# PRÁCTICA DE
# PROCESADORES DEL LENGUAJE II
## Curso 2018 – 2019

APELLIDOS Y NOMBRE**: Rodríguez Sánchez, Manuel**

IDENTIFICADOR: **mrodrigue212**

CENTRO ASOCIADO MATRICULADO: **C.A. Motril**

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: **C.A. Motril**

EMAIL DE CONTACTO: **mrodrigue212@alumno.uned.es**

GRUPO (A o B): **B**

¿REALIZAS LA PARTE OPCIONAL? (SÍ o NO): **NO**

# Contenido

# 1. El analizador semántico y la comprobación de tipos

A lo largo de este apartado, se van a ir describiendo todas las operaciones y estructuras usadas en cada una de las producciones de la gramática, que han permitido realizar un control de la parte semántica; esto lleva incorporado la tabla de tipos, para la comprobación de éstos, y la tabla de símbolos para su gestión.

A partir de aquí, en cada uno de los ámbitos que se abren, se crea la correspondiente tabla de símbolos y su tabla de tipos. Para este y los sucesivos menesteres, se ha creado una clase llamada ***Soporte,*** ubicada dentro del paquete ***soporte*** y a su vez dentro de ***compiler,*** donde se van a añadir todos los métodos necesarios para el análisis semántico, entre ellos la gestión de la tabla de tipos, la gestión de la tabla de símbolos y la gestión y organización de la memoria.

Existen dos producciones dentro del parser.cup, donde se crea el ámbito:

- **Inicio del programa (producción program)**

```
program ::=
  {:syntaxErrorManager.syntaxInfo ("Starting parsing...");
       Soporte.creaAmbito("AmbitoGlobal");:}
```

- **Inicio de un procedimiento o función (producción cabProcedure)**

```
cabProcedure::= PROCEDURE IDENTIFICADOR:id procParenParam:pPP tipoRetorno:tR
PUNTOYCOMA
{:Soporte.gestionSimboloProcFun(id.getLexema(),tR);
      Soporte.creaAmbito(id.getLexema());
      Soporte.gestionParametros(pPP);
      RESULT=new CabProcedure(id.getLexema());:}
```

Al crear los ámbitos, a través de un método de la clase ***Soporte***, las tablas de tipos y de símbolos se crearán a la misma vez; ahora solo queda rellenarlas.

## 1.1. Descripción del manejo de la tabla de símbolos y de la tabla de tipos.

Como hemos indicado anteriormente, conforme se crea el ámbito se crean las tablas de tipos y símbolos de ese ámbito; esto hace que podamos gestionar primeramente la tabla de tipos, creándolos y almacenándolos. Para ello utilizamos el método ***gestionTablaTipos(ScopeIF ambito)*** de la clase ***Soporte***. Al llamar a este método, se crean los siguientes tipos:

- Dos ***TypeSimple***, uno para los números enteros, y otro para los valores booleanos.
- Un ***TypeVoid***, que será solo para los procedimientos, que no devuelven nada.
- Un ***TypeArray*** para las estructuras de tipo array. Este se podrá configurar para contener valores enteros o valores booleanos.
- Un ***TypeListParam***, es un tipo especial que no existía en la arquitectura, y que se ha creado con el objetivo de almacenar en una lista el número de parámetros y su tipo, para controlar cuando se hacen llamadas a procedimientos y funciones.

Se puede observar en la ilustración 1, el diagrama de secuencia del caso de uso de creación de tipos.

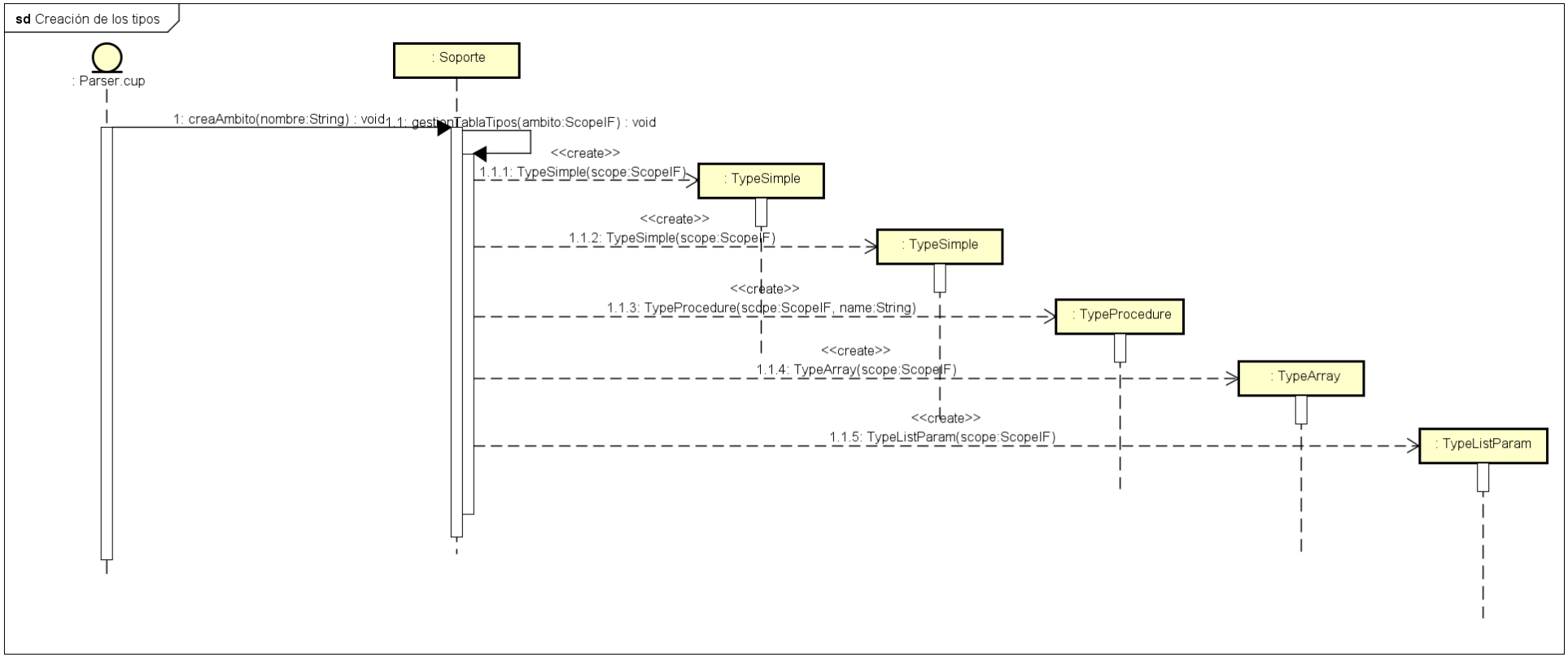



*Ilustración 1 - Proceso de creación de los tipos en forma de objetos, antes de introducirlos en la tabla de tipos.*

**Manejo y gestión de producciones del parser.cup**

A continuación, vamos a describir algunas de las producciones más significativas e influyentes, donde podremos ver como hacen uso y gestión de las tablas de símbolos y tipos. Es muy importante destacar, que gran parte del código java para la gestión de la parte semántica, se desarrolla en forma de métodos alojados en la clase ***Soporte***.

Producción axiom - Comprobamos el nombre de inicio y fin del módulo principal. Para ello se usa un método de la clase ***Soporte*** creado para pasarle el atributo nombre del programa que viene propagado en el objeto ***CabModule (cM)*** y en el objeto ***Cuerpo(c).*** Si no coinciden los nombres, se detiene la compilación.

```
axiom ::= cabModule:cM cuerpo:c
{:Soporte.comprobarNombresProgramas(cM.getNombrePrograma(),c.getNombrePrograma());:}
```

*Reglas de producción para cabecera de programa principal y cuerpo de los módulos*

Producción cabModule – Llamamos al método ***gestionNombrePrograma(identificador)*** para comprobar si el identificador (que será el nombre del programa en este caso) que se le pasa como parámetro no existe, y si es así, se crea el nuevo símbolo *identificador* junto a su ámbito y el tipo, que en este caso es de tipo void.

Aquí tenemos un claro ejemplo en el uso de la tabla de símbolos, donde primero consultamos si existe el símbolo identificador o en este caso el nombre del programa, y en segundo lugar, si no existe, lo añadimos a la tabla de símbolos del ámbito global.

Producción cuerpo – En esta producción se va a comprobar que el símbolo sea una función, y si lo es, que haya un *return*, y que el tipo del *return* sea igual al tipo de la función. Para estas comprobaciones se llama al método ***comprobarReturn(listaSententencias, nombreFuncion)*** de la clase ***Soporte***.

Podemos observar que no se ha comprobado si existe el símbolo identificador en la tabla de símbolos; esto no es necesario, ya que si se da el caso de que no coincide el identificador del inicio con el identificador del fin del cuerpo de sentencias de una función, procedimiento o programa principal, se infiere que uno de esos dos identificadores no existe, por tanto, si se da este caso se parará la compilación.

*Reglas de producción para constantes, tipos y variables*

Producción expConst y valorConst – Aquí vamos a comprobar que el identificador no exista, y si es así, crearlo y guardarlo en la tabla de símbolos, añadiéndole el valor que va a tener este símbolo constante. Para ello creamos en el objeto *SymbolConstant* un atributo valor, que recogerá el valor que ha de tener el símbolo constante. Para agregar el símbolo constante, se llama al método ***gestionSimboloConstante(identificador, valor)*** de la clase ***Soporte***; se le pasa como parámetros el identificador y el valor recogido de la producción ***valorConst***. En el método se realizan las siguientes acciones:

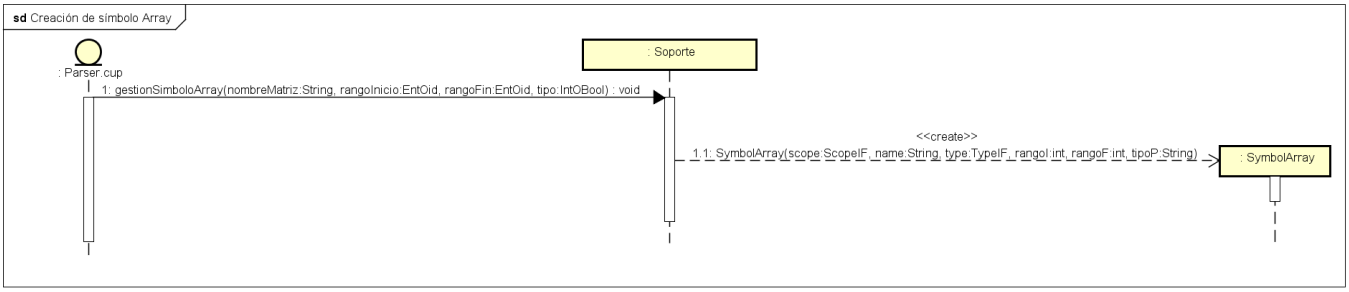
- Que el identificador o símbolo no exista.
- Si no existe, se comprueba si el valor que trae es booleano o entero.
- Se agrega el símbolo a la tabla de símbolos junto con su valor.

```
ExpConst ::= IDENTIFICADOR:simbolo IGUAL valorConst:vC PUNTOYCOMA
{:Soporte.gestionSimboloConstante(simbolo.getLexema(),vC.gettValor());:}

valorConst ::= ENTERO:numero
```

```
{:ValorConst vC=new ValorConst(numero.getLexema());
  RESULT = vC;:}
| vBooleano:vB {:ValorConst vC=new ValorConst(vB.getvBoolean());
  RESULT = vC;:};
```

Producción expTipo – En esta producción se gestionan los símbolos de tipo array o matrices; para ello utilizamos el método ***gestiónSimboloArray(matriz,rangoInicio,rangoFin,tipo)*** de la clase **Soporte**. Las producciones implicadas en esta derivación son ***entOid***, que nos traerá propagado un identificador variable o un número entero; ***intOBool*** que nos traerá propagado el tipo primitivo de datos que albergará la matriz: enteros o booleanos. Dentro del método ***gestionSimboloArray***() primeramente extraeremos el valor o la variable de los rangos, que de cada objeto ***EntOid*** trae al crearlo en su correspondiente producción. Estos valores los almacenaremos en el objeto ***SymbolArray***, el cual se ha modificado de la arquitectura original, para almacenar estos valores o variables en los atributos creados para este fin (***rangoInicio*** y ***rangoFin***); también se han creado dos atributos más, que contendrán el tipo primitivo del array (***Integer*** o ***Boolean***), y el tipo definido en la declaración de variables, por que podemos crear una variable de un tipo matriz declarada anteriormente. En segundo lugar, si el identificador no está repetido, procedemos a crear el objeto ***SymbolArray***, pasándole al constructor de la clase: ***ámbito***, ***nombreMatriz***, ***tipoSimboloArray***, ***rangoInicio***, ***rangoFin***, ***tipoPrimitivo***. Lo agregamos a la tabla de símbolos del ámbito en el que estamos. Podemos observar que ***tipoSimboloArray***, será el tipo no primitivo del array que estamos creando. (Creamos un array de un array).




Producción expVar – Aquí nos encontramos que podemos crear tres tipos de variables: de tipo entero, de tipo ***boolean*** y de tipo ***array***. Se llama al método ***gestiónSimboloVariable***(), pasándole dos parámetros, una lista de variables que nos vendrá propagada desde la producción ***cadIdVar***, y el tipo de esas variables desde la producción ***tipoVar***. Una vez dentro del método comprobaremos de cada variable de que tipo es y crearemos su objeto ***SymbolVariable***() con el ámbito, nombre y tipo, solo para los tipos ***Boolean*** e ***Integer*** Para las variables de tipo ***array***, se creará un objeto ***SymbolArray***(). Se añaden todas ellas a la correspondiente tabla de símbolos.

*Reglas de producción para procedimientos y funciones*Producción stmSubprogram – Mediante el método ***comprobarNombresProgramas***(), se comprueba que el nombre de inicio y fin del subprograma es el mismo.

Produccion cabProcedure – En esta producción:

1- Creamos el símbolo, comprobando que sea un procedimiento o una función, esto nos lo dirá lo que venga propagado desde la producción ***tipoRetorno*** que, si el objeto viene vacío (null) se creará un ***SymbolProcedure*** con tipo ***void***, y si no viene vacío, se creará un ***SymbolFunction*** con el tipo que venga desde la producción ***tipoRetorno*** (Integer o Boolean). Esto lo haremos a través del método ***Soporte.gestionSimboloProcFun(nombre,tipoDeRetorno).***

2- Creamos un nuevo ámbito, llamando al método ***Soporte.creaAmbito(nombre).***
3- Hemos de gestionar los parámetros, para ello se procede de la siguiente forma:
   a. En la producción ***procParam*** se crea un objeto **ProcParam** con una lista de variables y su tipo. Se propaga a la siguiente producción ***procListParam***.
   b. En la producción ***procListParam***, se crea un objeto **ProcListParam**, que contendrá una lista de objetos **ProcParam**. Se propaga a la producción ***procParenParam***.
   c. En la producción ***procParenParam***, se crea un objeto **ProcParenParam** con todos los parámetros encapsulados, y se sube a la producción ***cabProcedure***.

Cuando llegan los parámetros a la producción ***cabProcedure***, se llama al método ***gestionParametros(),*** pasándole como argumento la lista encapsulada ***ProcParenParam***, y extraemos cada uno de los parámetros, comprobamos su unicidad y los metemos en la tabla de símbolos de ese nuevo ámbito.

Hemos de controlar en las llamadas a ***subprogamas***, el número de parámetros y el tipo de cada uno de ellos. Para esto se ha creado un tipo llamado *Lista*; se desarrolla para ello una clase llamada ***TypeListParam***, que contendrá una estructura de tipo ***ArrayList*** con cada tipo de cada parámetro del subprograma. Este tipo se crea y se guarda en la tabla de tipos del ámbito del subprograma, usando el método ***guardarTipoEnLista(tipo).***

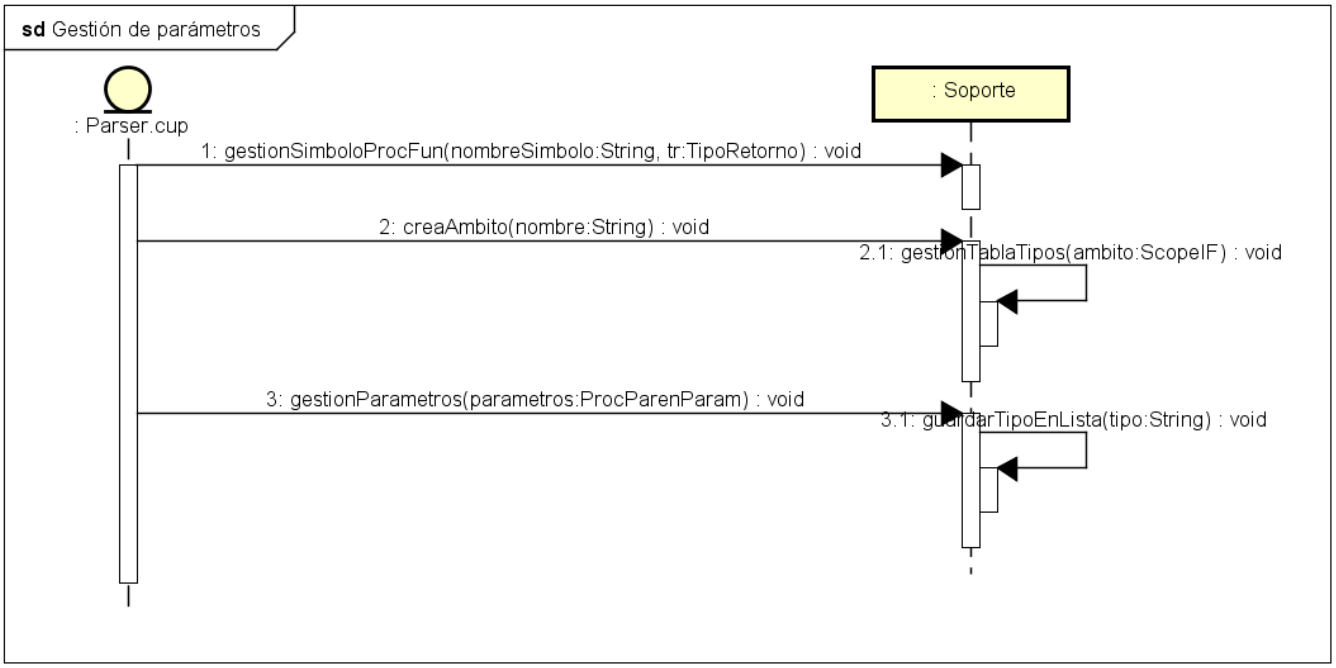



*Ilustración 2 – Control de subprogramas y sus parámetros.*

Ejemplo: Supongamos que tenemos la siguiente declaración de un subprograma.

*PROCEDURE resta(a,b:INTEGER;c:BOOLEAN):INTEGER*

Podemos observar que tenemos tres parámetros, dos de ellos de tipo entero y uno booleano. Al compilar el ***parser.cup*** se genera en la tabla de tipos del ámbito lo siguiente:

*Type - TypeListParam [scope = resta, name = Lista] - {listaTipo=[BOOLEAN, INTEGER, INTEGER]}*

Se ha creado un tipo ***Lista*** cuyo objeto ***TypeListParam*** contiene un array ***listaTipo*** con tres tipos de datos, que corresponden a los tres parámetros de la declaración. Cuando hagamos una llamada a este subprograma

desde el cuerpo de sentencias, no tendremos más que recuperar este tipo *Lista* del ámbito del subprograma, y compararlo con los parámetros de la llamada. Siguiendo con el ejemplo anterior, si hacemos la llamada:

i := resta(3,2,TRUE)

En la producción ***variables::=IDENTIFICADOR parFuncion*** se hará la llamada al método ***Soporte.compararListasParametros(),*** para realizar las operaciones para recuperar la lista hecha en la producción ***cabProcedure***, y que se almacenó en la tabla de tipos del ámbito y compararla con la lista de parámetros de la llamada.

*Reglas de producción para expresiones lógicas y aritméticas*

En la producción ***expresion***, crearemos un objeto ***Expresion***, que contendrá el tipo de la expresión (Integer, boolean, matriz o void). Ésta la usaremos para las distintas comparaciones y comprobaciones de compatibilidad de tipos en las distintas producciones para expresiones lógicas, aritméticas, y sentencias como IF, FOR o sentencias de asignación.

A partir de la producción ***expresiones*** comenzamos a trabajar con las tablas de tipos para la comparación de éstos, a la vez que hemos de controlar la existencia de los símbolos usando las tablas correspondientes.

En las expresiones aritméticas y lógicas, generalmente tendremos dos expresiones, que deberán cumplir que sus tipos sean iguales. Para este fin usamos el método ***Soporte.compararExpresiones(exp1, exp2***), que abortará la compilación, si los tipos de las dos expresiones pasadas como parámetros no coinciden.

*Reglas de producción para sentencias*

Con el objetivo de comprobar si existe o no un ***return*** en el cuerpo de un subprograma, se ha creado una estructura que almacenará en un ***arrayList***, todas las sentencias que nos vayamos encontrando en el cuerpo del programa o subprograma. En cada producción de cada instrucción se crea su objeto correspondiente, y se propaga hacia la producción ***sentencia***, creando en este punto un objeto de tipo ***Sentencia*** que albergará la instrucción que viene propagada. Este objeto ***Sentencia*** se propagará hasta la producción ***listSentencia*** donde se ha creado el ***arrayList*** para almacenar cada una de las sentencias que vengan de la propagación. Una vez hecha la lista se propagará hacia arriba, donde en la producción ***cuerpo*** comprobaremos si existe o no un ***return*** y si el tipo que trae asociado, es compatible con el tipo de la función declarada. Esto lo haremos con el método ***Soporte.comprobarReturn(listaSententencias, nombreSubprograma).*** En este método también se comprobará si el subprograma es una función o un procedimiento que, si se da este último caso, no se comprobaría la existencia de ***return***.

*Reglas de producción para sentencias de llamada a función o procedimiento*

En esta producción, destacar que hemos de comprobar los parámetros, y aquí es donde entra en juego la lista guardada en la tabla de tipos. Disponemos de dos métodos de la clase ***Soporte***:

- c***omprobarListaParametros(símbolo, ámbito)***
- ***compararListaParametros(símbolo, parametrosFuncion).***

El primer método se usa para comprobar si un símbolo es una función o un procedimiento, y si tiene parámetros declarados.

El segundo método es para comparar las listas de los parámetros de la declaración, con la lista de los parámetros de la llamada en un procedimiento o función.

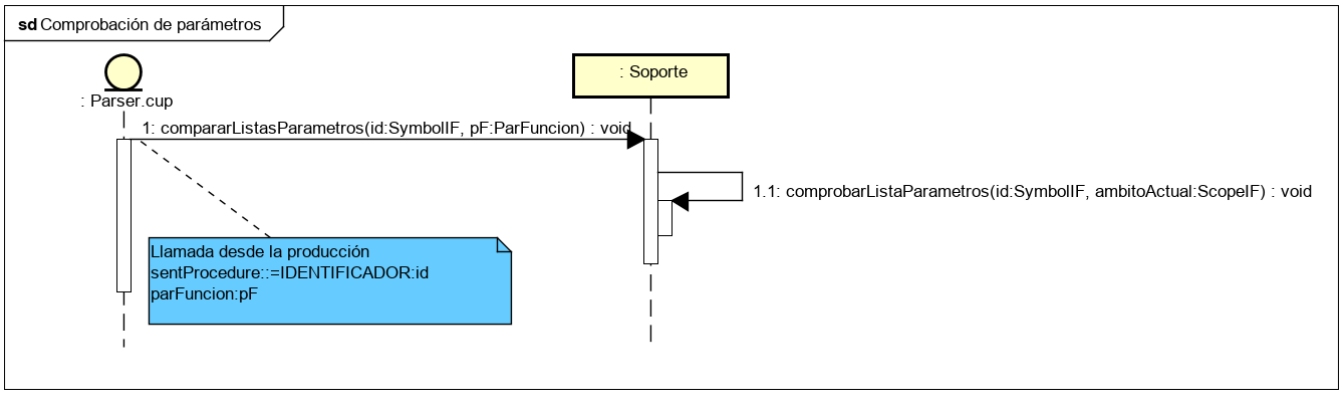



*Ilustración 3 - Comparación de parámetros en las llamadas.*

Desde la producción ***sentProcedure::=IDENTIFICADOR***, no es necesario llamar al método ***compararListasParametros***, se llama directamente al método ***comprobarListaParametros().***

*Reglas de producción para sentencias de salida*

La única producción que hemos de controlar el tipo es ***sWriteInt***, ya que la expresión asociada a ésta, ha de ser de tipo entero.

*Reglas de producción para tipos básicos*

Disponemos de la producción ***variables***, que deriva en:

IDENTIFICADOR: se controla que el símbolo exista y si es una función, se llamará al método ***comprobarListaParametros()***, ya que tenemos que comprobar si este símbolo ha de llevar parámetros, en el caso que el método encuentre en la declaración del símbolo que tiene un ***tipoListaParametros*** detendrá la compilación.

IDENTIFICADOR ***idArray***: se ha de controlar que el identificador exista y sea de tipo array. Para el caso del no terminal ***idArray***, tiene que ser de tipo entero, ya que es una expresión que indica una posición del array. Este ya ha sido comprobado desde su producción, no es necesario subirlo.

IDENTIFICADOR ***parFuncion***: tenemos que controlar que el identificador o símbolo exista, tiene que ser una función, y hemos de extraer el tipo de la función (entero o booleano). Aquí se llama primero a ***compararListasParametros*** y desde esta comprobamos que la tenga mediante el método ***comprobarListaParametros***.

Producción idArray: tenemos que devolver el tipo de la expresión, que por inferencia ha de ser entero, ya que es una posición del array.

## 2. Generación de código intermedio

## 2.1. Descripción de la estructura utilizada.

Para la generación de código intermedio, se ha creado una clase llamada ***SoporteCI***, que está dentro del paquete ***soporte***; aquí se almacenarán todos los métodos necesarios para generar el código intermedio de cada una de las producciones de la gramática que lo precisen. El código intermedio que se ha utilizado es el que aparece en la documentación de Javier Vélez.

Se ha de destacar que se ha creado una instrucción de código intermedio, que controla el bucle ***FOR***. El nombre que se le ha dado a esta instrucción es ***CMP(***el nombre coincide con el de CF de Ens2001***)***, y los detalles e implementación de este código podemos verlos en la clase ***SoporteCI***, método *generarCIfor()*. Esta instrucción de código intermedio forma parte del conjunto de instrucciones necesarias para el manejo de la estructura iterativa.

## 3. Generación de código final

Para el código final, se ha implementado una clase llamada ***TraductorCF***, que se encuentra en el paquete ***code*** y a su vez dentro de ***compiler***. Esta clase contendrá los métodos para la traducción de las instrucciones de código intermedio a las instrucciones de código final correspondientes al simulador ENS2001.

Esta clase será llamada desde ***ExecutionEnvironmentEns2001***.

## 3.1. Descripción de donde se ha llegado

Se ha desarrollado todo excepto la parte de subprogramas en el código intermedio y código final.
Todas las estructuras parece que generan correctamente el código final, ejecutándose éste de forma adecuada en ENS2001, a excepción del archivo *testCase05.muned* correspondiente a las estructuras de array. He de comunicarles que me ha sido imposible conseguir que el código final de vectores se genere correctamente; el código final se genera, pero en muchos casos el comportamiento no es el esperado.

## 4. Indicaciones especiales.

*Bibliografía utilizada*


- *Compiladores: principios, técnicas y herramientas. Sethi, Ravi; Lam, Mónica S; Aho, Alfred V.; Ullman, Jeffrey.*
- *Construcción de compiladores: principios y práctica. Louden, Kenneth C.*
- *Compiladores: teoría e implementación. Jacinto Ruiz Catalán*
- *Teoría, diseño e implementación de compiladores de lenguajes. Martínez López, Francisco Javier; Ramallo Martínez, Alejandro.*
- *Manual Ens2001*
- *Transparencias de Javier Vélez Reyes.*