DOMOL

Asignatura: Teoría de Lenguajes

Curso: **2017/2018**

Autor: Francisco Javier Rojo Martín

DNI: 76042262-F

Fecha: 21 de mayo de 2018

Índice de contenido

1. Introducción

Este proyecto, pretende diseñar e implementar un compilador para el lenguaje DOMOL (*Domotic Model Language*).

El lenguaje DOMOL, permite diseñar sistemas domóticos, basándose en el uso de una máquina de estados, que define el usuario, así como permite definir variables, actuadores,… y otros muchos elementos, que ayudarán a especificar el comportamiento de la máquina de estados para cada estado.

El objetivo del compilador es, a partir de un fichero de entrada, en formato DOMOL (.dml), obtener un fichero de salida .mch, como resultado.

Para realizar el compilador, hemos hecho uso de Bison y Flex, desarrollando un analizador léxico y un analizador sintáctico, para la gramática propia de DOMOL. Además, se ha empleado C/C++ para realizar toda la funcionalidad de la gramática atribuida.

2. Gramática

2.1. Descripción de la gramática

0 $accept: programa $end

1 programa: %empty

2 | zona_variables zona_sensores zona_actuadores zona_estados zona_transiciones zona_comportamiento zona_casos_uso zona_ejemplos

3 zona_variables: VARIABLES '\n' zona_variables_instrucciones

4 zona_variables_instrucciones: %empty

5 | zona_variables_instrucciones operacion_asignacion

6 operacion_asignacion: ID '=' expr '\n'

7 | ID '=' exprLogica '\n'

8 | error '\n'

9 exprLogica: LOGICA

10 | expr S_MENOR expr

11 | expr S_MENORIGUAL expr

12 | expr S_MAYOR expr

13 | expr S_MAYORIGUAL expr

14 | expr S_IGUAL expr

15 | expr S_DISTINTO expr

16 | LOGICA S_IGUAL id_logica

17 | LOGICA S_DISTINTO id_logica

18 | exprLogica S_OR exprLogica

19 | exprLogica S_AND exprLogica

20 | '(' exprLogica ')'

21 | S_NOT exprLogica

22 id_logica: ID

```
23 expr: REAL
24     | NUMERO
25     | ID
26     | expr '+' expr
27     | expr '-' expr
28     | expr '*' expr
29     | expr '/' expr
30     | expr '%' expr
31     | expr '^' expr
32     | '(' expr ')'
33     | '+' expr
34     | '-' expr

35 zona_sensores: SENSORES '\n' zona_sensores_instrucciones

36 zona_sensores_instrucciones: %empty
    37                                  | zona_sensores_instrucciones TIPO_ENTERO identificadores_sensores_enteros '\n'
    38                                  | zona_sensores_instrucciones TIPO_REAL identificadores_sensores_reales '\n'
    39                                  | zona_sensores_instrucciones TIPO_BOOL identificadores_sensores_bool '\n'
40                      | zona_sensores_instrucciones error '\n'

41 identificadores_sensores_enteros: ID
42                          | ID ',' identificadores_sensores_enteros

43 identificadores_sensores_reales: ID
44                          | ID ',' identificadores_sensores_reales

45 identificadores_sensores_bool: ID
46                          | ID ',' identificadores_sensores_bool

47 zona_actuadores: ACTUADORES '\n' zona_actuadores_instrucciones
```

```
48 zona_actuadores_instrucciones: %empty
49                          | zona_actuadores_instrucciones ID '\n'
50                          | zona_actuadores_instrucciones error '\n'

51 zona_estados: ESTADOS '\n' zona_estados_instrucciones

52 zona_estados_instrucciones: %empty
53                        | zona_estados_instrucciones ID '\n'
54                        | zona_estados_instrucciones error '\n'

55 zona_transiciones: TRANSICIONES '\n' zona_transiciones_instrucciones

56 zona_transiciones_instrucciones: %empty
57                             | zona_transiciones_instrucciones ID ':' ID ASIGNACION ID
'\n'
58                           | zona_transiciones_instrucciones error '\n'

59 zona_casos_uso: CASOSUSO '\n' zona_casos_uso_instrucciones
60              | %empty

61 zona_casos_uso_instrucciones: %empty
62                          | zona_casos_uso_instrucciones CASO ID ':' caso_uso2
63                          | zona_casos_uso_instrucciones error '\n'

64 caso_uso2: ID '\n'
65          | ID ';' caso_uso2

                    66 zona_comportamiento: COMPORTAMIENTO '\n'
zona_comportamiento_instrucciones

67 zona_comportamiento_instrucciones: %empty

68 $@1: %empty
```

```
69 zona_comportamiento_instrucciones: zona_comportamiento_instrucciones ID $@1 INICIOLLAVES '\n' comportamiento2 FINLLAVES '\n'
70                          | zona_comportamiento_instrucciones error '\n'

71 $@2: %empty

72 comportamiento2: comportamiento2 SI exprLogica_arbol INICIOLLAVES $@2 '\n' comportamiento2 FINLLAVES '\n' comportamiento3
73              | comportamiento2 ACTIVAR ID '\n'
74              | comportamiento2 DESACTIVAR ID '\n'
75              | comportamiento2 TRANSICION ID '\n'
76              | comportamiento2 operacion_asignacion_arbol
77              | %empty
78              | comportamiento2 error '\n'

79 comportamiento3: %empty

80 $@3: %empty

81 comportamiento3: SINO INICIOLLAVES $@3 '\n' comportamiento2 FINLLAVES '\n'
82              | SINO error

83 zona_ejemplos: EJEMPLOS '\n' zona_ejemplos_instrucciones
84            | %empty

85 zona_ejemplos_instrucciones: %empty
86                          | zona_ejemplos_instrucciones EJEMPLO ID '\n' ejemplos2
87                          | zona_ejemplos_instrucciones error '\n'

88 ejemplos2: %empty
89         | ejemplos2 ID LOGICA '\n'
90         | ejemplos2 ID NUMERO '\n'
91         | ejemplos2 ID REAL '\n'
```

```
92 operacion_asignacion_arbol: ID '=' expr_arbol '\n'
93                           | ID '=' exprLogica_arbol '\n'

94 exprLogica_arbol: LOGICA
95                 | expr_arbol S_MENOR expr_arbol
96                 | expr_arbol S_MENORIGUAL expr_arbol
97                 | expr_arbol S_MAYOR expr_arbol
98                 | expr_arbol S_MAYORIGUAL expr_arbol
99                 | expr_arbol S_IGUAL expr_arbol
100                | expr_arbol S_DISTINTO expr_arbol
101                | exprLogica_arbol S_IGUAL id_logica_arbol
102                | exprLogica_arbol S_DISTINTO id_logica_arbol
103                | exprLogica_arbol S_OR exprLogica_arbol
104                | exprLogica_arbol S_AND exprLogica_arbol
105                | '(' exprLogica_arbol ')'
106                | S_NOT exprLogica_arbol

107 id_logica_arbol: ID

108 expr_arbol: REAL
109           | NUMERO
110           | ID
111           | expr_arbol '+' expr_arbol
112           | expr_arbol '-' expr_arbol
113           | expr_arbol '*' expr_arbol
114           | expr_arbol '/' expr_arbol
115           | expr_arbol '%' expr_arbol
116           | expr_arbol '^' expr_arbol
117           | '(' expr_arbol ')'
118           | '+' expr_arbol
119           | '-' expr_arbol
```

3. Ampliaciones

3.1. Casos de uso

La primera ampliación realizada es la ampliación de los casos de uso, planteada en el enunciado del proyecto.

Para realizar esta ampliación, se han empleado los TAD definidos en 'TADTablaTransiciones.h' y 'TADTipoTabla.h', que habían sido definidos previamente para la funcionalidad básica. Además, se emplean una serie de estructuras básicas, definidas directamente sobre 'expresiones.y'.

```
/*******************************/
/**AMPLIACION DE CASOS DE USO**/

//estructuras necesarias para almacenar los casos de uso;
typedef char nombreEstado[50]; //cada nombre de estado, tendrá, como mucho, 50 caracteres (la misma longitud que tipo_cadena)
struct casoUso{
        nombreEstado nomCaso;
        nombreEstado estados[20];//cada caso de uso, tendrá, como mucho, 20 estados
        nombreEstado transiciones[20];//cada caso de uso, tendrá, como mucho, 20 transiciones
        int numEstados;
        int numTransiciones;
};
typedef casoUso * casoType;
typedef casoType vectorCasosUso[10]; //cada programa, tendrá, como mucho, 10 casos de uso
int numCasos=0;

//variables necesarias para almacenar los casos de uso;
vectorCasosUso vectorCasos;
casoType caso;
bool errorCasoUso;

//procedimiento para mostrar los casos de uso;

void mostrarCasosUso(ofstream & fsal){
        fsal<<"Comprobacion de casos de uso"<<endl;
        fsal<<"==========================="<<endl;
        for(int i=0; i<numCasos; i++){
                casoType casoTemp= vectorCasos[i];
                fsal<<"CASO "<<casoTemp->nomCaso<<endl;
                for(int j=casoTemp->numTransiciones-1; j>=0; j--){
                        fsal<<"    "<<casoTemp->transiciones[j]<<endl;
                }
        }
}
/*******************************/
```

Para hacer uso de esta funcionalidad añadida, solo debemos añadir al **.dml** la zona CASOS DE USO, con el siguiente formato:

CASOS DE USO

CASO NombreCaso: EstadoInicial; Estado1; ... EstadoN; EstadoFinal

ejemplo:

```
CASOS DE USO
CASO Invierno: Inicial; Todo_OFF; C_ON; Todo_OFF; C_ON; Todo_OFF; Final
CASO Primavera: Inicial; Todo_OFF; Final
```

La salida que generamos, es una lista, con los distintos casos que definamos en el **.dml**, incluyendo, para cada caso, las transiciones por las que debemos pasar, si queremos ir por los distintos estados definidos junto al caso en el **.dml**.

Para el ejemplo anterior, encontraríamos una salida como esta:

```
Comprobacion de casos de uso
============================
CASO Invierno
    iniciar
    encender_calef
    apagar_calef
    encender_calef
    apagar_calef
    acabar1
CASO Primavera
    iniciar
    acabar1
```

Hay que tener en cuenta los siguientes aspectos, para hacer uso de esta ampliación:

- La zona de CASOS DE USO es **totalmente opcional**, puede aparecer o no. Si no aparece, el programa funcionará igual de bien, pero no procesará casos de uso.
- En caso de aparecer, esta zona debe definirse, dentro del archivo de entrada, justamente **después de la zona de comportamiento.**
- El **nombre de los casos de uso, no debe estar siendo usado ya** por otra cosa (para nombrar estado, variable, transición,…). Si no se cumple, dará un error semántico.
- El nombre de cada uno de los estados, debe ser el de uno de los **estados definidos previamente**. Si no se cumple, dará un error semántico.
- El **primer estado**, debe ser siempre el estado inicial. Si no se cumple, dará un error semántico.
- El **último estado**, debe ser siempre el estado final. Si no se cumple, dará un error semántico.
- Para **dos estados consecutivos** en el caso de uso, **debe existir alguna transición** que parta del primero al segundo, definida previamente en la zona de transiciones. Si no se cumple, dará un error semántico.

3.2. Ejemplos de funcionamiento (versión alternativa)

La segunda ampliación realizada es la ampliación de los ejemplos de funcionamiento, planteada en el enunciado del proyecto.

Sin embargo, lo primero que hay que tener en cuenta es que, la implementación que yo he realizado, puede ser un poco distinta de la que se esperaba por parte de la profesora de la asignatura. Por tanto, para explicar está ampliación, comenzaré explicando como funciona realmente mi ampliación.

3.2.1. Descripción general de la ampliación.

La ampliación desarrollada, nos permite definir una serie de ejemplos sobre el modelo creado.

Cada ejemplo, contendrá una serie de valores para distintos sensores definidos previamente en la gramática.

Lo que hará el analizador, para cada ejemplo, será crearse una copia de la tabla variables actual, y modificar sobre ella los valores de los sensores que se han definido en el ejemplo (de tal forma que siempre tengamos la tabla original de variables, para copiarla de nuevo y modificarla en cada ejemplo, de forma independiente).

Una vez tengamos estos nuevos valores cargados, se procederá a tratar la zona de comportamiento para el estado inicial, ejecutando las instrucciones de dicho estado. Cuando encontramos una instrucción 'transición', saltamos del estado actual al estado de destino de dicha transición. Cada transición empleada, será la salida del ejemplo.

Esto se repetirá para cada estado, hasta llegar al final (pues no tendrá ninguna transición), o llegar a una instrucción donde necesitamos hacer uso del valor de un sensor, cuyo valor no hemos definido en el apartado EJEMPLO, o no hemos calculado previamente en una de las instrucciones de asignación por las que hemos pasado. En caso de terminar con esta segunda condición, se alertará por consola de que el EJEMPLO NombreEjemplo no ha llegado a un estado final por dicha condición.

Resulta importante añadir que, por la implementación realizada, ejemplos en los que le damos dos valores a un mismo sensor, no tienen sentido, pues el valor que se le guardará realmente al sensor será el último definido.

```
EJEMPLO A
      S verdadero              //transición iniciar
      T1   10.0                //transición encender_calefacción
      S falso                  //transición acabar2
```

En el ejemplo anterior (sacado de 'ejemploAmpl2.dml'), el sensor S tan solo tomará el valor 'falso' para todo el ejemplo (pues, como he explicado, los valores se asignan antes de procesar las instrucciones para el ejemplo, y prevalece el último en caso de dar varios valores a un sensor).

Otra situación que causa problemas, es el hecho de que existan bucles en las transiciones (como ocurre, también, en 'ejemploAmpl2.dml') pues, si no se controla con alguna condición y asignaciones para hacer, llegados a un punto, la condición falsa; el programa entrará en un bucle infinito saltando entre los dos estados. Al final, se dará una Violación del segmento, al llenar el vector del número máximo de transiciones que puede haber para un ejemplo (50 transiciones).

NOTA: En la carpeta 'domol/ejemplos', encontramos un ejemplo creado por mi (*ejemploAmpl2FUNCIONAL.dml*), que funciona correctamente, pues cumple todas las condiciones, y con el que se recomienda que se pruebe esta ampliación.

3.2.2. Estructuras de datos empleadas.

Para esta ampliación, se han debido definir muchas estructuras, además de usar estructuras previamente definidas.

- Se han reutilizado las estructuras definidas en 'TADTablaTransiciones.h' y 'TADTipoTabla.h'.
- Se han creado las estructuras definidas en 'TADArbol.h', 'TADTablaComandos.h' y 'TADEjemplosAux.h'.
- En '*expresiones.y*', se han definido una serie de variables simples, entre las que encontramos un vector de contadores. Todas ellas, se usan con el fin de apoyar o ayudarnos a crear la Tabla de Instrucciones, como se explicará en el *Apartado 4.1* de esta documentación.

3.2.3. Entrada y salida esperada.

Para hacer uso de esta funcionalidad añadida, solo debemos añadir al **.dml** la zona EJEMPLOS, con el siguiente formato:

EJEMPLOS

EJEMPLO NombreEjemplo

NombreSensor1 valorSensor1

NombreSensor2 valorSensor2

...

ejemplo:

```
EJEMPLOS
  EJEMPLO A
        S verdadero             //transición iniciar
        T1   10.0               //transición encender_calefacción
        S falso                 //transición acabar2

EJEMPLO B
        S verdadero             //transición iniciar
        S falso                 //transición acabar1

EJEMPLO C
        S verdadero             //transición iniciar
        T1   15.5               //transición encender_calefacción
        T2   24.7               //transición apagar_calefacción
        S falso                 //transición acabar1
```

La salida que generamos, es una lista, con los distintos casos que definamos en el **.dml**, incluyendo, para cada caso, las transiciones por las que debemos pasar, si queremos ir por los distintos estados definidos junto al caso en el **.dml**.

```
Comprobacion de casos de uso
============================
CASO Invierno
    iniciar
    encender_calef
    apagar_calef
    encender_calef
    apagar_calef
    acabar1
CASO Primavera
    iniciar
    acabar1
```

Para el ejemplo anterior, encontraríamos una salida como esta:

```
SIMULACION DE LOS EJEMPLOS
      EJEMPLO A
          iniciar
          encender_calef
      EJEMPLO B
          iniciar
          encender_calef
      EJEMPLO C
          iniciar
          encender_calef
          apagar_calef
          finalizar
```

Además, en el fichero **.mch**, encontramos también la Tabla de Correspondencia Estado-Instrucciones (que nos permite saber que instrucciones son de que estado) y la Tabla de Comandos (que nos permite ver codificadas las instrucciones de la zona de comportamiento).

```
TABLA DE CORRESPONDENCIA ESTADO-INSTRUCCIONES
=============================================
ESTADO            PRIMERA INSTRUCCION       NUMERO INSTRUCCIONES
=====================================================================
Inicial             0                           5
Final             5                           3
Final_Intermedio            21                          3
Todo_OFF             8                          5
C_ON            13                          4
AC_ON             17                          4
```

```
TABLA DE COMANDOS
=========================================
NUM. INSTRUCCION       COMANDO       PARAM1                                                  PARAM2
=================================================================================================
0       SI_SIMPLE (4)        (falso==S)                                              4-0
1       ACTIVAR (1)        SYSTEM                                          NULO
2       DESACTIVAR (2)        Calefaccion                                          NULO
3       DESACTIVAR (2)        A_C                                          NULO
4       TRANSICION (3)        iniciar                                          NULO
5       DESACTIVAR (2)        Calefaccion                                          NULO
6       DESACTIVAR (2)        A_C                                          NULO
7       DESACTIVAR (2)        SYSTEM                                          NULO
8       ACTIVAR (1)        Calefaccion                                          NULO
9       TRANSICION (3)        encender_calef                                          NULO
10       ACTIVAR (1)        A_C                                          NULO
11       TRANSICION (3)        encender_airea                                          NULO
12       TRANSICION (3)        acabar1                                          NULO
13       SI_SIMPLE (4)        T2>temp_verano                                          2-0
14       DESACTIVAR (2)        Calefaccion                                          NULO
15       TRANSICION (3)        apagar_calef                                          NULO
16       TRANSICION (3)        acabar2                                          NULO
17       SI_SIMPLE (4)        T1<temp_invierno                                          2-0
18       DESACTIVAR (2)        A_C                                          NULO
19       TRANSICION (3)        apagar_airea                                          NULO
20       TRANSICION (3)        acabar3                                          NULO
21       DESACTIVAR (2)        A_C                                          NULO
22       DESACTIVAR (2)        Calefaccion                                          NULO
23       TRANSICION (3)        finalizar                                          NULO
```

Así mismo, se podrá mostrar, por casos como el comentado en el primer apartado de esta ampliación, un mensaje similar a este por **consola**:

```
ecto/PRYUEBA/prueba para entregar/domol$ ./domol ejemplos/ejemploAmpl2FUNCIONAL.dml
NOTA: EJEMPLO A no ha podido llegar a estado final. Hay sensores sin inicializar
NOTA: EJEMPLO B no ha podido llegar a estado final. Hay sensores sin inicializar
```

3.2.4. Aspectos a tener en cuenta.

Hay que tener en cuenta los siguientes aspectos, para hacer uso de esta ampliación:

- La zona de EJEMPLOS es **totalmente opcional**, puede aparecer o no. Si no aparece, el programa funcionará igual de bien, pero no procesarán ejemplos.
- En caso de aparecer, esta zona debe definirse, dentro del archivo de entrada, **como la última de las zonas.**
- El **nombre de los ejemplos** puede estar siendo usado ya, o ser usado por varios ejemplos, pues no causará conflicto alguno.
- El nombre de cada uno de los sensores, debe ser el de uno de los **sensores definidos previamente**. Si no se cumple, dará un error semántico.
- En caso de darle **varios valores al mismo sensor**, prevalecerá unicamente el último valor.
- El **valor dado a un sensor, debe ser apropiado al tipo del sensor**. Si no se cumple, dará un error semántico.
- Debe asegurarse que **no existen bucles en las transiciones** de la maquina de estados; **o bien, que dichos bucles son controlados** con instrucciones ‘si’ y asignaciones, de tal forma que no se provoque un bucle infinito de saltos entre dos estados (esto se explicó en el *Apartado 3.2.1*).
- En caso de **encontrarnos con sensores sin un valor asignado**, durante la ejecución de un ejemplo, dicha ejecución se parará en ese punto y se mostrará un mensaje de pantalla que indica la situación.

4. Estructuras de datos auxiliares

Para aquellas variables de datos primitivos, o estructuras muy simples, no se ha definido un nuevo TAD con tipos nuevos, ni nada similar, sino que se han especificado directamente en el analizador sintáctico.

4.1. Analizador sintáctico

4.1.1. Variables para versión básica

Para llevar a cabo la versión básica del proyecto, hemos definido las siguientes variables:

```
//banderas y variables para almacenar los datos.
int banderaTipo;
int errorModulo;
int errorDefinida;
int estadoComportamiento;
tipo_tabla tabla;
tTrans tTransicion;
tipo_datoTS dato;

//contadores para asignar ids a estados y transiciones.
int ultEstAsign=-1;
int ultTranAsign=-1;

//flujo para escribir los resultados en un fichero
ofstream fsal;
```

Para el manejo de dichas variables, se han definido también una serie de procedimientos:

```
int getTipo(){

        return banderaTipo;
}

void setTipo(int tipo){
        banderaTipo=tipo;
}

void setErrorModulo(int error){
        errorModulo=error;
}

void setErrorDefinida(int error){
        errorDefinida=error;
}
```

Podemos explicar, para que vale cada variable:

- banderaTipo: toma el valor 0 si estamos en una operación con enteros, y 1 si es con reales.

- ErrorModulo: pasa a valer 1 cuando se intenta hacer el módulo de un real, o con un real. Toma el valor 0 al principio de una expresión aritmética.
- ErrorDefinida: pasa a valer 1 cuando se da un error con alguna variable empleada en una expresión. Al principio de cada expresión, vale siempre 0.
- estadoComportamiento: Almacena el id del estado que estamos tratando en ese momento en la zona de comportamiento.
- Tabla: Almacena la Tabla de Símbolos.
- Ttransicion: Almacena la Tabla de Transiciones.
- Dato: sirve para leer sobre ella un dato de la Tabla de Símbolos.
- UltEstAsign: almacena el último id que se le asignó a un estado.
- UltTranAsign: almacena el último id que se le asignó a una transición.
- Fsal: flujo de salida asociado al fichero *.mch*.

4.1.2. Variables para ampliación de casos de uso

Para los casos de uso, se ha definido una simple estructura, que almacenará la información de cada caso de uso, así como dos tipos para crear un vector de punteros a casos de uso:

```
//estructuras necesarias para almacenar los casos de uso;
typedef char nombreEstado[50]; //cada nombre de estado, tendrá, como mucho, 50 caracteres (la misma longitud que tipo_cadena)
struct casoUso{
        nombreEstado nomCaso;   //nombre del caso de uso
        nombreEstado estados[20];//cada caso de uso, tendrá, como mucho, 20 estados
        nombreEstado transiciones[20];//cada caso de uso, tendrá, como mucho, 20 transiciones
        int numEstados;
        int numTransiciones;
};
typedef casoUso * casoType;
typedef casoType vectorCasosUso[10]; //cada programa, tendrá, como mucho, 10 casos de uso
int numCasos=0;

//variables necesarias para almacenar los casos de uso;
vectorCasosUso vectorCasos;
casoType caso;
bool errorCasoUso;
```

La variable *errorCasoUso*, nos permite saber si ha habido algún error semántico en el caso de uso, sabiendo así si almacenarlo en el vector o descartarlo.

También, se ha definido una operación para mostrar los casos de uso:

```
//procedimiento para mostrar los casos de uso;

void mostrarCasosUso(ofstream & fsal){
        fsal<<"Comprobacion de casos de uso"<<endl;
        fsal<<"============================"<<endl;
        for(int i=0; i<numCasos; i++){
                casoType casoTemp= vectorCasos[i];
                fsal<<"CASO "<<casoTemp->nomCaso<<endl;
                for(int j=casoTemp->numTransiciones-1; j>=0; j--){
                        fsal<<"    "<<casoTemp->transiciones[j]<<endl;
                }
        }
}
```

4.1.3. Variables para ampliación de ejemplos

Para la ampliación de ejemplos, se han debido crear un número mayor de variables y TADs. Los TADs que se implementaron para esta ampliación, se puede ver en el apartado correspondiente de la documentación, dedicado a explicar la misma.

```
//variables necesarias para almacenar los comandos;
tablaComandos tComandos; //tabla de comandos
tipo_instruccion * instruccion; //variable para manejar las instrucciones a almacenar
int ultInstrAsign; // variable para asignar el numero de instruccion a cada instruccion
int ultInstEstadoAnterior; //variable para guardar el numero de la ultima instruccion asignada en el estado anterior

//variables para almacenar la informacion de ejemplos
vectorSensores vSensores;
vectorEjemplos vEjemplos;
EjemploStruct * ejemplo;
```

Además de estas variables, se crearon una serie de variables para poder contar el número de instrucciones de un ‘si’, ‘sino’ y estado (con el fin de incluir esta información en la Tabla de Comandos o de correspondencia Estado-Instrucciones). Dichas variables, son:

```
/*CONTADORES DE INSTRUCCIONES DE CADA ZONA*/
bool banderaElse;

int contadores [20]; //podremos anidar, hasta 20 zonas

int numContElse; //para contar, cuantos de los contadores, estan asociados a ELSEs (que no crearán una nueva instruccion)
```

En este caso, considero más interesante explicar el funcionamiento que se le da a las mismas, por lo que lo haré:

- banderaElse: nos permite saber cuando estamos tratando una zona por un else (para poder decidir, a la hora de tratar el ‘si’, si la operación será un SI_SIMPLE o un SI_NO).
- Contadores: es la variable realmente interesante, pues contendrá el número de instrucciones de cada zona, representando la anidación de estas por el índice. Cada vez que entramos en una nueva zona, por ‘si’ o ‘sino’, se pone un contador más a un valor válido (pues, inicialmente, los 20 estarán a -1). Cuando incrementamos, incrementamos todos los contadores válidos. Cuando terminamos de procesar una zona, además, sacaremos el contador válido con mayor índice. Dicho contador nos valdrá para saber el número de instrucciones que debemos detallar en el ‘si’ o ‘sino’, o en el estado (el contador 0 siempre será el del estado a tratar, en ese momento). Igualmente, el contador nos valdrá para saber que valor tomará el número de instrucción (pues, las instrucciones de condición se tratarán después de las que contienen dentro, pero el número de instrucción debe ser menor).
- NumContElse: sirve para complementar lo último comentado para la variable ‘contadores’.

Para hacer uso de estas variables, se han creado una serie de procedimientos, que las usan:

```
void inicializarContadoresZonas(){
        for(int i=0; i<20; i++){
                contadores[i]=-1;
        }
}

void nuevoContador(){
        int i=0;
        bool creado=false;
        while(!creado && i<20){
                if(contadores[i]==-1){
                        contadores[i]=0;
                        creado=true;
                }
                i++;
        }
}

int getUltimoContador(){
        bool encontrado=false;
        int contador=-1;

        for(int i=19; i>=0; i--){
                if(!encontrado){
                        if(contadores[i]!=-1){
                                contador=contadores[i];
                                encontrado=true;
                        }
                }
        }
        return contador;
}
```

```
void eliminarUltimoContador(){
        bool eliminado=false;

        for(int i=19; i>=0; i--){
                if(!eliminado){
                        if(contadores[i]!=-1){
                                contadores[i]=-1;
                                eliminado=true;
                        }
                }
        }
}

void incrementarContadores(){
        for(int i=0; i<20; i++){
                if(contadores[i]!=-1){
                        contadores[i]++;
                }
        }
}

int cuantosContadores(){
        int i=0;
        while(i<20 && contadores[i]!=-1){
                i++;
        }
        return i;
}
```

Con lo explicado, resulta fácil saber cual es la misión de cada procedimiento, por lo que no volveré a explicarlo.

4.2. Librería 1 : TADTipoTabla.h

Esta librería, contiene todas aquellas estructuras y operaciones necesarias para generar la Tabla de Símbolos.

4.2.1. Estructuras

```
typedef char tipo_cadena[50];
union tipo_valor{
    int valor_entero;
    float valor_real;
    bool valor_logico;
};
struct tipo_datoTS{
    tipo_cadena nombre;
    int tipo;
    tipo_valor valor;
    bool inicializado;
};

struct nodo{
    nodo * next;
    tipo_datoTS variable;
};

typedef nodo * tipo_tabla;
```

4.2.2. Operaciones

```
/* DESC:   Inicializa la lista, si no lo está
 * PRE:    Lista apuntando a basura
 * POST:   Lista apuntando a NULL
 * PARAM:  tabla: tabla de símbolos
 * COMP:   O(1)
 */
void  inicializar (tipo_tabla &tabla);

/* DESC:   Consulta si la lista esta vacia
 * PRE:    Lista creada
 * POST:   -
 * PARAM:  tabla: tabla de símbolos
 * RET:    TRUE:  lista vacia
 *         FALSE: lista no vacia
 * COMP:   O(1)
 */
bool  estaVacia (tipo_tabla tabla);

/* DESC:   Copia la tabla 'tOriginal' en la tabla 'tCopia'
 * PRE:    tOriginal inicilizada
 * POST:   -
 * PARAM:  tOriginal -> tabla original, con datos
           tCopia -> tabla sobre la que copiaremos los datos
 * RET:
 * COMP:   O(n)
 */
void  copiarTabla (tipo_tabla & tCopia, tipo_tabla tOriginal);

/* DESC:   Inserta un elemento en la lista, al final de la misma
 * PRE:    Lista creada.
 * POST:   Lista con un elemento mas al final de la misma
 * PARAM:  tabla: tabla de símbolos
          E: dato -> Elemento a insertar
 * RET:    -
 * COMP:   O(n)
 */
void  insertar (tipo_tabla & tabla,tipo_datoTS dato);

/* DESC:   Comprueba si existe un elemento en la lista
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:   tabla: tabla de símbolos
           E: nombre -> nombre del elemento a buscar
 * RET:          TRUE: existe una variable con el nombre especificado
                 FALSE: NO existe una variable con el nombre especificado
 * COMP:   O(n)
 */
bool  existe (tipo_tabla tabla,tipo_cadena nombre);
```

```
/* DESC:   Modifica el elemento de la lista con el mismo nombre que 'dato.nombre'
 * PRE:    Lista creada.
 * POST:   Lista con un elemento modificado
 * PARAM:  E: dato -> Elemento a modificar
 * RET:         TRUE: Se ha podido modificar el elemento
 *                    FALSE: No existe el elemento, y no se ha podido modificar
 * COMP:   O(n)
 */
bool  modificar (tipo_tabla tabla,tipo_datoTS dato);

/* DESC:   Devuelve el elemento de la lista con nombre igual al pasado por parametro
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:   tabla: tabla de símbolos
          S: dato ->  la variable a devolver
          E: nombre -> nombre del dato a devolver
 * RET:    bool, a true si existe el dato. Si no existe, a false.
 * COMP:   O(n)
 */
bool consultar (tipo_tabla tabla, tipo_cadena nombre, tipo_datoTS & dato);

/* DESC:   Devuelve el nombre de la transicion con id pasado por parametro
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:   tabla: tabla de símbolos
          S: dato ->  la variable a devolver
          E: id -> id de la transicion a consultar
 * RET:    bool, a true si existe la transicion. Si no existe, a false.
 * COMP:   O(n)
 */
bool consultarTransicion (tipo_tabla tabla, int id, tipo_datoTS & dato);

/* DESC:   Devuelve un array, donde, el valor de la posicion, es el valor con el que se codifica
el estado cuyo nombre almacena esa posicion.
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:   tabla: tabla de símbolos
          S: estados -> array de los estados
          S: numEstados -> numero de estados que tiene el array
 * RET:    -
 * COMP:   O(n)
 */
void obtenerCodificacionEstados (tipo_tabla tabla, tipo_cadena estados[], int & numEstados);
```

```
/* DESC:   Devuelve un array, donde, el valor de la posicion, es el valor con el que se codifica la transicion cuyo nombre almacena esa posicion.
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:   tabla: tabla de símbolos
            S: transiciones -> array de las transiciones
            S: numTransiciones -> numero de transiciones que tiene el array
 * RET:    -
 * COMP:   O(n)
 */
void obtenerCodificacionTransiciones (tipo_tabla tabla, tipo_cadena transiciones[], int & numTransiciones);


/* DESC:   Borra el elemento de la lista con el nombre pasado por parametro
 * PRE:    Lista creada.
 * POST:   Num. elementos en la lista disminuye en 1 si se puede borrar.
 * PARAM:  tabla: tabla de símbolos
           E: nombre -> nombre del dato a eliminar
 * RET:    -
 * COMP:   O(n)
 */
void borrar (tipo_tabla &tabla, tipo_cadena nombre);

/* DESC:    Muestra los datos de todas las variables almacenadas en la tabla por salida estandar,
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:  tabla: tabla de símbolos
 * RET:    -
 * COMP:   O(n)
 */
void mostrar (tipo_tabla tabla);

/* DESC:   Muestra los datos de todas las variables almacenadas en la tabla por el flujo de salida 'fsal'
 * PRE:    Lista creada.
 * POST:   -
 * PARAM: tabla: tabla de símbolos
          fsal -> flujo de salida asociado al fichero donde debe mostrarse la información
 * RET:    -
 * COMP:   O(n)
 */
void mostrar (tipo_tabla tabla, ofstream & fsal);

/* DESC:   Libera la memoria ocupada por los nodos de la lista
 * PRE:    Lista creada.
 * POST:   Num. elementos en la lista = 0. Lista vacia == TRUE
 * PARAM:  tabla: tabla de símbolos
 * RET:    -
 * COMP:   O(n)
 */
void borrarMemoria(tipo_tabla &tabla);
```

4.3. Librería 2: TADTablaTransiciones.h

Esta librería, contiene todas aquellas estructuras y operaciones necesarias para generar la Tabla de Transiciones.

4.3.1. Estructuras

```
typedef char tipo_cadena[50];

struct tablaTransiciones{
    int transiciones [TAMTABLA][TAMTABLA]; //filas == estados de entrada; columnas == estados de salida;
};

typedef tablaTransiciones * tTrans;
```

4.3.2. Operaciones

/ DESC: Inicializa todas las transiciones a NO_DEFINIDO*
* PRE: matriz apuntando a basura apuntando a basura*
* POST: matriz con valores NO_DEFINIDO.*
* PARAM: -*
* COMP: O(n^2)*
/
void inicializar (tTrans & tabla);

/ DESC: añade un nuevo valor a la transicion [estadoEntrada][estadoSalida]*
* PRE: -*
* POST: Devuelve FALSE si ya tenía un valor asignado la transicion (no se ha podido completar*
la acción, por tanto), y TRUE si no lo tenía.
* PARAM: estadoEntrada: valor de la fila de la matriz de transiciones (estado de entrada)*
estadoSalida: valor de la columna de la matriz de transiciones (estado de salida)
transicion: valor asignado a la transicion en cuestion. Será el valor que tome [estadoEntrada][estadoSalida]
* RET: Devuelve FALSE si ya tenía un valor asignado la transicion (no se ha podido completar*
la acción, por tanto), y TRUE si no lo tenía.
* COMP: O(1)*
/
bool nuevaTransicion(tTrans tabla, int estadoEntrada, int estadoSalida, int transicion);

/ DESC: devuelve el valor del estado de entrada y salida, para una transicion. Si no se encuentra la transacion, se devuelve false, si se encuentra, se devuelve true,*
* PRE: -*
* POST: -*
* PARAM: SALIDA estadoEntrada: valor de la fila de la matriz de transiciones (estado de entrada)*
SALIDA estadoSalida: valor de la columna de la matriz de transiciones (estado de salida)
transicion: valor que habrá en la posicion [i][j], donde i es el valor del estado salida buscado.
* COMP: O(n)*
/
bool estadosTransicion(tTrans tabla, int transicion, int & estadoEntrada, int & estadoSalida);

```
/* DESC:   devuelve el valor de la transicion que va de estadoEntrada a estadoSalida
* PRE:    -
* POST:   Devuelve el valor  de la transicion, y -1 si no lo tiene.
* PARAM:  estadoEntrada: valor de la fila de la matriz de transiciones (estado de entrada)
          estadoSalida: valor de la columna de la matriz de transiciones (estado de salida)
* RET: valor que habrá en la posicion [i][j], donde i es el valor del estadoEntrada buscado y j de estadoSalida.
* COMP:   O(1)
*/
int consultarTransicion(tTrans tabla, int estadoEntrada, int estadoSalida);

/* DESC:   devuelve el valor del estado de salida, para una transicion y una entrada. Si no hay estado de salida, devuelve -1, si no, el valor del estado.
* PRE:    -
* POST:   Devuelve el valor del estado de salida si tiene un valor asignado la transicion, y -1 si no lo tiene.
* PARAM:   estadoEntrada: valor de la fila de la matriz de transiciones (estado de entrada)
           transicion: valor que habrá en la posicion [estadoEntrada][i], donde i es el valor del estado salida buscado.
* COMP:   O(n)
*/
int estadoSalida(tTrans tabla, int estadoEntrada, int transicion);

/* DESC:    Muestra la matriz de transiciones, donde se codifican las transaciones entre estados, por el flujo de salida 'fsal'
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:  fsal -> flujo de salida asociado al fichero donde debe mostrarse la información
 * RET:    -
 * COMP:   O(n²)
 */
void mostrarTransiciones (tTrans tabla, ofstream & fsal);

/* DESC:   Muestra la matriz de transiciones, donde se codifican las transaciones entre
     estados, por el flujo de salida 'fsal' (hasta numEstados)
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:  fsal -> flujo de salida asociado al fichero donde debe mostrarse la información
 * RET:    -
 * COMP:   O(n²)
 */
void mostrarTransiciones (tTrans tabla, int numEstados, ofstream & fsal);
```

```
/* DESC:   Muestra la matriz de transiciones, donde se codifican las transaciones entre
      estados, por el flujo de salida 'fsal' (hasta numEstados). Cambia los ids por los nombres de
estados y transiciones
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:  fsal -> flujo de salida asociado al fichero donde debe mostrarse la información
 * RET:    -
 * COMP:   O(n²)
 */
void mostrarTransiciones (tTrans tabla, int numEstados, ofstream & fsal, tipo_cadena estados
[], tipo_cadena transiciones []);

/* DESC:     comprueba si hay estados inaccesibles. Usa internamente un algoritmo de
backtracking.
 * PRE:    Lista creada.
 * POST:   -
 * PARAM: numEstados -> cuantos estados validos tiene la matriz
 * RET:    TRUE si existen estados inaccesibles y FALSE si no existen,
 * COMP:   O(n²)
 */
bool estadosInaccesibles (tTrans tabla, int numEstados);

/* DESC:     comprueba si hay estados muertos.  Usa internamente un algoritmo de
backtracking.
 * PRE:    Lista creada.
 * POST:   -
 * PARAM:   numEstados -> cuantos estados validos tiene la matriz
 * RET:    TRUE si existen estados muertos y FALSE si no existen,
 * COMP:   O(n²)
 */
bool estadosMuertos (tTrans tabla, int numEstados);
```

4.4. Librería 3: TADTablaComandos.h

Esta librería, contiene todas aquellas estructuras y operaciones necesarias para generar la Tabla de Comandos, así como la Tabla de Correspondencia Estado-Instrucciones. Esto, es usado exclusivamente en la **ampliación 2.**

4.4.1. Estructuras

```
typedef char tipo_cadena[50];


/*************************************/
/****** TABLA CORRESPONDENCIA **********/
/*************************************/

/*Estructura que almacenará la correspondencia entre las intrucciones y
  una determinada zona (estado, if, else, ...)*/
struct instruccionesZona{
  int primeraInstruccion; //numero de la primera instruccion para la zona
  int numInstrucciones; //numero de instrucciones que tendrá dicha zona
};

typedef  instruccionesZona tablaCorrespondencia [NUM_ESTADOS_MAX];

////////////////////////////////////////////
////////////////////////////////////////////
```

```
/*************************************/
/****** TABLA INSTRUCCIONES   **********/
/*************************************/

/*Estructura que almacena la informacion de distinto tipo que puede tener un
  parametro de una instruccion*/
struct tipo_parametro{
  arbolBinario expresion; //expresion aritmetica o logica
  instruccionesZona instrucciones1; //INSTRUCCIONES PARA SI (DE SI_NO) O SI_SIMPLE
  instruccionesZona instrucciones2; //INSTRUCCIONES PARA NO (DE SI_NO)
  tipo_cadena nombre; //nombre de una variable, sensor,...
};

/*Estructura que almacena la informacion para una instruccion*/
struct tipo_instruccion{
    int numInstruccion;
    int comando;
    tipo_parametro param1;
    tipo_parametro param2;
};

struct nodo_Instruccion{
  nodo_Instruccion * next;
  tipo_instruccion instruccion;
  nodo_Instruccion * before;
};
typedef nodo_Instruccion * tablaInstrucciones;

////////////////////////////////////////////
////////////////////////////////////////////
```

```
/*************************************************************************/
/****** COMANDOS = TABLA CORRESPONDENCIA + TABLA INSTRUCCIONES   **********/
/*************************************************************************/

/*Estructura principal, que almacenará el vector de correspondencia
  instrucciones-estado, y la tabla donde estan codificadas las instrucciones*/
struct tablaComandosEstatico{
  tablaCorrespondencia insEst;
  tablaInstrucciones instrucciones;
};
typedef tablaComandosEstatico * tablaComandos;
```

4.4.2. Operaciones

```
/* DESC:   Inicializa todas las estructuras necesarias para la tabla de comandos.
 * PRE:    -
 * POST:   Estructuras inicializadas
 * PARAM:  -
 * RET:    -
 * COMP:   O(n)
 */
void  inicializar (tablaComandos & tComandos);

/* DESC:     Nos devuelve 'true' si no hay ninguna instruccion en la tablaInstrucciones 'instrucciones'.
No es necesario comprobar 'insEst', pues, si la tabla de instrucciones está vacía, esta informacion carece
de importancia.
* PRE:    tabla inicilizada
* POST:   -
* PARAM:  -
* RET:    -
* COMP:   O(1)
*/
bool  estaVacia(tablaComandos tComandos);

/* DESC:   Inserta una nueva instrucciona en la tablaInstrucciones de 'tComandos'.
* PRE:    tabla inicializaada
* POST:   estructura con una instruccion más al final
* PARAM:  instruccion-> instruccion a insertar
* RET:    -
* COMP:   O(n)
*/
void  insertarInstruccion(tablaComandos tComandos, tipo_instruccion intruccion);

/* DESC:    Inserta una nueva instrucciona en la tablaInstrucciones de 'tComandos'. Se usa para insertar
   la instruccion en la zona adecuada, en lugar de al final. Para instrucciones: SI_SIMPLE, SI_NO
  * PRE:    tabla inicializaada
     * POST:     estructura con una instrucción más en la posicion adecuada (según numInstruccion)
  * PARAM:  instruccionZona -> instruccion a insertar
  * RET:    -
  * COMP:   O(n)
  */
void insertarInstruccionZona(tablaComandos tComandos, tipo_instruccion instruccionZona);
```

```
/* DESC:   Modifica los parametros de 'instruccionesZona', para la celda 'estado' del vector 'insEst'
* PRE:    tabla inicializaada
* POST:   valor de la tabla de correspondencia para un estado modificado
* PARAM:  estado -> estado sobre el que modificar
          primeraInstruccion -> numero de la primera instruccion del estados
          numInstrucciones -> numero de instrucciones del estado
* RET:    -
* COMP:   O(1)
*/
void actualizarInstrEstado(tablaComandos tComandos, int estado, int primeraInstruccion, int numInstrucciones);

/* DESC:   Muestra los datos de todas las estructuras almacenadas por el flujo de salida 'fsal'
   (Tabla de comandos y de correspondencia Estado-instrucciones)
 * PRE:    tabla inicializada
 * POST:   -
 * PARAM:   fsal -> flujo de salida asociado al fichero donde debe mostrarse la información
            numEstados -> numero de estados
            estados -> array con la codificacion de los estados y su nombre
 * RET:    -
 * COMP:   O(n)
 */
void mostrarComandos (tablaComandos tComandos, ofstream & fsal, int numEstados, tipo_cadena estados []);


/* DESC:   trata un estado para un ejemplo, empezando por el estado
 * PRE:    tabla inicializada
 * POST:   -
 * PARAM:   tTransiciones -> tabla de transiciones
            variables -> tabla de simbolos (con los valores de sensores actualizados)
            numEstado -> numero de estado a tratar
            sensoresEjemplo -> vector con los valores para los sensores en el ejemplo
            ejemplo -> estructura para almacenar el resultado del ejemplo
 * RET:    -
 * COMP:   O(n)
 */
bool tratarEstado (tablaComandos tComandos, tTrans tTransiciones,  int numEstado, tipo_tabla variables, vectorSensores sensoresEjemplo, EjemploPuntero ejemplo);

 /* DESC:   trata un ejemplo, empezando por el estado 0
  * PRE:    tabla inicializada
  * POST:   -
  * PARAM:   tTransiciones -> tabla de transiciones
             variables -> tabla de simbolos
             sensoresEjemplo -> vector con los valores para los sensores en el ejemplo
             ejemplo -> estructura para almacenar el resultado del ejemplo
  * RET:    -
  * COMP:   O(n)
  */
bool tratarEjemplo (tablaComandos tComandos, tTrans tTransiciones,  tipo_tabla variables, vectorSensores sensoresEjemplo, EjemploPuntero ejemplo);
```

4.5. Librería 4: TADArbol.h

Esta librería, contiene todas aquellas estructuras y operaciones necesarias para generar los árboles (binarios) de expresiones, en la zona de comportamiento. Esto, es usado exclusivamente en la **ampliación 2.**

4.5.1. Estructuras

Se han definido, en primer lugar, una serie de valores para asociarlos a las distintas operaciones de forma más sencilla.

```
/*SI EL NODO ES UNA HOJA, EL VALOR SERÁ TOMADO COMO TAL*/
/*SI EL NODO NO ES UNA HOJA, EL VALOR HARÁ APLICAR LA OPERACION APROPIADA*/
#define SUMA            0
#define RESTA           1
#define MULTIPLICACION  2
#define DIVISION        3
#define MODULO          4
#define POTENCIA        5
#define MAS_UNARIO      6
#define MENOS_UNARIO    7

#define PAR_IZQ        10
#define PAR_DER        11



#define MENOR          20
#define MENOR_IGUAL    21
#define MAYOR          22
#define MAYOR_IGUAL    23
#define IGUAL          24
#define DISTINTO       25
#define OR             26
#define AND            27
#define NOT            28
```

Después, se han definido las estructuras:

```
typedef char tipo_cadena[50];

struct tipo_valorArbol{
    float valor;
    bool valorLogico;
    tipo_cadena nombreID;
};

struct nodoArbol{
  nodoArbol * hijoIzq;
  tipo_valorArbol dato;
  bool esID;
  bool esLogico;
  nodoArbol * hijoDer;
};

typedef nodoArbol * arbolBinario;
```

4.5.2. Operaciones

```
/* DESC:   Crea un arbol juntando los dos arboles pasados, y poniendo en el nodo
   el valor entero pasado.
 * PRE:    -
 * POST:   nodo creado
 * PARAM:   hijoIzq -> arbol que ocupará la posicion de hijo izquierdo
          hijoDer -> arbol que ocupara la posicion de hijo derecho
          valor -> valor que tomará el nuevo nodo
 * RET:    arbol resultante
 * COMP:   O(1)
 */
arbolBinario crearEntero(arbolBinario hijoIzq, int valor, arbolBinario hijoDer);

/* DESC:   Crea un arbol juntando los dos arboles pasados, y poniendo en el nodo
   el valor real pasado.
 * PRE:    -
 * POST:   nodo creado
 * PARAM:   hijoIzq -> arbol que ocupará la posicion de hijo izquierdo
          hijoDer -> arbol que ocupara la posicion de hijo derecho
          valor -> valor que tomará el nuevo nodo
 * RET:    arbol resultante
 * COMP:   O(1)
 */
arbolBinario crearReal(arbolBinario hijoIzq, float valor, arbolBinario hijoDer);

/* DESC:   Crea un arbol juntando los dos arboles pasados, y poniendo en el nodo
   el nombre de la variable pasado.
 * PRE:    -
 * POST:   nodo creado
 * PARAM:   hijoIzq -> arbol que ocupará la posicion de hijo izquierdo
          hijoDer -> arbol que ocupara la posicion de hijo derecho
          id -> nombre de la variable
 * RET:    arbol resultante
 * COMP:   O(1)
 */
arbolBinario crearCadena(arbolBinario hijoIzq, tipo_cadena id, arbolBinario hijoDer );

/* DESC:   Crea un arbol juntando los dos arboles pasados, y poniendo en el nodo
   el valor logico pasado.
 * PRE:    -
 * POST:   nodo creado
 * PARAM:   hijoIzq -> arbol que ocupará la posicion de hijo izquierdo
          hijoDer -> arbol que ocupara la posicion de hijo derecho
          logico -> valor que tomará el nuevo nodo
 * RET:    arbol resultante
 * COMP:   O(1)
 */
arbolBinario crearBool(arbolBinario hijoIzq, bool logico, arbolBinario hijoDer );
```

```
/* DESC:   Comprueba si un arbol es una hoja (sus hijos son NULL)
 * PRE:    arbol creado, no a NULL
 * POST:   -
 * PARAM:
 * RET:    True si es hoja y false si no lo es
 * COMP:   O(1)
 */
bool esHoja(arbolBinario arbol);

/* DESC:   Muestra el arbol por el fichero de salida
 * PRE:    arbol creado, no a NULL
 * POST:   -
 * PARAM: fsal-> flujo de salida asociado al fichero para mostrar el arbol
 * RET:   -
 * COMP:   O(n)
 */
void mostrarArbol(arbolBinario arbol, ofstream & fsal);

/* DESC:   Muestra el arbol por consola
 * PRE:    arbol creado, no a NULL
 * POST:   -
 * PARAM:
 * RET:   -
 * COMP:   O(n)
 */
void mostrarArbol(arbolBinario arbol);

/* DESC:   Obtiene el resultado de un arbol con valor lógico
 * PRE:    arbol creado, no a NULL
 * POST:   -
 * PARAM: variables -> tabla de simbolos
          expresion -> arbol a resolver
          S: correcto -> indica si el arbol, o alguno de los arboles hijo, no
            se ha podido resolver correctamente, por encontrarse con una variable
            sin inicializar o un árbol mal formado.
 * RET:   el valor logico resultante de resolver el arbol
 * COMP:   O(n)
 */
bool resolverArbolLogico(tipo_tabla variables, arbolBinario expresion, bool &correcto);

/* DESC:   Obtiene el resultado de un arbol con valor numérico
 * PRE:    arbol creado, no a NULL
 * POST:   -
 * PARAM: variables -> tabla de simbolos
          expresion -> arbol a resolver
          S: correcto -> indica si el arbol, o alguno de los arboles hijo, no
            se ha podido resolver correctamente, por encontrarse con una variable
            sin inicializar o un árbol mal formado.
 * RET:   el valor numérico resultante de resolver el arbol
 * COMP:   O(n)
 */
float resolverArbolFloat(tipo_tabla variables, arbolBinario expresion, bool &correcto);
```

4.6. Librería 5: TADEjemplosAux.h

Esta librería, contiene todas aquellas estructuras y operaciones necesarias para procesar los distintos ejemplos. Podríamos haberla dividido en dos librerías independientes, una con el TAD para almacenar la solución de los ejemplos, y otra con el TAD para almacenar los sensores definidos en cada ejemplo. Sin embargo, como no se deberían usar los dos TAD por separado, se ha optado por ponerlos juntos en la misma librería.

Esto, es usado exclusivamente en la **ampliación 2.**

4.6.1. Estructuras

En primer lugar, encontramos las estructuras definidas para **almacenar los Ejemplos**:

```
struct EjemploStruct{
  tipo_cadena transicciones[50];
  int numTransicciones;
  tipo_cadena nomEjemplo;
};
typedef EjemploStruct * EjemploPuntero;

struct vectorEjemplosEstatico{
  EjemploStruct ejemplos[20];
  int numEjemplos;
};
typedef vectorEjemplosEstatico * vectorEjemplos;
```

En segundo lugar, encontramos las estructuras definidas para **almacenar los Sensores**:

```
struct vectorSensoresEstatico{
  tipo_datoTS sensores[20];
  int numSensores;
};

typedef vectorSensoresEstatico * vectorSensores;
```

4.6.2. Operaciones

- **Operaciones sobre las estructuras para almacenar Ejemplos:**

```
/* DESC:   Inicializa el vector de ejemplos
 * PRE:    vEjemplos apuntando a basura
 * POST:   vEjemplos válido y numEjemplos=0
 * PARAM:  -
 * RET:    -
 * COMP:   O(1)
 */
void inicializarVectorEjemplos(vectorEjemplos & vEjemplos);
```

```
/* DESC:   Inserta una transicion a la lista de transiciones de un ejemplo
 * PRE:    vEjemplos inicializado
 * POST:   ejemplo con una transicion más
 * PARAM: ejemplo -> puntero al ejemplo sobre el que insertar
                transiccion-> transicion a insertar
 * RET:    -
 * COMP:   O(1)
 */
void insertarTransiccion(EjemploPuntero ejemplo, tipo_cadena transiccion);

/* DESC:   Inserta un ejemplo a la lista de ejemplos
 * PRE:    vEjemplos inicializado
 * POST:   vEjemplos con un ejemplo más
 * PARAM:  ejemplo-> puntero al ejemplo a insertar
 * RET:    -
 * COMP:   O(1)
 */
void insertarEjemplo(vectorEjemplos vEjemplos, EjemploStruct ejemplo);

/* DESC:   Muestra todos los ejemplos almacenados en el vector de ejemplos
 * PRE:    vEjemplos inicializado
 * POST:   -
 * PARAM:  fsal-> flujo de salida para mostrar los ejemplos
 * RET:    -
 * COMP:   O(n²)
 */
void mostrarEjemplos(vectorEjemplos vEjemplos,  ofstream & fsal);
```

- **Operaciones sobre las estructuras para almacenar Sensores:**

```
/* DESC:   Inicializa el vector de sensores
 * PRE:    vSensores apuntando a basura
 * POST:   vSensores válido y numSensores=0
 * PARAM:  -
 * RET:    -
 * COMP:   O(1)
 */
void inicializarSensoresEjemplo(vectorSensores & vSensores);

/*DESC: Inserta un sensor en la lista de sensores. Si ya existe un sensor con el mismo nombre,
lo sobreescribe
* PRE:    vSensores inicializado
* POST:   vSensores con un sensor más, o con un sensor modificado
* PARAM:  sensor -> sensor a insertar
* RET:    -
* COMP:   O(n)
*/
void insertarSensor(vectorSensores vSensores, tipo_datoTS sensor);
```

```
/*DESC: Elimina todos los sensores del vector de sensores
* PRE:    vSensores inicilizado
* POST:   numSensores=0;
* PARAM:  -
* RET:    -
* COMP:   O(1)
*/
void resetSensores (vectorSensores vSensores);

/*DESC: Devuelve true si ese sensor está en la lista. Si no lo esta devuelve falso,
                    y el campo sensor no tiene un valor válido
* PRE:    vSensores inicilizado
* POST:   -
* PARAM:  S:sensor -> sensor a devolver
                nomSensor -> nombre del sensor a consultar
* RET:    True si se ha encontrado un sensor con nombre 'nomSensor'. False si no
* COMP:   O(n)
*/
bool obtenerSensor(vectorSensores vSensores, tipo_cadena nomSensor, tipo_datoTS & sensor);
```

5. Ejemplo de fichero de entrada

```
//Ejemplo de un programa básico en DOMOL
//No tiene errores
VARIABLES        //Una variable entera, dos de tipo real y una de tipo lógico
        temp_basica = 20
        temp_verano = temp_basica * 1.1
        temp_invierno = temp_basica / 1.1
        verano = falso

SENSORES
        float T1,T2      //miden la temperatura, devuelve float
        int Reloj        //mide la hora en el reloj del sistema, devuelve int
        bool S           //mide si el sistema está o no operativo, devuelve bool

ACTUADORES
        Calefaccion      //calefacción
        A_C              //aire acondicionado
        SYSTEM           //sistema domótico


ESTADOS
        Inicial
        Final
        Final_Intermedio
        Todo_OFF
        C_ON
        AC_ON


TRANSICIONES
        iniciar:         Inicial  -> Todo_OFF
        encender_calef:          Todo_OFF -> C_ON
        apagar_calef:            C_ON     -> Final_Intermedio
        encender_airea:          Todo_OFF -> AC_ON
        apagar_airea:            AC_ON    -> Todo_OFF
        finalizar:               Final_Intermedio -> Final
        acabar1:         Todo_OFF -> Final
        acabar2:         C_ON     -> Final
        acabar3:         AC_ON    -> Final


COMPORTAMIENTO
        Inicial [

        si(falso==S)[
                        activar SYSTEM
                        desactivar Calefaccion
                        ]
        ]

        Final [
                desactivar Calefaccion
                desactivar A_C
                desactivar SYSTEM
        ]

        Todo_OFF [

                        activar Calefaccion
                        transicion encender_calef
                        activar A_C
                        transicion encender_airea

                        transicion acabar1

        ]

        C_ON [
                si T2 > temp_verano  [
                        desactivar Calefaccion
                        transicion apagar_calef
                ]

                        transicion acabar2

        ]

        AC_ON [
                si T1 < temp_invierno [
```

```
                        desactivar A_C
                        transicion apagar_airea
                ]

                        transicion acabar3
        ]

        Final_Intermedio[
                desactivar A_C
                desactivar Calefaccion
                transicion finalizar
        ]

CASOS DE USO
        CASO Invierno: Inicial; Todo_OFF; C_ON; Todo_OFF; C_ON; Todo_OFF; Final
        CASO Primavera: Inicial; Todo_OFF; Final

EJEMPLOS
  EJEMPLO A
        S verdadero             //transición iniciar
        T1   10.0               //transición encender_calefacción
        S falso                 //transición acabar2

EJEMPLO B
        S verdadero             //transición iniciar
        S falso                 //transición acabar1

EJEMPLO C
        S verdadero             //transición iniciar
        T1   15.5               //transición encender_calefacción
        T2   24.7               //transición apagar_calefacción
        S falso                 //transición acabar1     desactivar A_C
```

6. Ejemplo de fichero de salida

```
TABLA DE SIMBOLOS
=================
NOMBRE                          TIPO                    VALOR
================================================================
temp_basica                     entero                  20
temp_verano                     real                    22
temp_invierno                   real                    18.1818
verano                  logico                  falso
T2                      sensor real                     ???
T1                      sensor real                     ???
Reloj                   sensor entero                   ???
S                       sensor logico                   ???
Calefaccion                     actuador                0
A_C                     actuador                0
SYSTEM                  actuador                0
Inicial                 estado                  0
Final                   estado                  1
Final_Intermedio                estado                  2
Todo_OFF                estado                  3
C_ON                    estado                  4
AC_ON                   estado                  5
iniciar                 transicion                      0
encender_calef                  transicion                      1
apagar_calef                    transicion                      2
encender_airea                  transicion                      3
apagar_airea                    transicion                      4
finalizar                       transicion                      5
acabar1                 transicion                      6
acabar2                 transicion                      7
acabar3                 transicion                      8


TABLA DE TRANSICIONES
=====================
 ******        Inicial        Final        Final_Intermedio        Todo_OFF        C_ON        AC_ON
================================================================================

Inicial   ||    ------     ------     ------     iniciar      ------     ------

Final   ||    ------     ------     ------     ------     ------     ------

Final_Intermedio   ||    ------     finalizar      ------     ------     ------     ------

Todo_OFF   ||    ------     acabar1      ------     ------     encender_calef      encender_airea

C_ON   ||    ------     acabar2      apagar_calef      ------     ------     ------

AC_ON   ||    ------     acabar3      ------     apagar_airea      ------     ------

Comprobacion de casos de uso
============================
CASO Invierno
    iniciar
    encender_calef
    apagar_calef
    encender_calef
    apagar_calef
    acabar1
CASO Primavera
    iniciar
    acabar1


TABLA DE CORRESPONDENCIA ESTADO-INSTRUCCIONES
=============================================
ESTADO           PRIMERA INSTRUCCION       NUMERO INSTRUCCIONES
================================================================
Inicial          0                           5
Final          5                           3
Final_Intermedio          21                          3
Todo_OFF          8                           5
C_ON          13                          4
AC_ON          17                          4


TABLA DE COMANDOS
=================================================
```

```
NUM.     INSTRUCCION                              COMANDO                                  PARAM1
PARAM2
=============================================================================================================
=====
0        SI_SIMPLE (4)          (falso==S)                                                    4-0
1        ACTIVAR (1)          SYSTEM                                                       NULO
2        DESACTIVAR (2)         Calefaccion                                                  NULO
3        DESACTIVAR (2)         A_C                                                        NULO
4        TRANSICION (3)         iniciar                                                      NULO
5        DESACTIVAR (2)         Calefaccion                                                  NULO
6        DESACTIVAR (2)         A_C                                                        NULO
7        DESACTIVAR (2)         SYSTEM                                                     NULO
8        ACTIVAR (1)          Calefaccion                                                  NULO
9        TRANSICION (3)         encender_calef                                               NULO
10        ACTIVAR (1)          A_C                                                      NULO
11        TRANSICION (3)          encender_airea                                            NULO
12        TRANSICION (3)          acabar1                                                  NULO
13        SI_SIMPLE (4)         T2>temp_verano                                               2-0
14        DESACTIVAR (2)          Calefaccion                                              NULO
15        TRANSICION (3)          apagar_calef                                              NULO
16        TRANSICION (3)          acabar2                                                  NULO
17        SI_SIMPLE (4)         T1<temp_invierno                                             2-0
18        DESACTIVAR (2)          A_C                                                    NULO
19        TRANSICION (3)          apagar_airea                                             NULO
20        TRANSICION (3)          acabar3                                                  NULO
21        DESACTIVAR (2)          A_C                                                    NULO
22        DESACTIVAR (2)          Calefaccion                                              NULO
23        TRANSICION (3)          finalizar                                               NULO


SIMULACION DE LOS EJEMPLOS
     EJEMPLO A
         iniciar
         encender_calef
     EJEMPLO B
         iniciar
         encender_calef
     EJEMPLO C
         iniciar
         encender_calef
         apagar_calef
         finalizar
```

7. Conceptos relacionados con la asignatura DMSS

Concepto de TL	Concepto de DMSS	Comentarios
Gramática	Metamodelo	Al definir la gramática, estamos definiendo un metamodelo.
Entrada a la YYIN	Modelo .xmi	Lo que nosotros pasamos a los analizadores, son los modelos que definimos en DMSS.
Funciones atribuidas	Generación de código Acceleo	Al hacer operaciones y mostrar resultados, obtenemos algo similar a lo que hacemos con Acceleo.

8. Conclusiones y principales problemas

Como conclusión, me gustaría añadir que, el hecho de haber hecho este proyecto, me ha ayudado a asentar los conocimientos estudiados en teoría, sobre todo para el segundo parcial.

Igualmente, he considerado interesante el punto de comparar lo realizado en este proyecto, con lo realizado en el proyecto de la asignatura de DMSS.

Otra de las cosas que me han parecido útiles de este proyecto, es que me ha permitido volver a refrescar mis conocimientos acerca de C/C++, incluso ampliarlos en algunos aspectos (sobre todo en tratamiento de cadenas).

Resulta interesante ver como puedes crear tu propio “compilador”, para un lenguaje propio, inventado por ti, apoyandote en las herramientas que Bison y Flex proporcionan.

En cuanto a los problemas que he experimentado durante el desarrollo de esta práctica, muchos han sido debidos a las producciones recursivas, teniendo que controlar bien donde hacer la recursividad, así como donde añadir un fin de linea (pues, un símbolo tan simple, provocaba en muchas ocasiones errores difíciles de identificar). También, he tenido problemas, en la ultima parte (tratar los ejemplos), por el hecho de pasar estructuras estáticas por parámetro (pues no se modificaban los valores al salir del método). Este error fue fácil de solucionar, cambiando estas estructuras por punteros a las estructuras, como había hecho en el resto del proyecto.

En resumen, el proyecto ha sido enriquecedor, pues he aprendido, no solo conocimientos de esta asignatura, sino que he asentado y ampliado conocimientos previos.