Proyecto de Teoría de Lenguajes

Curso 21/22

Traductor lenguaje LISTAS

Autores:
Francisco Javier Mesa Martín
David Salguero Carrasco

Índice

En la presente documentación se abordan los conceptos de mayor relevancia respecto al proyecto desarrollado, tratando diferentes temas como las estructuras de datos empleadas, los analizadores confeccionados o los errores controlados. Se ofrece de forma adicional y junto al código una documentación interna que explica de forma más detallada diferentes aspectos de la implementación.

Las decisiones de diseño tomadas se indican a lo largo de este documento precedidas de tres asteriscos rojos (***).

1. Estructuras de datos

En esta primera sección se presentan las características principales de las estructuras de datos implementadas para la confección del proyecto. Concretamente, se ha optado por hacer uso de dos estructuras, una destinada a almacenar la información relativa a los identificadores (variables) definidos en el programa (a modo de tabla de símbolos), y otra para la definición y gestión exclusivamente de listas.

1.1. Identificadores

El principal objetivo de esta primera estructura, implementada como una librería a través de los ficheros **estructura.h** y **estructura.cpp**, es almacenar los identificadores definidos a lo largo del programa para, posteriormente, acceder a su valor y poder operar con ellos. Algunas operaciones pueden implicar una actualización de dicho valor.

Partiendo de lo anterior, se ha primado el uso de una estructura que permitiese búsquedas y accesos a elementos de forma eficiente. Por tanto, se ha optado por el uso de un **mapa** (librería ***map***), es decir, un árbol binario de búsqueda en el que cada uno de sus nodos se trata de una pareja **clave-valor**. Haciendo uso de dicha librería, la declaración del mapa supone su inicialización de forma dinámica, siendo esto lo óptimo para no limitar el número de identificadores que se pueden emplear en el programa[1].

Así, las **complejidades** de las operaciones básicas son las siguientes:

- La inserción de un único elemento supone una complejidad de O(log n).
- La búsqueda de un único elemento supone una complejidad de O(log n). En el peor de los casos se tendría un árbol degenerado, siendo esta complejidad de O(n).

[1] ***En el enunciado del proyecto se limita la definición de variables a 100 elementos. Para eliminar esta restricción, se ha optado por recurrir a una estructura dinámica con la que no atender dicha limitación y que el usuario defina tantas variables como desee.

- Mostrar todos los elementos del mapa supondría realizar un recorrido del árbol completo, presentando una complejidad de O(n).

Como ya se ha indicado, el mapa está constituído por parejas **clave-valor**. En este caso, la **clave** se corresponde con el nombre del identificador. Se solventa así el problema de contar con variables duplicadas, puesto que en la estructura empleada no se permite la inserción de elementos repetidos (con mismo valor de clave). Esta última situación también es controlada en las acciones asociadas a la gramática desarrollada. Por otro lado, el **valor** se trata de un objeto **Identificador** definido como un ***struct*** para albergar los siguientes campos:

- ***int*** **tipo**: entero que almacena el tipo de identificador según la siguiente correspondencia:
    - 0 - entero
    - 1 - real
    - 2 - lógico
    - 3 - lista
- ***tipo_valor*** **valor**: almacena el valor "real" del identificador. Para ello se hace uso de una *union* (*tipo_valor*) que almacena los siguientes campos:
    - ***int*** **valor_entero**
    - ***float*** **valor_real**
    - ***bool*** **valor_booleano**
    - ***char*** **valor_identificador[25]**: array de 25 caracteres como máximo para almacenar el nombre de un identificador procedente de una lista.
- ***int*** **n_linea**: último número de línea en el que se actualizó el identificador.

Los dos siguientes campos, pertenecientes también al objeto Identificador, únicamente son empleados cuando el identificador a almacenar es del tipo Lista, es decir, cuando el campo *tipo* es igual a 3. Esto se ha decidido así para no tener que realizar búsquedas adicionales sobre la estructura que almacena las listas.

- ***string*** **nombreLista**: cadena que almacena el nombre de la lista sobre la que se ha definido el identificador.
- ***int*** **tipo_lista**: entero que almacena el tipo de valores que contiene la lista (es decir, el tipo de lista) según la siguiente correspondencia. Este valor permitirá determinar qué campo de la unión *tipo_valor* debe emplearse para almacenar el valor "real" del identificador si este último ha sido definido sobre una lista.
    - 0 - entero
    - 1 - real
    - 2 - lógico
    - 3 - identificador

Por último, las **operaciones** definidas sobre la estructura son las siguientes:

- **insertarIdentificador**: inserta un nuevo identificador en la estructura. En caso de que ya exista, se actualiza su valor siempre y cuando los tipos coincidan.
- **buscarIdentificador**: determina si un identificador existe o no en la estructura.
- **mostrarIdentificadores**: muestra por pantalla todo el contenido de la estructura, ofreciendo la información básica de cada uno de los identificadores.

1.2. Listas

Esta segunda estructura se ha destinado a almacenar únicamente las listas que son definidas en el bloque de LISTAS. Para ello, se ha confeccionado una librería a través de los ficheros **estructuraListas.h** y **estructuraListas.cpp**.

De nuevo, se pone en valor una estructura que permita búsquedas y accesos a elementos de forma eficiente, ya que el principal cometido de esta nueva estructura es realizar consultas sobre las listas ya definidas y los valores que estas contienen. Por ello, según lo que se ha explicado ya en el apartado anterior, se ha decidido volver a recurrir a un mapa (librería ***map***) que se inicializa de forma dinámica. ***Se permite así definir tantas listas como se desee, eliminado el límite establecido a 20.

En este caso, la **clave** se corresponde con el nombre de la lista. Al igual que sucedía con los identificadores, se solventa así el problema de definir una lista con mismo nombre más de una vez (aunque se trata de una situación que sigue controlándose también a través de las acciones asociadas a la gramática). Por otro lado, el **valor** se trata de un objeto **Lista** definido como un ***struct*** para albergar los siguientes campos:

- ***int* tipo**: entero que indica el tipo de elementos que almacena la lista según las siguientes correspondencias:
  - 0 - enteros
  - 1 - reales
  - 2 - lógicos
  - 3 - identificadores
- ***int* n_linea**: número de línea en el que se ha definido la lista. Este campo se ha añadido como información adicional, aunque en la salida final no será mostrado por pantalla junto al resto de información.
- ***vector<int>* lista_enteros**: vector de enteros inicializado de forma dinámica empleado para almacenar los elementos de la lista si esta es definida como una lista de enteros.

- ***vector<float>* lista_reales**: vector de valores reales inicializado de forma dinámica empleado para almacenar los elementos de la lista si esta es definida como una lista de reales.
- ***vector<bool>* lista_logicos**: vector de valores booleanos inicializado de forma dinámica empleado para almacenar los elementos de la lista si esta es definida como una lista de valores lógicos.
- ***vector<string>* lista_identificadores**: vector de cadenas inicializado de forma dinámica empleado para almacenar los elementos de la lista si esta es definida como una lista de identificadores.

Como se puede observar, se ha recurrido a la librería ***vector***, ya que permite trabajar de forma dinámica y ofrece ciertas facilidades en el acceso a elementos. ***Se consigue así poder definir listas de tantos elementos como se quiera, eliminando la restricción impuesta de trabajar con listas con longitud menor a 50 elementos. Esto se ha considerado mejor opción que, por ejemplo, definir arrays convencionales con un tamaño máximo predefinido muy alto. Como único inconveniente, el uso de vectores ha supuesto tener que prescindir del uso de una *union* tal y como se ha hecho con los identificadores. Esto se debe a que dentro de una *union,* por su naturaleza, no se pueden incluir elementos dinámicos.

Por último, las **operaciones** que han sido definidas junto a la estructura son las siguientes:

- **insertarLista**: inserta una nueva lista en la estructura. Si ya existe una lista con mismo nombre, no se hace nada y no se inserta, puesto que a diferencia de los identificadores las listas no se pueden actualizar.
- **buscarLista**: determina si una lista existe o no en la estructura.
- **mostrarListas**: muestra por pantalla todo el contenido de la estructura, ofreciendo la información básica de cada una de las listas definidas.
- **primeroLista**: devuelve como cadena el primer elemento de la lista.
- **ultimoLista**: devuelve como cadena el último elemento de la lista.
- **enesimoLista**: devuelve como cadena el n-esimo elemento de la lista. Dentro de esta función se controla que el índice del elemento a recuperar sea el correcto.
- **limpiarLista**: elimina todos los elementos de la lista.

2. Analizador léxico

Respecto al análisis léxico se ha incluido el reconocimiento de tokens necesario teniendo en cuenta las Actividades 05 y 06, y los nuevos requisitos expuestos en la propia descripción del proyecto. Sin embargo, cabe recalcar cómo se han considerado en diversas situaciones los saltos de línea que se deben dar en el programa de entrada:

```
^[ \t]*LISTAS                   return LISTAS;
^[ \t]*VARIABLES                return VARIABLES;
^[ \t]*INICIO                   return INICIO;
^[ \t]*FIN                      return FIN;
```

Como se puede observar en los anteriores casos, se reconocen los tokens LISTAS, VARIABLES, INICIO y FIN (asociados a los bloques que deben aparecer en el programa) controlando directamente que sean los primeros tokens reconocidos en sus respectivas líneas. Esto se ha hecho así para controlar que, por ejemplo, el token INICIO no pueda aparecer en la misma línea que la última definición de variable.

Esto se ha hecho de esta forma ya que controlar situaciones como las expuestas con saltos obligatorios y saltos opcionales supone la aparición de conflictos en la gramática. Partiendo del ejemplo indicado en el párrafo anterior, en la gramática la secuencia de variables está compuesta por variables seguidas de un salto opcional ya que se pueden declarar tantas variables como se quiera en la misma línea. De esta forma, para que el token INICIO apareciese en una nueva línea sería necesario incluir un salto obligatorio tras el bloque de definición de dichas variables. Esto último genera un conflicto que únicamente se ha podido solventar modificando el analizador léxico tal y como se ha indicado arriba.

La misma situación se ha dado con la parte Sino de los condicionales:

```
[ \t\n]*Si_no                   return SI_NO;
```

3. Analizador sintáctico

A continuación, se exponen los aspectos clave que se deben contemplar de la gramática y sus respectivas acciones para una mejor comprensión del analizador sintáctico desarrollado.

Antes de continuar, indicar únicamente que se ha optado por una gramática ligeramente más compleja en beneficio de no contar con ningún tipo de conflicto gramatical, de modo que el comportamiento de la misma sea el esperado.

3.1. Variables y objetos auxiliares

Para controlar diversas situaciones con la gramática y sus acciones, se han definido una serie de variables. Cada una de ellas tiene su propio propósito, comentado a continuación:

- ***bool* errorSemantico**: bandera empleada para el control y la propagación de errores semánticos.
- ***bool* errorAux**: bandera empleada para controlar si en una expresión aritmética se han empleado identificadores lógicos y/o del tipo lista. Esto se debe a que con *errorSemantico*, entre otras cosas, se controla que no puedan aparecer identificadores de dichos tipos en expresiones aritméticas en vista a asignaciones u operaciones como tal. Sin embargo, con la inclusión de las listas sí que pueden aparecer en ciertas expresiones, como por ejemplo en las referencias. Por ello, se ha decidido emplear esta nueva bandera para no generar ni mostrar errores semánticos en situaciones erróneas.
- ***int* tipo_variable**: entero que almacena, durante el bloque de VARIABLES, el tipo de variable que se está definiendo. '0' corresponde al tipo entero, '1' al real, '2' al lógico y '3' al tipo lista.
- ***bool* definicionLista**: bandera para determinar si la ejecución del programa se encuentra por el bloque de LISTAS. Esto se debe, principalmente, a que en la definición de listas se pueden emplear identificadores. Como estos forman parte de las expresiones aritméticas, se comprobaba que el identificador en cuestión existiese en la estructura pertinente. De este modo, la bandera *definicionLista* permite filtrar el flujo para evitar esta última comprobación y no generar el error semántico correspondiente.
- ***bool* esIdentificador**: bandera empleada para determinar si en una expresión aritmética o lógica aparece o no un identificador. Esto resulta de gran utilidad ya que se dan múltiples situaciones en las que únicamente se pueden emplear expresiones en las que no aparezcan identificadores. Por ejemplo, en la definición de una lista por rango.
- ***bool* noSoloId**: bandera que indica si en una expresión aparecen elementos numéricos (valores numéricos u operadores a excepción de los paréntesis). El uso de esta nueva bandera se fundamenta en que existen situaciones en las que pueden aparecer únicamente identificadores en las expresiones (ej: en la definición de una lista de identificadores), otras en las que pueden aparecer tanto identificadores como elementos numéricos (ej: asignaciones) y otras en las que solo pueden aparecer elementos numéricos (ej: definición de una lista por rango). La combinación de esta bandera con la anterior bandera *esIdentificador* permite controlar todas las situaciones que se acaban de exponer.
- ***string* nombreLista, valorAux**: cadenas auxiliares para manejar referencias a listas e inserciones de identificadores en la definición de listas.
- ***stack<bool>* banderasCondicional, aux**: pilas empleadas para la anidación de condicionales. La implementación se ha decidido llevar a cabo con dos pilas para no tener que desapilar y volver a apilar sobre la pila principal en ciertas situaciones.
- ***vector<string>* vAux**: vector para la gestión de definición de variables y la ejecución de los parámetros de la instrucción *Escribir()* cuando se producen

errores sintácticos. Esencialmente, se emplea este vector para evitar que, por ejemplo, ante la presencia de un error sintáctico en la definición de una variable esta se inserte en la tabla de símbolos.

- ***Identificador* id**: objeto *Identificador* auxiliar empleado para la interacción con la estructura de identificadores.
- ***Lista* lista**: objeto *Lista* auxiliar empleado para la interacción con la estructura destinada a las listas.

3.2. Aspectos de relevancia

En este subapartado se enumeran aspectos concretos que deben tenerse en consideración acerca de la implementación de la gramática y sus respectivas acciones. Siempre que sea posible, se incluye el número de línea del fichero *listas.y* en el que se encuentra el aspecto referenciado.

- ***Línea 39** - Las cadenas de caracteres que se pueden emplear en la instrucción *Escribir()* se han definido con una extensión máxima de 101 caracteres, incluyendo el fin de cadena y las dobles comillas. Es decir, el valor útil de la cadena tendrá una extensión, como máximo, de 98 caracteres.
- ***Línea 73** - Antes de encontrar el token LISTAS se ha permitido incluir tantos saltos de línea como se desee. Es decir, el inicio del bloque LISTAS no tiene por qué encontrarse en la primera línea del fichero de entrada. Para ello se ha recurrido al símbolo *saltoOpcional*.
- ***Línea 76** - El bloque de LISTAS puede aparecer vacío, es decir, no hace falta definir ninguna lista si así se desea.
- ***Línea 77** - En el enunciado del proyecto se indica textualmente que <<*La definición de una lista se hará en una línea con el siguiente formato:*>>. Esto último se ha interpretado como que únicamente se puede definir una sola lista por línea.
- **Línea 80** - No se ha indicado el delimitador del error sintáctico (que se trataría de un *saltoObligatorio*), ya que este se trata justo en la producción recursiva del símbolo inmediatamente superior (*secuenciaListas*). En caso de llevar dicho salto al error y al resto de producciones del símbolo lista, implicaría mostrar un número de línea erróneo al detectar el error sintáctico si en el fichero de entrada, después de la lista mal formada, se incluyen saltos de línea.
- **Línea 90** - Para la inserción de cada lista se tiene que hacer uso del objeto auxiliar *Lista lista*. Esto quiere decir que para cada lista insertada uno de los cuatro vectores con los que cuenta el objeto ha sido rellenado con los valores pertinentes. Por ello, para no encontrar más tarde valores basura, tras la inserción se debe limpiar el vector que haya sido empleado. Aquí entra en juego la función *limpiarLista()* ya comentada en secciones anteriores. Esta

misma forma de proceder se repite al capturar errores sintácticos, ya que el objeto *lista* podría quedar en un estado inesperado.

- **Línea 130** - Únicamente se han definido dos tipos de secuencias de valores que pueden encontrarse en la definición de listas: secuencia de expresiones booleanas (para listas con valores de tipo lógico) y secuencia de expresiones aritméticas. Estas últimas incluyen tanto los valores enteros, como los reales y los identificadores. Por tanto, el primer elemento insertado en la lista definirá su tipo (**Línea 136**). ***Adicionalmente, se ha decidido que una lista no puede definirse vacía.
- **Línea 248** - No se ha indicado el delimitador del error sintáctico (que se trataría de un *saltoOpcional*), ya que este se trata justo en la producción recursiva del símbolo inmediatamente superior (*secuenciaVariables*).
- ***Línea 261** - Se ha considerado que después del token FIN se debe tener uno o más saltos de línea. Para ello se ha hecho uso del símbolo *saltoObligatorio*.
- **Línea 269** - A diferencia del resto de instrucciones (con excepción del condicional), el ';' tras la asignación se ha derivado a las tres producciones del símbolo *asignacion*. A pesar de que así la gramática resulta ligeramente menos intuitiva, esto se ha hecho de esta forma para un control más exhaustivo de los errores sintácticos. De no ser así, asignaciones que no tuviesen, por ejemplo, ';' al final, serían ejecutadas y los cambios se reflejarían en memoria a pesar de lanzarse el error sintáctico.
- **Línea 272** - No se ha indicado el delimitador del error sintáctico (que se trataría, de nuevo, del *saltoOpcional*), ya que este se trata justo en la producción recursiva del símbolo inmediatamente superior (*secuenciaInstrucciones*). No se ha incluido dicho saltoOpcional en cada una de las instrucciones ya que suponía la aparición de conflictos gramaticales, y tal como se ha indicado al inicio de este apartado, se ha primado conseguir una gramática sin ningún tipo de conflicto.
- ***Línea 369** - Se ha definido una nueva secuencia de instrucciones, *secuenciaInstruccionesBloque*, en la que únicamente puede aparecer una sola instrucción en una misma línea. Esto se debe a que, tal y como se indica al inicio del apartado 3.5 del enunciado del proyecto, <<*Cada instrucción se escribirá en una línea distinta*>>. A pesar de que el ejemplo de condicional propuesto en el mismo enunciado se contradice con la anterior afirmación, se ha optado finalmente por mantener una sola instrucción por línea.
- ***Si se produce un error sintáctico en algunos de los parámetros pasados a la instrucción *Escribir()*, no se ejecutará ni traducirá ninguno de ellos hasta que el error no sea solventado.
- ***Si se produce un error sintáctico en cuanto a la estructura básica del programa (definición de bloques), el fichero de salida es generado pero sin la opción de compilar adecuadamente.
- ***Los elementos de la instrucción *Escribir()* se pasan como cadenas por simplicidad, puesto que su único cometido es mostrar por pantalla.

- ***Al definir una variable de tipo entero o real, se inserta por defecto en la estructura con el valor 0. Al definir una variable de tipo lógico, se inserta por defecto en la estructura con el valor ‘falso’.
- ***Si se define más de una lista en la misma línea, solo se almacenará la primera de ellas.

4. Control de errores

Otro aspecto que se ha querido priorizar en la confección de este proyecto ha sido la detección, el control y la recuperación de una amplia gama de errores, tanto sintácticos como semánticos.

Por un lado, los errores sintácticos pueden ser muy variados (todo lo que no cumpla con la gramática definida generará uno de estos errores). Todos ellos son recuperados a excepción de los errores reportados en la definición de la estructura básica del lenguaje, como por ejemplo, que no se incluya el token de uno de los bloques. En casos como este último la ejecución del programa finaliza.

Por otro lado, a continuación se ofrece una lista con todos los errores semánticos controlados, y para cada uno de ellos se proporciona el mensaje de error que es mostrado al usuario por pantalla. Cabe recalcar que todo error semántico, al ser mostrado, comienza con “> *Error semántico en la instrucción XX: ...*”. En la siguiente lista se incluirá únicamente el texto que sigue al anterior mensaje.

- Se define una lista ya definida anteriormente: *…la lista 'XXXX' ya está definida*
- En la definición de una lista por rango se emplea como valor inicial y/o final una expresión que no es entera: *…el rango debe definirse con expresiones enteras*
- En la definición de una lista por rango, el valor de inicio es mayor que el valor de fin: *…el valor de inicio del rango debe ser menor o igual que el valor final*
- Se incluye un identificador en algunas de las expresiones enteras empleadas en la definición de una lista por rango: *…no es posible emplear identificadores en las expresiones enteras que determinan un rango al definir listas*
- En la definición de una lista de enteros, reales o identificadores se emplea una expresión en la que interviene uno o más identificadores acompañados de algún elemento numérico: *…no es posible emplear identificadores junto a expresiones numéricas en la definición de listas*
- En la definición de una lista de tipo entero o real se incluye un identificador: *…no es posible insertar un identificador en una lista definida sobre valores no identificadores*

- En la definición de una lista de enteros o identificadores se incluye un valor real: *…no es posible insertar un valor real en una lista definida sobre valores no reales*
- En la definición de una lista de reales o identificadores se incluye un valor entero: *…no es posible insertar un valor entero en una lista definida sobre valores no enteros*
- En la definición de una lista de tipo lógico se incluye un identificador: *…no es posible emplear identificadores en expresiones lógicas al definir una lista*
- En la definición de una variable de tipo lista se indica el nombre de una lista que no ha sido definida de forma previa: *…la lista 'XXXX' no está definida*
- Se define una variable con el mismo nombre que otra variable ya definida anteriormente: *…la variable 'XXXX' ya está definida*
- En una expresión aritmética se hace uso de un identificador de tipo lógico o lista: *…variables de tipo lógico o lista no pueden aparecer en expresiones*
- En una referencia se indica el nombre de una lista que no está definida: *…la lista referenciada (XXXX) no está definida*
- En una referencia *enesimo()* se indica como segundo parámetro un valor que se encuentra fuera de rango: *…el índice introducido en la función se encuentra fuera de rango -> [1, tamaño lista]*
- En una referencia *enesimo()* se indica como segundo parámetro una expresión que no es del tipo entero: *…el índice introducido debe tratarse de una expresión aritmética de tipo entero*
- En una referencia *enesimo()* se introduce como segundo parámetro una expresión mal formada (es decir, que arrastra un error semántico anterior): *…el índice introducido en la función no es correcto*
- En una asignación en la que la parte derecha se trata de una expresión aritmética, el tipo de las partes implicadas no coincide: *…el tipo de 'XXXX' es XXXX y no se le puede asignar un valor de tipo XXXX*
- El identificador empleado en la parte izquierda de una asignación no existe: *…el identificador 'XXXX' no se ha definido y no se puede operar con él*
- En una asignación en la que la parte derecha se trata de una expresión lógica, el identificador que recibe la asignación no es booleano: *…el tipo de 'XXXX' es XXXX y no se le puede asignar un valor de tipo booleano*
- En una asignación en la que la parte derecha se trata de una referencia, el identificador que recibe la asignación no es del tipo lista: *…el identificador 'XXXX' no es del tipo lista*
- En una asignación en la que la parte derecha se trata de una referencia, el identificador que recibe la asignación es del tipo lista, pero esta última no coincide con la lista referenciada en la parte derecha: *…la lista sobre la que se ha definido el identificador 'XXXX' (XXXX) no coincide con la lista referenciada (XXXX)*
- En una expresión se emplea un identificador que no existe: *…la variable 'XXXX' no está definida y no se puede operar con ella*

- Se intenta realizar un división por cero en una expresión aritmética: *…división por 0*
- Se intenta realizar el módulo cero en una expresión aritmética: *…% con división por 0*
- Se usa el módulo con algún operando real en una expresión aritmética: *…se usa la operación % con operandos reales*

Junto a los ficheros entregados se encuentra un ejemplo de entrada (errores.list) en el que se han incluido todos los errores semánticos controlados y gran parte de los errores sintácticos que se pueden dar.

Respecto a los errores sintácticos, únicamente indicar que aquellos detectados en la estructura de los condicionales provocan que el fichero generado como salida no compile correctamente.

5. Ampliaciones

Como ampliación se ha incluido la anidación de condicionales. Dicha anidación no se ha limitado a un nivel concreto, siendo posible anidar tantos condicionales como se desee.

Tal y como ya se ha indicado a lo largo de esta documentación, lo anterior se ha conseguido haciendo uso de dos pilas de valores booleanos, *banderasCondicional* y *aux*:

- **banderasCondicional**: se trata de la pila “principal”. Antes de llamar en el main al procedimiento del analizador, en el fondo de esta pila se introduce un valor verdadero. Dicho valor permanecerá ahí hasta el final de la ejecución, ya que será la “bandera” a comprobar cuando se encuentren instrucciones fuera de condicionales, por lo que dichas instrucciones siempre deberían ser ejecutadas. Esto último supone apilar y desapilar por cada instrucción condicional encontrada, es decir, independientemente de la anidación, tras la ejecución de un condicional con o sin anidamiento, el estado de la pila debe quedar restaurado a un único valor verdadero en el fondo. En definitiva, esta pila irá almacenando las evaluaciones de las expresiones lógicas asociadas a cada condicional, con independencia del contexto (da igual el nivel de anidación y cómo hayan evaluado los condicionales anteriores). Como se verá a continuación, la ejecución de las instrucciones que se encuentren dentro de la parte Si o Sino no dependerá únicamente de dichas evaluaciones, sino también de que no se hayan dado errores semánticos u otras evaluaciones negativas en caso de existir niveles de anidación previos. Ahora sí se tiene en cuenta el contexto, el cual es controlado con la siguiente pila.

- **aux**: se trata de una pila secundaria en la que se introducirá un valor para cada valor apilado en la pila principal. El valor introducido en esta segunda pila será falso siempre y cuando la expresión lógica del condicional “actual” evalúe como verdadero y la traza del programa no se encuentre dentro de cualquier otro condicional (independientemente del nivel de anidación o de si se trata de la parte Si o Sino) que haya evaluado como falso. En caso contrario, toma valor verdadero. De esta forma, esta pila recoge para cada nivel de anidación si en alguno de los anteriores niveles se produjo un error o se evaluó negativamente una condición, invalidando así la ejecución de las instrucciones (tanto de la parte Si como de la Sino) de los nuevos condicionales anidados que se encuentren a pesar de cómo evalúen las expresiones lógicas de estos últimos.

El uso de estas dos pilas podría reducirse a una única pila en la que habría que ir apilando y desapilando para ir comprobando el valor introducido en evaluaciones anteriores al existir anidación. Sin embargo, esto supondría tener que trabajar no solo con el valor de la cima, sino también con valores hundidos, por lo que se ha optado por trabajar con dos pilas y siempre centrar la atención en el valor accesible.

Se proporciona un fichero de entrada, entradaAmpliacion.list, en el que se presentan un total de cinco ejemplos de condicionales anidados para verificar el correcto funcionamiento de la ampliación.