizar el análisis nos muestre el árbol AST construido y la tabla de símbolos con nuestras variables y su información asociada.

```
----------------------------------------------------------------------
using System;
using Antlr.Runtime;
using Antlr.Runtime.Tree;
using Antlr.StringTemplate;
using System.Collections.Generic;
namespace PruAntlr
{
class Program
```

```
{
static void Main(string[] args)
{
ANTLRFileStream input =
new ANTLRFileStream("C:\\pruantlr\\prueba.fks");
FKVMLexer lexer = new FKVMLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
FKVMParser parser = new FKVMParser(tokens);
FKVMParser.programa_return result = parser.programa();

if (parser.numErrors == 0)
{
CommonTree t = (CommonTree)result.Tree;
Console.WriteLine("Arbol AST:");
Console.WriteLine(t.ToStringTree() + "\n");
Console.WriteLine("Tabla de Simbolos:");
foreach (string k in parser.symtable.Keys)
{
Console.WriteLine(((Symbol)parser.symtable[k]).numvar +
" - " + k + " -> " + ((Symbol)parser.symtable[k]).type);
}
}
else
{
Console.WriteLine("Errores: " + parser.numErrors);
}
Console.ReadLine();
}
}
}
```

El árbol AST lo obtenemos mediante la propiedad Tree del objeto que representa a la regla principal de nuestra gramática programa_return y lo imprimimos por pantalla mediante el método toStringTree(). Por su parte, a la tabla de símbolos podemos acceder como a un atributo más del objeto parser ya que la declaramos en la sección @members.

De esta forma, para el siguiente fichero de entrada:

```
program Prueba {

int a;

float b;

a = 1;

}
```

```
------------------------------------------------------------------
```

Obtendríamos la siguiente salida (la formateo para que sea legible):

```
------------------------------------------------------------------
Arbol AST:
(program Prueba
(LISTAINSTRUCCIONES
(DECLARACION int a)
(DECLARACION float b)
(ASIGNACION a 1)
)
)
Tabla de Simbolos:
0 - a -> int
1 - b -> float
------------------------------------------------------------------
```

Análisis Semántico de nuestro código fuente:

Dedicaremos esta sección a comentar todos los aspectos prácticos del desarrollo de la etapa de análisis semántico del compilador para nuestro lenguaje código fuente. Veremos en primer lugar las modificaciones necesarias que hay que realizar al árbol AST construido en la fase anterior, posteriormente describiremos el tipo de comprobaciones que se realizarán en esta fase del análisis y por último veremos cómo podemos implementarlas con ANTLR. Todas las tareas del analizador semántico se realizarán mediante el recorrido de árboles AST, lo que en ANTLR se llama tree grammar. Veremos más adelante cómo implementar un analizador de este tipo con ANTLR v3.

- Tareas del análisis semántico:

Durante la etapa de análisis semántico nuestro compilador realizará una serie de comprobaciones que ya nada tienen que ver con la sintaxis del lenguaje, y que en nuestro caso serán las siguientes:

· Comprobación de la existencia de variables y funciones.
· Cálculo de tipos en expresiones.
· Chequeo de tipos en instrucciones.
· Chequeo de tipos en expresiones.

Iremos enumerando todas esas comprobaciones a medida que vayamos comentando la implementación sobre ANTLR por lo que no entraremos por el momento en más detalle.

- Enriqueciendo los nodos del árbol AST:

Por defecto, el tipo de árbol construido por ANTLR es del tipo CommonTree cuyos nodos únicamente contienen un tipo (atributo Type) y un valor (atributo Text). Esta información no suele ser suficiente en la mayoría de las ocasiones por lo que se hace necesario definir un tipo de árbol personalizado con toda la información que necesite nuestro compilador. En nuestro caso vamos a necesitar para los elementos simples (identificadores y literales) toda la información contenida en la clase Symbol que ya definimos y dos campos adicionales para almacenar el tipo de las expresiones compuestas expType y el tipo de sus subexpresiones expSecType. Para conseguir esto simplemente tendremos que definir una clase derivada de CommonTree que contenga toda esta información y un constructor adecuado que inicialice todo lo necesario y llame al constructor de la clase padre. Veamos cómo quedaría esta clase a la que llamaremos FkvmAST:

```
----------------------------------------------------------------------
using Antlr.Runtime;
using Antlr.Runtime.Tree;
public class FkvmAST : Antlr.Runtime.Tree.CommonTree
{
public Symbol symbol;
public string expType = "";
public string expSecType = "";
public FkvmAST(IToken t) : base(t)
{
//Nada que inicializar
}
}
----------------------------------------------------------------------
```

Una vez tenemos definida la clase que describirá nuestros nodos del árbol AST debemos indicar a ANTLR que use ésta para construir el árbol durante la fase de análisis sintáctico. Para ello deberemos modificar el valor asignado a la opción ASTLabelType indicando nuestra nueva clase.

```
----------------------------------------------------------------------

options {

output=AST;

ASTLabelType=FkvmAST;

language=CSharp;

}

----------------------------------------------------------------------
```

Pero no nos basta con esto. Debemos crear también un adaptador para nuestro nuevo tipo de árbol. Este adaptador debe derivar de la clase CommonTreeAdaptor y tan sólo deberemos redefinir el método Create para devolver un árbol de tipo FkvmAST. Veamos cómo quedaría nuestro adaptador:

```
-----------------------------------------------------------------
class FKTreeAdaptor : CommonTreeAdaptor
{
public override object Create(IToken payload)
{
return new FkvmAST(payload);
}
}
-----------------------------------------------------------------
```

Una vez creado el nuevo adaptador debemos indicar a ANTLR en el programa principal que debe hacer uso de éste para la construcción del árbol de sintaxis abstracta durante la fase de análisis sintáctico. Para ello asignaremos la propiedad TreeAdaptor de nuestro parser antes de comenzar el análisis:

```
-----------------------------------------------------------------
CommonTokenStream tokens = new CommonTokenStream(lexer);
FKVMParser parser = new FKVMParser(tokens);
ITreeAdaptor adaptor = new FKTreeAdaptor();
parser.TreeAdaptor = adaptor;
FKVMParser.programa_return result = parser.programa();
-----------------------------------------------------------------
```

- Implementación del analizador en ANTLR v3:

Como indicamos anteriormente, para la fase de análisis semántico vamos a utilizar un analizador de árboles AST, lo que en ANTLR se define como tree grammar. Las opciones de este analizador serán las mismas que las establecidas en la fase anterior, con la excepción de la opción tokenVocab que indicará a ANTLR que debe utilizar los mismos tokens que se generaron para la fase de análisis sintáctico. De esta forma, ANTLR importará estos tokens desde el fichero FKVM.tokens generado en la fase anterior.

```
-------------------------------------------------------------------
tree grammar FKVMSem;
options {
tokenVocab=FKVM;
ASTLabelType=FkvmAST;
language=CSharp;
}
-------------------------------------------------------------------
```

Por otro lado, el analizador semántico recibirá como entrada, además del árbol AST, la tabla de símbolos generada anteriormente por lo que deberá declararse la variable que la contendrá. Como siempre haremos esto en las secciones @header y @members. Incluiremos también la variable numErrors para ir realizando el recuento de errores de forma análoga a la fase anterior.

```
-------------------------------------------------------------------
@header {
using System.Collections.Generic;
}
@members {
public Dictionary<string,Symbol> symtable;
public int numErrors = 0;
}
-------------------------------------------------------------------
```

El paso de parámetros a una regla en ANTLR se indica a continuación de la regla y entre corchetes, y la asignación de dicho parámetro a nuestra variable interna la podemos realizar en la sección @init de nuestra regla principal. Vemos cómo quedaría por la primera regla de nuestra gramática:

```
-------------------------------------------------------------------
programa[Dictionary<string,Symbol> symtable]
@init {
this.symtable = symtable;
}
: ^(PROGRAMA declaraciones_api principal) ;
-------------------------------------------------------------------
```

En el último apartado de esta sección veremos cómo podemos pasar en el programa principal la tabla de símbolos construida durante el análisis sintáctico como parámetro del analizador semántico.

- Cálculo y chequeo de tipos:

El cálculo y chequeo de tipos de una expresión se realizará comenzando por las expresiones más simples de nuestra gramática, es decir, los literales e identificadores. Una vez calculado el tipo de una expresión se asignará éste a su nodo del árbol y además se devolverá como valor de retorno para que pueda ser consultado por las reglas superiores a la actual. El valor de retorno se indica en ANTLR mediante la cláusula returns seguida de la declaración entre corchetes de la variable a devolver.

```
-----------------------------------------------------------------

expresion returns [String type]

-----------------------------------------------------------------
```

Esta variable se inicializará en la sección @init de la regla y se asignará convenientemente dentro de cada subregla. Además, dentro de esta sección también obtendremos una referencia al nodo principal del arbol de la expresión para poder asignarlo al fializar la regla. Este nodo lo obtendremos mediante el método LT() del objeto input, que representa la secuencia de nodos del árbol que estamos analizando. De esta forma, el siguiente nodo de la secuencia, lo obtendremos mediante la llamada input.LT(1). Por último, en la sección @after de la regla asignaremos a este nodo el tipo calculado y que será devuelto como retorno. Veamos pues como queda la regla por el momento:

```
-----------------------------------------------------------------
expresion returns [String type]
@init {
$type="";
FkvmAST root = (FkvmAST)input.LT(1);
}
@after {
root.expType = $type;
}
-----------------------------------------------------------------
```

Todo el cálculo de tipos se realizará en la regla que analiza las expresiones, donde están contenidos tanto los elementos más simples como los literales o identificadores, como las expresiones más complejas, donde habrá que calcular su tipo en función de los elementos que la componen. Empecemos por tanto por los elementos más sencillos.

- Literales:

El cálculo del tipo de un literal será tan sencillo como consultar el tipo de token devuelto por el analizador sintáctico y asignar directamente un tipo u otro en la subregla correspondiente. Como puede verse en el código siguiente, la técnica seguida en la regla literal para devolver el tipo calculado a la regla superior y asignarlo a su vez al nodo del árbol es la misma que la descrita para las expresiones.

```
-----------------------------------------------------------------
expresion returns [String type]
...
| literal {$type=$literal.type;}
;
literal returns [String type]
@init {
$type="";
FkvmAST root = (FkvmAST)input.LT(1);
}
@after {
```

```
root.expType = $type;
}
: LIT_ENTERO {$type="int";}
| LIT_REAL {$type="float";}
| LIT_CADENA {$type="string";}
| LIT_LOGICO {$type="bool";}
;
-----------------------------------------------------------------
```

- Identificadores:

El cálculo de tipo para un identificador es más sencillo aún que en el caso de los literales ya que nos basaremos directamente en el tipo almacenado en la tabla de símbolos para dicho identificador. Además, en caso de no encontrar en la tabla de símbolos algún identificador informaremos del error al usuario, de forma que no permitiremos en nuestro lenguaje el uso de variables que no hayan sido declaradas.

```
-----------------------------------------------------------------
expresion returns [String type]
...
| IDENT {
if(symtable.ContainsKey($IDENT.text))
{
root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;
}
else
{
registrarError(root.Line, "Identifier '" + $IDENT.text + "' has
not been declared.");
}
}
| literal {$type=$literal.type;}
;
-----------------------------------------------------------------
```

- Llamadas a función externa:

El tipo de una llamada a una función externa se calcula de la misma forma que el de los identificadores ya que el mecanismo que hemos utilizado para registrar su tipo durante el análisis sintáctico ha sido el mismo, la tabla de símbolos. Veamos por tanto cómo queda esta regla:

```
-----------------------------------------------------------------

expresion returns [String type]
...
| IDENT {root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;}
| literal {$type=$literal.type;}
| llamada {$type=$literal.type;}
```

```
;
llamada returns [String type]
@init {
$type="";
FkvmAST root = (FkvmAST)input.LT(1);
}
@after {
root.expType = $type;
}
: ^(LLAMADA IDENT lista_expr) {
if(symtable.ContainsKey($IDENT.text))
{
root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;
}
else
{
registrarError(root.Line, "Api function '" + $IDENT.text + "' has
not been declared.");
}
} ;
```

- Expresiones complejas:

Una vez hemos calculado el tipo de las expresiones más simples de nuestro lenguaje ya deberíamos ser capaces de calcular y chequear el de una expresión compuesta por estos elementos. Así, por ejemplo, para las expresiones lógicas consideraremos que su tipo es siempre bool y que ésta es correcta si los tipos de las dos subexpresiones son iguales o uno entero y otro real. Esto lo comprobaremos mediante el método comprobarTipoExpComp(tipo1, tipo2) que definiremos en la sección @members y si se determina que la combinación de tipos no es correcta se reportará al usuario el error correspondiente.

```
expresion returns [String type]

...

| ^(opComparacion e1=expresion e2=expresion) {
$type="bool";
if(!comprobarTipoExpComp($e1.type, $e2.type))
{
registrarError(root.Line, "Incorrect types in expression.");
}
}
| IDENT {root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;}
| literal {$type=$literal.type;}
| llamada {$type=$literal.type;}
```

```
;
@members {
public Dictionary symtable;
public int numErrors = 0;
public bool comprobarTipoExpComp(string t1, string t2)
{
bool res = false;
if(t1.Equals(t2) ||
(t1.Equals("int") && t2.Equals("float")) ||
(t1.Equals("float") && t2.Equals("int")) )
{
res = true;
}
return res;
}
}
-----------------------------------------------------------------
```

El resto de expresiones incluidas en el analizador (expresiones aritméticas, operador menos-unario y operador no-lógico) las definiremos de forma análoga:

```
-----------------------------------------------------------------
expresion returns [String type]
: ^(opComparacion e1=expresion e2=expresion) {
$type="bool";
if(!comprobarTipoExpComp($e1.type, $e2.type))
{
registrarError(root.Line, "Incorrect types in expression.");
}
}
| ^(opAritmetico e1=expresion e2=expresion) {
$type=$e1.type;
if(!comprobarTipoExpArit($e1.type, $e2.type))
{
registrarError(root.Line, "Incorrect types in expression.");
}
}
| ^(MENOSUNARIO e1=expresion) {
$type=$e1.type;
if(!($e1.type.Equals("int") || $e1.type.Equals("float")))
{
registrarError(root.Line, "Incorrect types in expression.");
}
}
| ^('!' e1=expresion) {
$type=$e1.type;
if(!$e1.type.Equals("bool"))
{
registrarError(root.Line, "Incorrect types in expression.");
}
}
```

```
| IDENT {root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;}
| literal {$type=$literal.type;}
| llamada {$type=$literal.type;}
;
-----------------------------------------------------------------
```

- Instrucciones:

El tipo de algunos de los elementos contenidos en instrucciones de nuestro lenguaje también debe ser comprobado durante la fase de análisis semántico. Así, por ejemplo, en las instrucciones de asignación deberemos comprobar que el tipo de la expresión derecha coincide con el de la variable que estamos asignando o que al menos es compatible. El tipo de la variable lo recuperaremos de la tabla de símbolos y el de la expresión accediendo a su atributo $type que ya debería haber sido calculado en la regla expresión. Las combinaciones válidas de tipos las comprobaremos como en el caso de las expresiones mediante un método definido en la sección @members. Veamos cómo quedaría esta regla:

```
-----------------------------------------------------------------
inst_asig
@init {
FkvmAST root = (FkvmAST)input.LT(1);
}
: ^(ASIGNACION IDENT e1=expresion) {
root.expType = $e1.type;
if(symtable.ContainsKey($IDENT.text))
{
$IDENT.symbol = (Symbol)symtable[$IDENT.text];
if(!comprobarTipoAsig(root.expType, $IDENT.symbol.type))
{
registrarError(root.Line, "Incorrect type in assignment.");
}
}
else
{
registrarError(root.Line, "Identifier '" + $IDENT.text +
"' has not been declared.");
}
};
-----------------------------------------------------------------
```

Por su parte, para las instrucciones IF y WHILE deberemos comprobar que la condición viene expresada por una expresión de típo lógico. Para ello tan sólo deberemos comprobar el atributo $type de la expresión que hemos calculado en la regla expresión. Veamos por ejemplo la instrucción IF:

```
-----------------------------------------------------------------
inst_if : ^(ins='if' e1=expresion lista_instrucciones
lista_instrucciones) {
if(!$e1.type.Equals("bool"))
```

```
{
registrarError($ins.Line, "Incorrect type in IF instruction.");
}
};
-----------------------------------------------------------------
```

- Programa principal**:**

Una vez finalizado nuestro analizador semántico podemos llamarlo desde el programa principal pasándole la tabla de símbolos calculada durante la fase anterior. Para ello, simplemente se creará el objeto FKVMSem con las estructuras necesarias procedentes del análisis sintáctico y se llamará a su método principal cuyo nombre debe coincidir con la regla principal de nuestra gramática, en este caso programa(). La tabla de símbolos se le pasará como parámetro de esta llamada ya que así lo definimos en la gramática.

```
-----------------------------------------------------------------
//Análisis léxico semántico

FKVMLexer lexer = new FKVMLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
FKVMParser parser = new FKVMParser(tokens);
parser.TreeAdaptor = adaptor;
FKVMParser.programa_return result = parser.programa();
//Si no hay errores léxicos ni sintácticos ==> Análisis Semántico
if (lexer.numErrors + parser.numErrors == 0)
{

//Analisis Semántico
CommonTree t = ((CommonTree)result.Tree);
CommonTreeNodeStream nodes2 = new CommonTreeNodeStream(t);
nodes2.TokenStream = tokens;
FKVMSem walker2 = new FKVMSem(nodes2);
walker2.programa(parser.symtable);
}
-----------------------------------------------------------------
```

Generación de código a partir de nuestro código fuente:

- Primeros pasos:

El paso de generación de código se implementará haciendo uso de ANTLR v3, mediante una gramática de tipo *tree grammar* al igual que la fase anterior, y utilizando la librería StringTemplate para generar los ficheros de salida a partir de plantillas predefinidas.
Empecemos como siempre por ver las opciones del analizador. En este caso, las opciones serán las mismas que las definidas en el analizador semántico más una adicional para indicar que la salida no será un árbol AST sino una plantilla de StringTemplate. Esta última opción se indicará con output=template.

```
------------------------------------------------------------------
tree grammar FKVMGen;
options {
tokenVocab=FKVM;
ASTLabelType=FkvmAST;
output=template;
language=CSharp;
}
------------------------------------------------------------------
```

En los siguientes apartados veremos cómo definir con ANTLR+StringTemplate la generación de código para cada uno de los elementos significativos de nuestro código fuente.

- Generación de código para literales e identificadores:

Vamos a empezar a comentar el proceso de análisis por las expresiones más simples del lenguaje como son los literales y los identificadores. Veamos en primer lugar un ejemplo de cómo se traduciría a código intermedio una sencilla instrucción de asignación en la que tan sólo interviene un literal entero:

```
------------------------------------------------------------------
//Código en Fuente
int a; //Variable número 1
a = 3;

#Código traducido a FKIL
ipush 3 #Almacena el valor 3 en la cima de la pila
istore 1 #Almacena la cima de la pila en la variable 1
------------------------------------------------------------------
```

Como puede observarse, para el literal la única instrucción de código FKIL que se genera es una instrucción de tipo PUSH con su tipo correspondiente, en este caso de tipo entero (IPUSH), seguida del literal correspondiente. La plantilla a definir para la generación de código de un literal entero será por tanto de la siguiente forma:

```
------------------------------------------------------------------
lit_entero(v) ::= "ipush <v>"
------------------------------------------------------------------
```

Para el resto de tipos de literales estas plantillas serán análogas, cambiando únicamente el tipo de la instrucción PUSH. Por su parte, en la gramática ANTLR tan sólo deberemos indicar la plantilla correspondiente dependiendo del tipo de literal leído y le pasaremos como argumento el texto del propio literal:

```
------------------------------------------------------------------
expresion
...
```

```
|literal -> {$literal.st}
;
literal : LIT_ENTERO -> lit_entero(v={$LIT_ENTERO.text})
| LIT_REAL -> lit_real(v={$LIT_REAL.text})
| LIT_CADENA -> lit_cadena(v={$LIT_CADENA.text})
| LIT_LOGICO -> lit_logico(v={$LIT_LOGICO.text})
;
-----------------------------------------------------------------
```

Ampliemos ahora el ejemplo anterior para que también intervenga un identificador en la parte derecha de una asignación y veamos cómo quedaría su traducción a código FKIL:

```
-----------------------------------------------------------------
//Código fuente
int a; //Variable número 1
int b; //Variable número 2
a = 3;
b = a;

#Código traducido a FKIL
ipush 3 #Almacena el valor 3 en la cima de la pila
istore 1 #Almacena la cima de la pila en la variable 1
iload 1 #Recupera el valor de la variable 1 a la cima de la pila
istore 2 #Almacena la cima de la pila en la variable 2
-----------------------------------------------------------------
```

El código generado en este caso para la variable "a" en la asignación "b=a" ha sido tan sólo una instrucción de tipo LOAD (en este caso de tipo entero, ILOAD) seguida del número de orden de la variable dentro el programa. La plantilla a definir y la generación de código por tanto será tan sencilla como en el caso de los literales. Veamos en primer lugar la plantilla que hemos definido:

```
-----------------------------------------------------------------
ident(op,nv) ::= <<
<op> <nv>
>>
-----------------------------------------------------------------
```

Vemos que en este caso le pasaremos dos argumentos a la plantilla, el primero de ellos indicando el operador que utilizaremos para recuperar la variable (ILOAD, FLOAD, SLOAD o BLOAD), que dependerá del tipo del identificador, y el segundo que contendrá el número de orden de la variable en el programa. La gramática quedaría de la siguiente forma:

```
-----------------------------------------------------------------
expresion
...
```

```
|IDENT {oper=traducirTipo($IDENT.expType)+"load";}
-> ident(op={oper},nv={$IDENT.symbol.numvar})
|literal -> {$literal.st}
;
-----------------------------------------------------------------
```

Para generar el operador hemos utilizado un método propio llamado traducirTipo() que devolverá el prefijo correcto del operador LOAD dependiendo del tipo del tipo del identificador pasado como parámetro, es decir, para un identificador de tipo entero devolverá "i", para uno de tipo real devolverá "f" y de forma análoga para el resto de tipos. Por otro lado, el número de la variable lo obtendremos directamente del atributo symbol.numvar del identificador, dato que generamos durante la fase de análisis sintáctico.

- Generación de código para expresiones aritméticas:

Veamos en primer lugar un ejemplo de generación de código para una expresión de este tipo:

```
-----------------------------------------------------------------
//Código fuente

int a; //Variable número 1

int b; //Variable número 2

a = 3;

b = a + 5;

#Código traducido a FKIL

ipush 3 #Almacena el valor 3 en la cima de la pila

istore 1 #Almacena la cima de la pila en la variable 1

iload 1 #Recupera el valor de la variable 1 a la cima de la pila

ipush 5 #Almacena el valor 3 en la cima de la pila

iadd #Realiza la suma de los dos valores superiores de la pila

#y almacena el resultado en la cima de la pila.

istore 2 #Almacena la cima de la pila en la variable 2

-----------------------------------------------------------------
```

Como se observa en el ejemplo, para la expresiones aritméticas lo que se hará será generar en primer lugar el código de las subexpresiones, en este caso un identificador (ILOAD 1) y un literal

(IPUSH 5) y posteriormente llamar al operador correspondiente según el tipo de expresión (en nuestro caso IADD). El patrón para generar la plantilla parece por tanto claro:

```
-----------------------------------------------------------------
op_aritmetico(op,e1,e2) ::= <<
<e1>
<e2>
<op>
>>
-----------------------------------------------------------------
```

La plantilla recibirá tres argumentos: el operador aritmético a utilizar y el código de las dos subexpresiones de la expresión que estamos generando. En la gramática se seguirá un proceso muy parecido al de los identificadores:

```
-----------------------------------------------------------------
expresion
...
| ^(opa=opAritmetico e1=expresion e2=expresion)
{oper=traducirTipo($opa.opType)+$opa.st.ToString();}
-> op_aritmetico(op={oper}, e1={$e1.st}, e2={$e2.st})
...
opAritmetico returns [string opType]
: op='+' -> {%{"add"}; $opType=$op.expType;}
| op='-' -> {%{"sub"}; $opType=$op.expType;}
| op='*' -> {%{"mul"}; $opType=$op.expType;}
| op='/' -> {%{"div"}; $opType=$op.expType;}
;
-----------------------------------------------------------------
```

El prefijo del operador lo obtendremos de la misma forma que para el caso de los identificadores, haciendo uso del método traducirTipo(), y el operador en concreto lo generaremos en la regla opAritmetico, donde también devolveremos el tipo de la expresión (que calculamos durante el análisis semántico) para que sirva como parámetro al método de traducción de tipos.

- Generación de código para expresiones lógicas:

El proceso de generación de código para una expresión lógica no será tan directo como para el resto de elementos que hemos comentado hasta el momento, y para tratar de explicar el por qué empecemos como siempre por ver un ejemplo de traducción de una expresión lógica a código FKIL:

```
-----------------------------------------------------------------
Expresión lógica: a > 3
#Traducción a FKIL
ipush 1 #Valor por defecto de la expresión = 1 (true)
iload 1 #Se recupera el valor de la variable 1 a la cima de la pila
ipush 3 #Se coloca el valor 3 en la cima de la pila
ncmp #Se comparan los dos valores superiores de la pila
```

```
ifgt etiq1 #Si el resultado de la comparación es >0 se salta a
"etiq1"
pop #Se desapila el valor por defecto
ipush 0 #Se apila el valor contrario =0 (false)
etiq1: #Etiqueta de salida
-----------------------------------------------------------------
```

Como vemos, la estrategia a seguir para calcular el resultado de este tipo de expresiones será siempre suponiendo que la expresión es verdadera, realizar la comparación, y en caso de ser falsa desapilar el resultado por defecto (true) y apilar el contrario (false). En este patrón sin embargo hay muchos elementos variables que se deberán tener en cuenta a la hora de definir la plantilla a utilizar para la generación de expresiones lógicas. El primero de ellos es el *tipo de comparación*. En el ejemplo hemos utilizado el operador ncmp (comparación numérica) debido a que las dos subexpresiones (la variable "a" y el literal "3") eran de tipo entero. Sin embargo, en el caso de comparaciones de cadenas debería utilizarse el operador scmp y para los valores lógicos el operador bcmp. El segundo parámetro a considerar será el operador utilizado para saltar a la etiqueta de salida. Así, para el operador ">" se utilizará ifgt, para "<" usaríamos iflt y de forma análoga para el resto de operadores lógicos. Por último, un factor importante será el nombre de la etiqueta de salida, ya que debemos asegurarnos de que en el programa resultante no se repita el nombre de ninguna etiqueta.

Teniendo todo esto en cuenta veamos cómo quedaría la gramática y la plantilla para estas expresiones:

```
-----------------------------------------------------------------
@members {
int nEtiqueta = 1;
private string operadorComparacion(String t)
{
string op = "";
if(t.Equals("int") || t.Equals("float"))
op = "ncmp";
else if(t.Equals("string"))
op = "scmp";
else if(t.Equals("bool"))
op = "bcmp";
return op;
}
}
expresion
...
: ^(opc=opComparacion e1=expresion e2=expresion) {operc =
operadorComparacion($opc.opSecType);}
-> op_comparacion(opc={operc}, op={$opc.st}, e1={$e1.st},
e2={$e2.st},
et1={nEtiqueta++})
...
opComparacion returns [string opSecType]
```

```
: op='==' -> {%{"ifeq"}; $opSecType=$op.expSecType;}
| op='!=' -> {%{"ifne"}; $opSecType=$op.expSecType;}
| op='>=' -> {%{"ifge"}; $opSecType=$op.expSecType;}
| op='<=' -> {%{"ifle"}; $opSecType=$op.expSecType;}
| op='>' -> {%{"ifgt"}; $opSecType=$op.expSecType;}
| op='<' -> {%{"iflt"}; $opSecType=$op.expSecType;}
;
-----------------------------------------------------------------
```

El primero de los problemas a resolver, la elección del operador de comparación, lo solventamos mediante el método propio operadorComparacion() (definido en la sección @members) al que le pasaremos el tipo de las subexpresiones de la expresión lógica que estamos generando. Este tipo lo almacenamos durante el análisis semántico en el atributo expSecType por lo que sólo tenemos que recuperarlo en la regla opComparacion. El operador IF a utilizar para el salto a la etiqueta de salida también lo decidimos en dicha regla y se asignará directamente dependiendo del tipo de expresión que hayamos leido de nuestro programa. Por último, el nombre de la etiqueta se generará a partir de un secuencial que iremos incrementando durante la ejecución de nuestro analizador cada vez que haga falta generar una etiqueta. Eset secuencial lo declararemos una vez más en la sección @members y en nuestro caso se llamará nEtiqueta. La plantilla por su parte quedaría de la siguiente forma:

```
-----------------------------------------------------------------
op_comparacion(opc,op,e1,e2,et1) ::= <<
ipush 1
<e1>
<e2>

<opc>
<op> etiq<et1>
pop
ipush 0
etiq<et1>:
>>
-----------------------------------------------------------------
```

- Generación de código para asignaciones:

La generación de código para las asignaciones es muy similar al ya visto para los identificadores ya que la única dificultad a salvar será el operador STORE a utilizar para almacenar el valor de la variable, y éste se calculará a partir del tipo de la expresión derecha mediante el método ya comentado traducirTipo().

```
-----------------------------------------------------------------
inst_asig
@init {
string oper = "";
} : ^(ASIGNACION IDENT expresion) {oper =
traducirTipo($ASIGNACION.expType)+"store";}
-> asignacion(op={oper}, nv={$IDENT.symbol.numvar}, val={$expresion.st});
-----------------------------------------------------------------
```

El número de orden de la variable se recuperará del atributo symbol.numvar que calculamos durante el análisis sintáctico. La plantilla, muy sencilla en este caso, quedaría así:

```
-----------------------------------------------------------------
asignacion(op, nv, val) ::= <<
<val>
<op> <nv>
>>
-----------------------------------------------------------------
```

- Generación de código para instrucciones condicionales y bucles:

Vemos ahora la generación de código para las instrucciones if y while del código fuente. Ninguna de ellas añade ninguna dificultad a las ya comentadas por lo que las trataremos muy brevemente. Veamos en primer lugar un ejemplo de traducción de cada una de ellas:

```
-----------------------------------------------------------------
//Instrucción IF
if(...expresion-logica...)
{

//instrucciones-si

}
else
{

//instrucciones-else

}
#Traducción a FKIL
...expresión-logica...
ifeq etiq1
...instrucciones-si...
goto etiq2
etiq1:
...instrucciones-else...
etiq2:

//Instrucción WHILE

while(...expresion-logica...)
{
//instrucciones-while
}
#Traducción a FKIL
etiq1
...expresion-logica...
ifeq etiq2
```

```
...instrucciones-while...
goto etiq1
etiq2:
-----------------------------------------------------------------
```

A partir de estos patrones, las plantillas de StringTemplate a definir son prácticamente directas:

```
-----------------------------------------------------------------
instif(cond,instsi,instelse,et1,et2) ::= <<
<cond>
ifeq etiq<et1>
<instsi>
goto etiq<et2>
etiq<et1>:
<instelse>
etiq<et2>:
>>
instwhile(cond,instrucciones,et1,et2) ::= <<
etiq<et1>
<cond>
ifeq etiq<et2>
<instrucciones>
goto etiq<et1>
etiq<et2>:
>>
-----------------------------------------------------------------
```

La gramática por su parte tampoco añade ninguna particularidad extra a las ya comentadas en apartados anteriores por lo que no nos pararemos demasiado en comentarla:

```
-----------------------------------------------------------------
inst_if : ^('if' expresion isi=lista_instrucciones
ielse=lista_instrucciones)
->
instif(cond={$expresion.st},instsi={$isi.st},instelse={$ielse.st},
et1={nEtiqueta++},et2={nEtiqueta++});
inst_while : ^('while' expresion li=lista_instrucciones)
-> instwhile(cond={$expresion.st},instrucciones={$li.st},
et1={nEtiqueta++},et2={nEtiqueta++});
-----------------------------------------------------------------
```

- Generación de código para el programa principal FKIL:

Una vez comentada la generación de código para cada uno de los elementos principales de nuestro lenguaje ya sólo nos queda ver cómo generar la estructura principal del programa. Esto lo haremos en la regla principal de la gramática.

```
-----------------------------------------------------------------
principal[int locales] : ^('program' tipo IDENT li=lista_instrucciones)
-> principal(nom={$IDENT.text}, loc={locales},
```

```
instr={$li.st});
-----------------------------------------------------------------
```

Como vemos, esta regla recibirá un parámetro externo llamado locales que contendrá el número de variables locales utilizadas por el programa. Este dato lo usaremos para generar la directiva .locals de FKIL. El parámetro lo deberemos pasar convenientemente desde el programa principal a la hora de llamar al analizador que acabamos de crear. Por lo demás, el único dato adicional necesario será el nombre del programa que lo obtendremos directamente del texto del identificador correspondiente en la regla. La plantilla nos quedaría de la siguiente forma:

```
-----------------------------------------------------------------
principal(nom, loc,instr) ::= <<
.program <nom>
.locals <loc>
<instr>
>>
-----------------------------------------------------------------
```

- Programa principal:

En este punto ya hemos finalizado nuestra gramática y nuestro fichero de plantillas por lo que estamos en condiciones de completar nuestro programa principal para llamar al generador de código al final del proceso.

```
-----------------------------------------------------------------
//Análisis léxico semántico

FKVMLexer lexer = new FKVMLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
FKVMParser parser = new FKVMParser(tokens);
parser.TreeAdaptor = adaptor;
FKVMParser.programa_return result = parser.programa();

//Si no hay errores léxicos ni sintácticos ==> Análisis Semántico

if (lexer.numErrors + parser.numErrors == 0)
{

//Analisis Semántico
CommonTree t = ((CommonTree)result.Tree);
CommonTreeNodeStream nodes2 = new CommonTreeNodeStream(t);
nodes2.TokenStream = tokens;
FKVMSem walker2 = new FKVMSem(nodes2);
walker2.programa(parser.symtable);

//Si no hay errores en el análisis semántico ==> Generación de
código

if (walker2.numErrors == 0)
{
```

```
//Plantillas
TextReader groupFileR = new StreamReader("FkvmIL.stg");
StringTemplateGroup templates = new
StringTemplateGroup(groupFileR);
groupFileR.Close();

//Generación de Código
CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
nodes.TokenStream = tokens;
FKVMGen walker = new FKVMGen(nodes);
walker.TemplateLib = templates;
FKVMGen.programa_return r2 = walker.programa(parser.numVars);
}
}
-----------------------------------------------------------------
```

En primer lugar leeremos el fichero de plantillas FkvmIL.stg para generar el objeto templates. Este objeto se pasará como entrada a nuestro analizador a través del atributo TemplateLib para indicar las plantillas a utilizar durante la generación de código. Por último, no deberemos olvidar pasar como parámetro de entrada el número de variables locales del programa, dato que obtendremos directamente del analizador sintáctico donde como ya comentamos definimos un atributo llamado numVars que contenía precisamente ese dato.

Aquí se da por finalizado el diseño del compilador construido con sus tres analizadores.

A continuación se darán unas especificaciones acerca del código fuente y el código generado por el compilador (FKIL) como ser el listado de instrucciones con distintos ejemplos para ambos códigos.

Especificación del código fuente:

- Código fuente

El código fuente estará compuesto por una serie de declaraciones de funciones API y una única función expresada mediante la sintaxis siguiente:

declaraciones-api
program *tipo nombrePrograma*
{
...CódigoPrograma...
}

El *tipo* podrá ser cualquiera de los indicados en el apartado siguiente e indica el tipo de dato del valor devuelto por el programa.

Por su parte, las declaraciones de la API seguirán la notación siguiente:

api *tipo nombreFuncion* (*lista_parámetros*);

La lista de parámetros estará formada por una serie de declaraciones de variable separadas por comas. Un ejemplo de declaración de función sería el siguiente:

api int sumaEnteros (int entero1, int entero2);

- Tipos de datos

Los tipos de datos permitidos serán:

int	**Valor numérico entero**
float	**Valor numérico real**
bool	**Valor lógico**
string	**Cadena de caracteres**

- Declaración de variables

La declaración de variables en el código fuente se realizará de la misma forma que en los lenguajes C# o Java, con la diferencia de no poder inicializarse dicha variable en la propia declaración. La sintaxis será la siguiente:

Tipo NombreVariable;

El tipo de dato debe ser uno de los indicados en el apartado anterior y el nombre de la variable debe cumplir las siguientes reglas:

1. Comenzar por una letra o por el caracter de subrayado '_'.
2. El resto de caracteres deben ser dígitos, letras o caracteres de subrayado '_'.

El lenguaje será sensible a mayúsculas y minúsculas, por lo que identificadores como por ejemplo NumLinea y numlinea se considerarán distintos.

Como ejemplo, para declarar una variable de tipo entero llamada numeroFila utilizaremos la siguiente sentencia:

int numeroFila;

- Instrucciones

1) Asignaciones:

Las instrucciones de asignación serán idénticas a las de C# o Java, siguiéndose la siguiente sintaxis:

```
NombreVariable = expresión;
```

La expresión asignada podrá ser un literal, un identificador o cualquier operación entre ellos. Así por ejemplo, serán asignaciones válidas:

```
miVariable = 3;
miOtraVariable = miVariable;
otraVariableMas = 3 + (miVariable * 5);
```

2) Instrucción condicional if

La instrucción if seguirá la siguiente sintaxis:

```
if ( expresionLogica )
{
Código ejecutado cuando 'expresionLogica' es cierta
}
else
{
Código ejecutado cuando 'expresionLogica' es falsa
}
```

El bloque else será opcional en esta instrucción.

3) Instrucción iterativa while

La instrucción while seguirá la siguiente sintaxis:

```
while ( expresionLogica )
{
Código ejecutado mientras 'expresionLogica' es cierta
}
```

4) Instrucción return

La instrucción return se utilizará para indicar la expresión devuelta por el programa y seguirá la siguiente sintaxis:

```
return expresion
```

- Expresiones

1) Expresiones Aritméticas

Las expresiones aritméticas permitidas para los tipos int y float serán las siguientes:

+	Suma
-	Resta/Menos unario
*	Producto
/	División

2) Expresiones Lógicas

Las expresiones lógicas permitidas serán las siguientes:

==	Igual
!=	Distinto
>	Mayor
>=	Mayor o igual
<	Menor
<=	Menor o igual
!	Negación lógica

3) Expresiones de cadenas

Las expresiones entre cadenas permitidas serán las siguientes:

+	Concatenación de cadenas

- Ejemplo de código fuente

A continuación se muestra un código fuente simple escrito en base a las especificaciones anteriores lenguaje:

```
api int sumaEnteros(int e1, int e2);

program Prueba
```

```
{
int c;

c = sumaEnteros(5,2);

if(c > 2)
 {
 c = 1;
 }
 return c;

}
```

Especificación del código generado del compilador (FKIL):

- Código generado

Un programa generado por el compilador estará compuesto por un único módulo con la siguiente estructura:

```
directiva_1
directiva_2
...
instrucción_1
instrucción_2
...
```

Nota: Todas las directivas incluidas en el programa deberán aparecer antes de cualquier instrucción.

- Comentarios

Se podrán incluir comentarios en el código precediendo la línea con el carácter '#'.

- Directivas

Las directivas permitidas en el código generado son las siguientes:

Directiva	Descripción	Valor por defecto
.program	**Indica el nombre del programa**	
.stack	**Indica el tamaño máximo de la pila**	**1024**

.heap	Indica el tamaño máximo de la memoria dinámica	1024
.locals	Indica el número de variables locales utilizadas	0

Ejemplos:

```
.program Prueba
.stack 1500
.heap 1000
.locals 4
```

- Juego de instrucciones

Las instrucciones permitidas en el código generado son las siguientes:

Instrucción	Descripción	Numero Par.	Parámetro
ipush	Coloca una constante entera en la pila	1	Constante
fpush	Coloca una constante real en la pila	1	Constante
spush	Coloca una constante cadena en la pila	1	Constante
bpush	Coloca una constante booleana en la pila	1	Constante
iload	Carga el valor de una variable entera en la pila	1	Numero de variable
fload	Carga el valor de una variable real en la pila	1	Numero de variable
sload	Carga el valor de una variable cadena en la pila	1	Numero de variable
bload	Carga el valor de una variable booleana en la pila	1	Numero de variable
istore	Almacena en una variable entera el primer elemento de la pila	1	Numero de variable

fstore	Almacena en una variable real el primer elemento de la pila	1	Numero de variable
sstore	Almacena en una variable cadena el primer elemento de la pila	1	Numero de variable
bstore	Almacena en una variable booleana el primer elemento de la pila	1	Numero de variable
pop	Elimina el primer elemento de la pila	0	
iadd	Suma de enteros	0	
fadd	Suma de reales	0	
isub	Resta de enteros	0	
fsub	Resta de reales	0	
imul	Producto de enteros	0	
fmul	Producto de reales	0	
idiv	División de enteros	0	
fdiv	División de reales	0	
nneg	Negación numérica	0	
bneg	NO lógico	0	
ncmp	Comparación numérica	0	
bcmp	Comparación booleana	0	
goto	Salto incondicional	1	Etiqueta
ifeq	Salto si igual a 0	1	Etiqueta
ifne	Salto si distinto de 0	1	Etiqueta
iflt	Salto si menor que 0	1	Etiqueta
ifgt	Salto si mayor que 0	1	Etiqueta
ifge	Salto si mayor o igual que 0	1	Etiqueta

ifle	Salto si menor o igual que 0	1	Etiqueta
scmp	Comparación de cadenas	1	Etiqueta
sadd	Concatenación de cadenas	1	
iret	Retorno de entero	0	
fret	Retorno de real	0	
sret	Retorno de cadena	0	
bret	Retorno de booleano	0	
callapi	Llamada a función de la API externa	1	Nombre de función

- Etiquetas

Las etiquetas incluidas en el código, a las cuales podrán hacer referencia todas las instrucciones condicionales se indicarán con un identificador seguido del caracter ':'

Ejemplo:

```
ifeq etiqueta1
ipsuh 1
etiqueta1:
ipush 2
```

- Ejemplo de código generado

A continuación se muestra un programa simple generado por el compilador:

```
#Programa de prueba

.program Prueba
.locals 1

ipush 5
istore 0
ipush 1
fload 0
ipush 2
ncmp
ifgt etiq1
pop
ipush 0
etiq1:
ifeq
etiq2
ipush 1
istore 0
goto etiq3
etiq3:
fload 0
ipush 3
fdiv
fret
```

Conclusiones:

En todo el desarrollo de la tesis el mayor desafío fue realizar el compilador, ya que se debían entender varios conceptos, herramientas y lenguajes hasta ahora nunca vistos en la carrera de ingeniería electrónica. Debimos aprender una herramienta nueva, el ANTLR, siendo la misma de uso actual, a pesar de no estar familiarizados con esta ni con sus posibles lenguajes de programación Java y C#.
A pesar de esto gracias a varios ejemplos encontrados en la red y algunos tutoriales pudimos ir avanzando y entendiendo algunos pasos básicos para así alcanzar los objetivos.
Si bien se puede desarrollar mucho más el tema de compiladores dado el escaso tiempo en la cursada de la materia tratamos de exponer toda la información y desarrollo de ejemplos de la forma más concisa y didáctica posible. Esto implico gran esfuerzo, pero de igual magnitud es la gratificación de haber concluido el trabajo y demás esta decir que sea de gran utilidad para quien desee entrar en el mundo de los compiladores.

Bibliografía:

- Fundamentos de robótica, Barrientos, Peñin, Balaguer y Aracil, Mc Graw Hill 2001.
- Apuntes de robótica cortesía de Samuel Oporto Diaz.
- “Robótica Manipuladores y robots móviles” de Ollero Baturone, Anibal
- www.atmel.com

Compilador:

- http://www.lsi.us.es/~troyano/documentos/guia.pdf “Guía práctica de Antlr 2.7.2 v1.0 Enrique Jose Garcia Cota “
- www.albertnogues.com
- www.sgoliver.net