Implementación del Lenguaje

Daniel Rus (danirus@tol-project.org)

18 de julio de 2005

Índice

1. Analisis Léxico-Sintáctico **2**
1.1. Análisis léxico 3
1.1.1. Análisis Léxico en Tol 3
1.2. Análisis sintáctico 5
1.2.1. Tipos de Analizadores Sintácticos 5
1.2.2. Análisis Sintáctico Recursivo Descendente 6
1.2.3. Notación BNF 6
1.2.4. Descripción BNF de la Gramática de TOL 6
1.3. El árbol sintáctico 7
1.3.1. Clases Relacionadas 7

2. El núcleo del lenguaje **8**
2.1. Objetos Evaluables 9
2.1.1. Características de los Objetos Evaluables 9
2.1.2. Control del ámbito de variables y funciones 10
2.2. Control del contenido y tipo de los objetos 10
2.2.1. Obtención del tipo de objeto 11
2.2.2. Obtención del tipo de dato 11
2.2.3. Cálculo del valor del objeto 11
2.3. Evaluación Perezosa 12
2.3.1. Que hay dentro de un TimeSet y dentro de una Serie 12
2.4. Constantes y Variables del Sistema 13
2.5. Operadores y Funciones Built-in 13
2.5.1. Creación de un Operador 14
2.5.2. Creación de una función Built-in 14
2.6. Variables de Usuarios 15
2.7. Funciones de Usuario 16
2.7.1. El operador `#F#` de cada tipo de dato 16
2.7.2. El objeto evaluable 16

3. La Tabla de Símbolos **16**
3.1. Operaciones Básicas 17
3.2. Función Hash 18
3.3. Implementación de la TS 19
3.3.1. Clases C++ 19
3.4. Objetos contenidos en la TS 25
3.5. Inicialización de la TS 26
3.5.1. Inserción de objetos propios de Tol 26
3.6. Funciones para el manejo de la TS 26

3.6.1. Búsqueda de un objeto . . . 26
3.6.2. Adición de un objeto . . . 27
3.6.3. Borrado de un objeto . . . 31

4. El Evaluador de Tol **31**
4.1. Componentes . . . 32
4.2. Análisis Semántico . . . 32
4.3. Evaluación de funciones de usuario . . . 33
4.3.1. Evaluación de la declaración y definición . . . 34
4.3.2. Evaluación de la llamada . . . 35

5. Entorno de Ejecución **35**
5.1. Gestión del Ámbito . . . 35
5.1.1. Implementación . . . 36
5.1.2. Ámbito Estático y Ámbito Dinámico . . . 37
5.2. La Pila Local . . . 38
5.2.1. Control del nivel del ámbito en la Pila Local . . . 38
5.2.2. Control de los niveles de búsqueda en Scope Léxico . . . 38

6. Implementación de los Tipos de Datos **39**
6.1. Introducción . . . 39
6.2. Templates . . . 40
6.3. Tipos de Datos perezosos (Lazy) . . . 41
6.4. Tipos de Datos impacientes (Eager) . . . 41
6.5. Declaración de los Tipos de Datos . . . 41
6.5.1. Tipo Real . . . 41
6.5.2. Tipo Date . . . 41
6.5.3. Tipo Set . . . 42
6.5.4. Tipo Code . . . 42
6.5.5. Tipo Text . . . 42
6.5.6. Tipo Serie . . . 42
6.5.7. Tipo TimeSet . . . 42
6.5.8. Tipo Complex . . . 42
6.5.9. Tipo Matrix . . . 43
6.5.10. Tipo Polyn . . . 43
6.5.11. Tipo Ratio . . . 43

El contenido de éste documento bien podría constituye una referencia de la implementación del lenguaje para los programadores de Tol, sin embargo, se ha disminuido la densidad de las explicaciones y se han pasado por encima ciertos detalles de implementación con el fín de reducir su extensión y facilitar una lectura rápida y, si es posible, más agradable.

El documento comienza con el Análisis Léxico-Sintáctico, continua explicando las características generales del Núcleo del Lenguaje, aquel que implementa la representación de los objetos evaluables, es decir, los objetos que representan las unidades evaluables para el programador de Tol. La implementación de la Tabla de Símbolos es objeto de estudio en la sección 3, y el Evaluador de Expresiones, la columna dorsal del entorno de ejecución, se estudia en el capítulo 4. Por último se analiza el proceso de implementación de los Tipos de Datos.

1. Analisis Léxico-Sintáctico

En ésta sección se analiza el funcionamiento del proceso de reconocimiento sintáctico. Es el primer proceso en intervenir al ejecutar un programa Tol. El intérprete cede el control al Parser, que analiza el código fuente, separando los comentarios del código, identificando las palabras reservadas y las sentencias que forman el programa y elaborando al final del proceso un árbol sintáctico. El árbol sintáctico construido

será analizado, fuera del proceso de Análisis Léxico-Sintáctico, por el Evaluador de Expresiones de Tol en la fase siguiente, la del Análisis Semántico.

El punto de entrada al Parser del intérprete se localiza en la función `MultyEvaluate`, definida en el fichero fuente language.cpp, en la raíz de los fuentes de Tol. `MultyEvaluate` solicita la intervención del Parser entregándole el `BText` con el código fuente a analizar.

En el proceso de elaboración del árbol sintáctico de TOL intervienen dos procesos: Análisis Léxico y Análisis Sintáctico.

1.1. Análisis léxico

El ánalisis léxico se lleva a cabo por el proceso conocido como "Scanner". Éste interviene cada vez que el analizador sintáctico requiere un nuevo Token del código fuente. El analizador léxico devuelve al analizador sintáctico el siguiente cada token leído secuencialmente del código fuente. Para ello el analizador léxico debe saltar los comentarios, espacios en blanco y tabuladores que encuentre en el código fuente.

Cada Token devuelto por el analizador léxico al analizador sintáctico lleva asociado dos atributos de gran importancia: el tipo de token devuelto y su valor. El tipo de Token puede tomar uno de los siguientes valores: `NONE`, `OPEN`, `CLOSE`, `SEPARATOR`, `MONARY`, `BINARY`, `ARGUMENT`, `FUNCTION` y `TYPE`, definidos en btoken.h (tol/bparser)

> Un analizador léxico para un lenguaje **`L`** está formado por un proceso reconocedor tal que para una cadena de entrada "**`x`**" responderá "Si Reconozco" si "**`x`**" es una sentencia de **`L`** y "No Reconozco" en caso contrario.

Uno de los métodos tradicionales para la implementación de reconocedores sintácticos consiste en la programación de Autómatas Finitos Deterministas. Otro de los métodos que en determinados casos es más eficiente, detecta y localiza palabras reservadas e identificadores basándose en la búsqueda dicotómica en una lista ordenada de palabras y símbolos reservados. Si la búsqueda tiene éxito, el token es una palabra reservada o un símbolo del lenguaje; si no, por exclusión, será un identificador. El analizador léxico indicará así al analizador sintáctico el tipo del token leído, para poder hacer el tratamiento adecuado. A partir de entonces el analizador sintáctico retomará su trabajo y elaborará el árbol sintáctico.

1.1.1. Análisis Léxico en Tol

El análisis léxico en Tol consta de dos fases.

La **primera fase** consiste en la inicialización del analizador léxico. En dicha inicialización se filtra el código a analizar eliminando comentarios, espacios sobrantes, tabulares y saltos de línea.

En la **segunda fase** el analizador léxico lee cada token separado por espacios del código filtrado en la fase anterior y se lo entrega al analizador sintáctico indicándole su tipo.

Durante la inicialización del Analizador Léxico se crea una matriz, conocida como Tabla de Tokens, la cual contiene todas las palabras reservadas y los símbolos reconocibles por el lenguaje. Es parte de la clase `BScanner`.

Descripción de la primera fase

En la primera fase, el analizador léxico filtra el código analizando carácter a carácter la entrada.

Primero se divide el código en bloques separando los comentarios, las instrucciones y los literales. Los bloques de comentarios son eliminados del texto final. Los bloques de instrucciones se filtran verificando que cada cadena de entrada es un identificador o un símbolo válido. Los identificadores y símbolos validos de TOL vienen definidos por las siguientes expresiones regulares:

Identificadores Válidos ->(letra|digito)*

Símbolos Válidos ->('|(|)|{|}|[|]|;|,|:|-|!|+|*|/|%|&|||<|=|>|@|\\|\n|\r|\t|?)

Los identificadores y símbolos se separan entre sí por espacios. El proceso de filtrado termina devolviendo el código de entrada sin comentarios. Todo el proceso de filtrado se lleva a cabo por la clase `BFilter`.

Sustituciones previas

Durante el proceso de filtrado de código, realizado en la primera fase del Análisis Léxico, se realizan varias sustitutiones con el fin de reducir el número de comparaciones necesarias para seguir una construcción gramatical. Esta tarea, el seguimiento de una estructura gramatical determinada, se lleva a cabo en el Analizador Sintáctico. Reducir el número de comparaciones hace más rápida la identificación del código que se está analizando, además de hacer más sencilla la implementación de la gramática del lenguaje.

Las sustituciones que se están realizando tienen lugar en el método `Clean` de la clase `BFilter`. Existe otro código en `BFilter` que se usaba con esa finalidad, pero que en la actualidad no es empleado: los dos métodos `AnyReplace y Replace` así como la estructura de datos `replaceTable`.

Las sustituciones que se llevan a cabo en el filtrado son:

Secuencia Leída	Secuencia de Substitución
“ `) {` “	“ `) #F# {` “
“ `[[` “	“ `SetOfAnything #{#` “
“ `[` “	“ `#E# #(#` “
“ `]]` “	“ `#}#` “
“ `]` “	“ `#)#` “

Descripción de la segunda fase

En la segunda fase, el analizador léxico analiza el código devuelto trás el proceso de filtrado. Este proceso es conocido como Scanner. El Scanner obtiene cada token del código filtrado. Cada token esta separado de los demás tokens por espacios. El scanner busca la palabra o símbolo leído en la Tabla de Tokens elaborada en la inicialización, con el fin de determinar si la palabra leída es un símbolo o palabra reservada del lenguaje. De lo contrario la palabra leída será un argumento.

En definitiva, el Scanner de Tol se encarga, a petición del Parser, de gestionar los tokens que se van leyendo del código fuente.

A continuación se presentan las clases que intervienen en estas tareas y las estructuras de datos empleadas.

Clases relacionadas

Las clases relacionadas con el Análisis Léxico son:

Clase	Responsabilidades
BFilter	Es la clase responsable de las tareas de filtrado del código fuente. Se encarga de quitar los comentarios e introducir espacios entre los identificadores válidos y los símbolos del lenguaje. El punto de entrada es el método **Transform()** **BFilter** devuelve el código fuente sin comentarios, con los identificadores y símbolos separados por espacios.
BToken	**BToken** encapsula un Token general del lenguaje TOL. Existen varios tipos de tokens definidos mediante subclases de la clase **BToken**, que los representan: **BTypeToken**, **BSeparatorToken**, **BOpenToken**, **BCloseToken**, **BMonaryToken** y **BBinaryToken**. Todos los objetos **BToken** y sus derivados se crean en la fase de iniciación del análisis léxico. La estructura de datos en la que residen es la Tabla de Tokens: BScanner::PutSymbolTable() `http://tol.bayesforecast.com/tolapi/classBScanner.html#f0`. El scanner busca en esta tabla los identificadores y símbolos que encuentra en el código, con el fín de determinar si la palabra o símbolo leído, bien forma parte del lenguaje o bien es un argumento o nombre de función.
BScanner	Es la clase responsable del análisis léxico. Define y crea la Tabla de Tokens del lenguaje TOL. Invoca al filtro para limpiar el código antes de comenzar a trabajar para el parser. Su tarea más importante consiste en extraer los identificador y símbolos del lenguaje, clasificándolos como tokens y pasándoselos de uno en uno al parser a medida que éste los solicita.

1.2. Análisis sintáctico

El análisis sintáctico para una gramática **G** es la fase del proceso de análisis de código que toma como entrada una cadena o conjunto de sentencias, y produce, bien el árbol sintáctico de esa cadena, o bien un mensaje de error que indica que la cadena no es una sentencia perteneciente al lenguaje generado por la gramática **G**.

1.2.1. Tipos de Analizadores Sintácticos

Existen dos tipos básicos de analizadores para las gramáticas libres de contexto[1]: los top-down o descendentes, y los bottom-up o ascendentes. Los ascendentes intentan reconstruir el árbol desde las hojas hasta la raiz, mientras que los descendentes comienzan por la raíz y bajan hasta las hojas.

Un analizador ascendente tratará de reconstruir el árbol sintáctico para cada cadena de entrada, empezando por las hojas hasta llegar a la raíz. El proceso se suele considerar una reducción de una cadena de símbolos al símbolo inicial de la gramática, es decir, una derivación en sentido inverso. Un analizador **LR(k)** es un analizador sintáctico ascendente limitado al analisis de '**k**' símbolos.

Un analizador descendente hará el trabajo contrario; tratará de reconstruir el árbol sintáctico empezando por la raiz y llegando hasta las hojas. Los analizadores sintácticos descendentes se conocen como **LL(k)**. Son más sencillos de implementar que los **LR(k)**.

TOL utiliza una técnica de Análisis Recursivo Descendente LL(2).

[1]Una Gramática Libre de Contexto, es un conjunto finito de variables, cada una de las cuales representa un lenguaje. Los lenguajes representados por las variables se describen recursivamente en términos de otros lenguajes o símbolos primitivos llamados *terminales*. Las reglas que describen el lenguaje asociado con cada variable se llaman *producciones*.

Una Gramática Libre de Contexto (GLC) viene dada por una 4-tupla $G = (VN, VT, S, P)$ donde VN y VT son conjuntos finitos de símbolos no terminales y terminales respectivamente, y donde S es el símbolo inicial, que pertenece a VN y P es el conjunto de producciones. Por ejemplo:

$S \rightarrow 0B, S \rightarrow 1A, A \rightarrow 0S, A \rightarrow 1AA, A \rightarrow 0, B \rightarrow 1B, B \rightarrow ABB, B \rightarrow 1$

1.2.2. Análisis Sintáctico Recursivo Descendente

El mecanismo de análisis basado en un analizador recursivo descendente tiene las siguientes dos características:

1. Existirá un procedimiento o función por cada símbolo no-terminal
2. Dada una producción, y una vez que estamos en el procedimiento que trata el símbolo no-terminal de la parte izquierda, para analizar la parte derecha se hará lo siguiente:
   a) Si hay alternativas, llamar al analizador léxico para obtener un nuevo token del texto fuente. Una vez leído, se compara el token con los símbolos directores de dichas alternativas. Si el token pertenece a uno de esos conjuntos de símbolos directores, sigue el análisis por el procedimiento que trate la alternativa correspondiente. Si no pertenece, se tratará de un error sintáctico.
   b) Si no hay alternativas, o bien ya dentro de una alternativa, el análisis será el siguiente:
      1) Si el primer símbolo es un no-terminal, llamar al procedimiento correspondiente.
      2) Si es un terminal, llamar al analizador léxico y seguir el análisis si el token leído coincide con dicho símbolo terminal.

1.2.3. Notación BNF

En el siguiente apartado se muestra la gramática de TOL en formato BNF. BNF es la abreviatura de Backus-Naur Form, y fue utilizada para describir el ALGOL 60 [Back59, Naur63]. Algunas definiciones para entender la notación:

TERMINAL Un símbolo es terminal cuando tiene entidad propia y se describe por si mismo, es decir, no requiere ninguna explicación.

NO TERMINAL Un símbolo es no terminal cuando requiere una explicación mediante una regla o producción.

METASÍMBOLOS Son aquellos símbolos de la notación utilizados para distinguir los elementos y propiedades de una regla. Por ejemplo '|'.

Cuando se describe una gramática en notación BNF, se utilizan una serie de reglas y producciones cuyo objetivo es la descripción de unidades sintácticas o símbolos no terminales. Los símbolos no terminales van encerrados entre los símbolos '<' y '>', y deben aparecer en alguna regla en la parte izquierda.

1.2.4. Descripción BNF de la Gramática de TOL

Gramática sin reducir (tal y como la intérpreta el Parser, en par.cpp):

```
    <SENTENCE> ::= <PARSE_SYMBOL>

<PARSE_SYMBOL> ::= <PARSE_NONE>
                 | <PARSE_BLOCK> <SENTENCE>
                 | <PARSE_SEPARATOR> <SENTENCE>
                 | <PARSE_TYPE> <SENTENCE>
                 | <PARSE_MONARY> <SENTENCE>
                 | <PARSE_BINARY> <SENTENCE>
<PARSE_BINARY> ::= <ARGUMENT> <BINARY_SYMBOL>
<PARSE_MONARY> ::= <ARGUMENT> <PARSE_BINARY>
                 | <BINARY_SYMBOL>
                 | <MONARY_SYMBOL>
  <PARSE_TYPE> ::= <TYPE_SYMBOL>
```

```
<PARSE_SEPARATOR> ::= <ARGUMENT> <SEPARATOR_SYMBOL>
    <PARSE_BLOCK> ::= <ARGUMENT> <BLOCK>
          <BLOCK> ::= “(“ <SENTENCE> “)”
                    | “[” <SENTENCE> “]”
                    | “{” <SENTENCE> “}”
                    | “#{#” <SENTENCE> “#}#”
                    | “#(#” <SENTENCE> “#)#”
     <PARSE_NONE> ::= <ARGUMENT> vacio

       <ARGUMENT> ::= Identificador_Válido
                    | <vacio>
    <TYPE_SYMBOL> ::= “Anything” | “Set”     | “Date”  | “Real”
                    | “Complex”  | “Matrix”  | “Polyn” | “Ration”
                    | “Text”     | “TimeSet” | “Serie” | “Code”
<SEPARATOR_SYMBOL> ::= “,” | “;”
   <MONARY_SYMBOL> ::= “!” | “Do” | “Struct”
   <BINARY_SYMBOL> ::= “=”   | “#F#” | “|”   | “&”   | “Of”
                    | “@”   | “>”   | “>=”  | “<=”  | “:>”
                    | “<:”  | “!=”  | “#+#” | “#-#” | “/”
                    | “%”   | “^”   | “**”  | “&&”  | “||”
                    | “:”   | “>>”  | “->”  | “#E#” | “==”
                    | “+”   | “-”   | “*”   | “<”   | “<<”
```

1.3. El árbol sintáctico

Se puede hacer una representación gráfica de las derivaciones con lo que se denomina Árbol Sintáctico. Con este método representamos explícitamente la jerarquía de la estructura sintáctica de las sentencias generadas por la gramática.

Cada nodo interior del árbol estará etiquetado por un símbolo no-terminal, y los nodos hijos estarán etiquetados, de izquierda a derecha, por los símbolos de la parte derecha de la producción en la cual el nodo padre sea la parte izquierda de la produción.

En el caso del parser de TOL, se crea un árbol sintáctico en memoria que es recorrido por el analizador semántico.

1.3.1. Clases Relacionadas

Las clases relacionadas con la creación del árbol sintáctico de TOL son:

Clase	Responsabilidades
List	El árbol sintáctico se almacena en un objeto de tipo List. List es una clase con dos atributos. El atributo car_ es un BCore* usado para apuntar a cada objeto BToken del árbol, y cdr_ es un List*, que representa la cola de la lista. La representación del árbol como List le corresponde a Tree.
Tree	Tree es la clase que maneja el árbol mientras se crea. Define dos atributos para ello: tree_ y mrFree_, los dos de tipo List*. El primero apunta al árbol sintáctico que se está creando. El segundo se usa para acceder a la posición más a la derecha del árbol, lugar donde se van insertando los nodos.
BToken	El árbol sintáctico tiene nodos y ramas. Cada nodo es una instancia de una clase BToken o una derivada de BToken (BTypeToken, BSeparatorToken...)
BParser	Es la clase responsable de elaborar el árbol sintáctico. El flujo de ejecución de los métodos del parser, iniciados en el método Parse() y continuados en ParseSymbol() representan la implementación en c++ de las derivaciones de la gramática en notación BNF.

List

Implementa una Lista tipo Lisp, con un elemento car_, que representa la cabeza de la lista, y un elemento cdr_, que representa la cola de la lista.

Esta dotada de los clásicos métodos get y set y otros métodos para extraer el n-ésimo elemento de la lista, calcular su longitud, duplicar la lista, o copiarla en un elemento BList. BList es la antigua implementación de Listas, ampliamente usada en todo el código fuente de TOL.

Tree

Implementa el comportamiento que el Parser necesita para crear el árbol sintáctico. Utiliza dos atributos: tree_ y mrFree_. Ambos son de tipo List*. El primero almacena el árbol sintáctico y el segundo es un puntero a la posición libre más a la derecha (de ahí el nombre mrFree_ -> Most Right Free).

Los métodos más importantes son los 4 siguientes:

```
static Tree* create (BCore* rootElement, List* branch);
static Tree* createMonary (BCore* rootElement, List* branch);
static Tree* createBinary (BCore* rootElement, List* leftBranch,
                          List* rightBranch);
BBool putMostRight (Tree*);
```

Estos cuatro métodos son invocados por el Parser en el proceso de creación del árbol sintáctico.

Una vez que se ha creado el árbol sintáctico, se extrae el objeto List* que lo almacena y se accede al contenido del mismo, habitualmente con los siguiente métodos, que extraen el contenido del Nodo, y de las ramas:

```
static List* treLeft (List*);
static List* treRight(List*);
static List* treNode (List*);
```

Además de estos métodos la clase **Tree** implementa otros en los que se apoya.

Parser

El parser, siguiendo la descripción BNF de la gramática, identifica los diferentes tipos de Tokens y lleva a cabo un comportamiento determinado para cada uno de ellos. Toda esta actividad se realiza por los métodos Parse, ParseSymbol, ParseOpen, ParseClose, ParseNone, ParseBinary, ParseMonary, ParseType y ParseSeparator, del Parser.

El árbol sintáctico verífica en cada derivación de la gramática el atributo complete_ y el tipo de token devuelto por el Scanner. El atributo complete_ controla el final de las derivaciones de la gramática. El tipo de token permite conocer la derivación gramatical que debe emplear el Parser para reconocer el token leído. Para realizar el reconocimiento del token, el Parser se apoya en dos atributos, nextSymbol_ y nextArgument_, que almacenan el token leído por el analizador léxico. La primera, nextSymbol_ almacena el token cuando ha sido reconocido como palabra reservada o símbolo del lenguaje, el resto de tokens son considerados argumentos y se almacenan en el atributo nextArgument_.

Tol requiere la lectura de dos tokens del código fuente para tomar una derivación gramatical. Esto implica que el analizador recursivo descendente sea de tipo LL(2).

2. El núcleo del lenguaje

Esta sección explica qué es y qué componentes forman el núcleo del Lenguaje. Veremos que son, desde el punto de vista de la implementación del Lenguaje, los Objetos Evaluable, los Tipos de Datos, las Constantes de Tol, las Variables del Sistema, las Variables de Usuario, las Funciones Built-in y las Funciones de Usuario. Algunos temas que quedan fuera por su entidad particular se estudian por separado en otros capítulos. Es el caso de la Tabla de Símbolos y el Evaluador.

2.1. Objetos Evaluables

Un Objeto Evaluable es todo aquel que está accesible por el programador de Tol. Puede ser cualquier variable del Lenguaje, una función de usuario, una constante, o una función Built-in de cualquier tipo de dato. Todos ellos son objetos evaluables del lenguaje, y comporten una interfaz común.

Un objeto evaluable se caracteriza, desde el punto de vista de la implementación de Tol, por ser hijo de la clase `BSyntaxObject`. Ésta es la clase básica que representa cualquier objeto evaluable. También llamado objeto sintáctico.

La clase `BSyntaxObject` es la clase básica que implementa las características y comportamiento que tendrán todas las constantes, variables y funciones de Tol. De esta manera se puede intuir que todos los tipos de datos tienen una relación directa con esta clase.

La Tabla de Símbolos y la Pila Local almacenan instancias de la clase `BSyntaxObject`, y el Evaluador utiliza únicamente referencias a objetos de ésta clase, añadiendo así una capa de abstracción en el núcleo del Lenguaje que permite llevar a cabo cualquier operación sobre los objetos evaluables sean estos constantes, variables o funciones de cualquier tipo de dato.

2.1.1. Características de los Objetos Evaluables

Las instancias de la clase `BSyntaxObject`, es decir, cada objeto evaluable, se caracteriza por poseer los siguientes atributos:

- Nombre
- Nombre local
- Descripción
- Ámbito del objeto
- Bit de sistema
- Parent

A continuación se describe cada uno de ellos:

Nombre es el identificador del objeto. Coincide siempre con el nombre que recibe en el momento en el que se crea. Si se está creando una variable de usuario, el nombre del objeto evaluable coincide con el nombre que recibe la variable al ser definida por el programador. Del mismo modo ocurre si se está creando una función de usuario. El nombre que da el programador a la función se utiliza para dar nombre al objeto evaluable que la representa internamente.[2]

Nombre Local es una característica empleada únicamente cuando una variable se pasa como parámetro en la llamada a una función. En ese caso la variable puede recibir un nombre distinto como parámetro de la función, y éste es almacenado como nombre local del objeto evaluable (en Tol las variables se pasan siempre como referencia, nunca como valor, por lo que una expresión de asignación con el parámetro en el lado izquiero de la misma tiene como consecuencia un cambio de valor en la variable fuera del ámbito de la función. Ver 2.7).

Descripción es un texto explicativo sobre el objeto evaluable. Se utiliza ampliamente con los objetos evaluables que proporciona Tol al programador, tales como constantes, variables del Lenguaje y funciones Built-in de los tipos de datos. Sin embargo, el programador también puede añadir una descripción a las variables y funciones creadas durante el programa utilizando la función PutDescription(Text). Las interfaces de programación Tol reservan siempre un espacio donde mostrar la descripción de los objetos creados, por lo que este atributo puede ser de gran utilidad.

[2]Éste atributo pertenece realmente a la clase `BObject`, pero su utilidad en el núcleo se encuentra asociada siempre con la clase `BSyntaxObject`, por lo que en la exposición de este apartado del documento no se habla de su origen. La clase `BSyntaxObject` hereda de la clase `BObject`. Ver BObject Class Reference `http://www.tol-project.org/tolapi/classBObject.html`

Ámbito del objeto es un número entero, que toma el valor 0 cuando el objeto evaluable pertenece al ámbito global del programa y toma un valor mayor que 0 cuando pertenece a algún ámbito local. Las constantes, variables y funciones Built-in de Tol pertenecen todas ellas al ámbito global. Se pueden invocar desde cualquier punto del programa, sea este el ámbito global o un ámbito local de cualquier profundidad. Éste atributo toma valor directamente desde el Evaluador (sección 4) en el momento en el que el objeto evaluable va a ser creado, trás haber sido interpretada una sentencia del árbol sintáctico.

Bit del sistema se utiliza en la preinicialización del Lenguaje para indicar qué objetos son propios de Tol y distinguirlos así de los creados por el programador. En la práctica no tiene utilidad como tal y el hecho de que esté activo no supone ningún cambio en el comportamiento del Lenguaje. Su existencia se explota para saber que objetos deben ser hechos globales después de la ejecución de la función `MakeGlobal` de Tol.

Parent también conocido como el fichero fuente de procedencia, es el objeto evaluable que representa al fichero fuente de código Tol donde se ha creado este objeto evaluable. Mediante el grupo de sentencias `Include` del tipo de datos `Set` es posible incluir ficheros de código en el flujo de ejecución de un programa Tol. Cada vez que se ejecuta una sentencia `Include`, se crean las variables y funciones que en él aparecen, tomando estas como atributo Parent el fichero en el que se han definido.

2.1.2. Control del ámbito de variables y funciones

Las variable o funciones que aparecen en el código fuente de un programa Tol están representadas, trás crearse éstas, mediante objetos `BSyntaxObject`. En el momento en el que el objeto `BSyntaxObject` es creado, se le asigna un valor al atributo "ámbito del objeto", internamente conocido como `level_`. El `level_` del objeto refleja el ámbito de código en el que se ha creado. De modo que valores mayores que 0 indican que el ámbito de código en el que se ha definido la variable o función es un ámbito local, y un valor 0 indica que el ámbito es global.

El ámbito de la variable o función entra en escena poco después de ser creada para determinar si debe ser añadida a la Tabla de Símbolos (3) o a la Pila Local (5.2). Si el ámbito o `level_` es 0, es decir, global, la variable o función será añadida a la Tabla de Símbolos, quedando así accesible desde cualquier ámbito en el menor tiempo posible. Si el ámbito o `level_` es mayor que 0, será añadida a la Pila Local. En ésta los elementos se organizan uno detrás de otro, utilizando una lista enlazada (ver 5.2).

Función de Tol: `MakeGlobal`

El valor del ámbito de una variable creada explicítamente por el programador puede cambiar (únicamente) de un ámbito local a un ámbito global mediante el uso de la función `MakeGlobal` del tipo de datos `Anything`. Esta función aplicada a una variable o función local, cambia su atributo `level_` estableciéndolo al valor 0, y la añade a la Tabla de Símbolos verificando previamente que no exista un objeto evaluable con el mismo nombre y el mismo tipo de datos.

2.2. Control del contenido y tipo de los objetos

Dado que la interfaz general que utiliza el nucleo de Tol para acceder a cualquier objeto evaluable es la clase `BSyntaxObject`, ésta provee a los algorítmos del núcleo del Lenguaje de un conjunto de métodos con los que obtener toda la información necesaria de cada objeto.

Los siguientes comportamientos de `BSyntaxObject` son utilizados en el núcleo del Lenguaje:

- Obtención del tipo de objeto
- Obtención del tipo de dato
- Calculo del valor del objeto
- Obtención de información general del objeto

2.2.1. Obtención del tipo de objeto

Tol permite al programador crear Variables y Funciones, pero internamente cuenta con mas tipos de objetos. Cada uno está identificado por una constante numérica, la cual permite que sean distinguidos en el Evaluador y en la Tabla de Símbolos. Los tipos de objetos que puede haber en Tol son:

OBJECTMODE empleado con constantes, variables del sistema y variables de usuario.

OPERATORMODE empleado con funciones de usuario, operadores binarios y funciones Built-in.

STRUCTMODE empleado para distinguir en el análisis semántico si el token leído en el árbol sintáctico representa una estructura de datos especial "`struct`".

BUSERFUNMODE empleado para distinguir entre operadores (funciones Built-in) y funciones de usuario.

El resto de tipos de objeto definidos en BSyntaxObject (btol/bgrammar/syn.cpp) están en desuso. Son `NOMODE`, `IMAGEMODE`, `GRAMMARMODE`, `FIELDMODE`, `METHODMODE` y `CLASSMODE`. Pero no se usan en el funcionamiento interno de Tol.

La operación de obtención del tipo de objeto tiene utilidad al acceder a la Tabla de Símbolos, durante el proceso de Análisis Semántico. Dependiendo del tipo de Token leído del árbol sintáctico, el Análisis Semántico buscará objetos evaluable bien del tipo `OBJECTMODE`, bien del tipo `OPERATORMODE`, o bien del tipo `BUSERFUNMODE`.

El método (pure virtual, es decir, implementado únicamente en clases hijas de `BSyntaxObject`) que devuelve el tipo de objeto es:

```
virtual int Mode() const=0
```

2.2.2. Obtención del tipo de dato

Esta operación retorna un puntero al objeto `BGrammar` que representa el tipo de dato del objeto evaluable sobre el que estamos interesados. Es usado muy ampliamente en el núcleo del lenguaje para:

1. Insertar, buscar y eliminar objetos evaluables de la Tabla de Símbolos y de la Pila Local, y
2. Crear variables de usuario de cada tipo de dato.

Tol tiene 12 tipos de datos. El método que devuelve el tipo de dato es:

```
virtual BGrammar *Grammar() const
```

2.2.3. Cálculo del valor del objeto

Cada sentencia completa que lee el analizador semántico tiene como resultado el cálculo de al menos un objeto evaluable, sin embargo, en una sola sentencia Tol pueden intervenir múltiples objetos evaluables, tales como operadores binarios (como el símbolo de asignación "="), funciones Built-in, funciones de usuario, y parámetros con llamadas a otras funciones, en definitiva una cascada de objetos relacionados unos con otros, de manera que el resultado final de la sentencia es consecuencia de aplicar reiteradamente la función de cálculo de cada uno de los objetos evaluables.

En la sentencia Tol siguiente, intervienen 3 objetos evaluables:

```
Real a = 1;
```

- El operador "=" es el primero objeto evaluable que interviene (clase `BEqualOperator`).
- El identificador entero "1" se crea a continuación (clase `BGraContens<BDat>`).
- Por último se crea la variable "a" de tipo `Real`, la cual contiene en su interior al anterior objeto representado por el identificador entero "1" (es de la clase `BRenContens<BDat>`).

El ejemplo ilustra cómo una simple sentencia Tol crea múltiples objetos evaluables.

Al final de cada sentencia se invoca, sobre el objeto resultado, que en el caso del ejemplo anterior es la variable "a" de Tipo `Real`, al método `virtual void Do()` para calcular su valor final. El método `Do()` es invocado directamente por el Analizador Semántico y también desde el método `MultyEvaluate(BText)`, que sirve de nexo entre las interfaces de usuario de Tol y el Lenguaje.

La utilización del método `Do()` activa el cálculo del valor de la variable creada, pero esto no ocurre siempre debido a que dos tipos de datos utilizan Evaluación Perezosa.

2.3. Evaluación Perezosa

Actualmente Tol cuenta con dos tipos de datos de evaluación perezosa. Son los tipos `TimeSet` y `Serie`. Las variables de estos dos tipos de datos no toman valor hasta el momento en el que éste es explícitamente requerido. El método `Do()` hace que todas las variables de Tol salvo las que pertenecen a estos dos tipos, sean calculadas.

El comportamiento del método `Do()` aplicado a variables `TimeSet` y `Serie` difiere del resto y también entre sí.

- El tipo `TimeSet` no cuenta con el método `Do()`. Al ser invocado sobre una variable `TimeSet` es el `Do()` de la clase básica `BSyntaxObject`, que esta vacío, el que se activa.
- El tipo `Serie` si cuenta con él, pero una llamada al mismo no activa el cálculo del valor completo de la variable Serie, incluso aunque haya sido definida entre fechas de inicio y fin.

Las tres expresiones que aparecen a continuación crean variables de evaluación perezosa. Ninguna de ellas toma valor después de ser evaluada. De manera que lo único que sabe Tol de éstas es su definición, pero no su valor:

```
TimeSet tms = WD(2);
Serie ser1 = Gaussian(0,1,tms);
Serie ser2 = SubSer(ser1, y2000, y2003);
```

La evaluación de la primera expresión termina con una llamada al método Do() de la clase BSyntaxObject. Do() esta declarado como virtual en BSyntaxObject, y tiene un comportamiento vacío en la clase base. Debido a que TimeSet no implementa éste método, al terminar de ejecutarse la sentencia, la variable tms, que representa "Todos los Martes", no será más que su propia declaración.

La evaluación de la segunda expresión crea una variable Serie de tipo Gaussian, que guarda los parámetros η, σy el conjunto temporal que define los instantes de tiempo para los que toma valor. Más allá de registrar éstos parámetros, no ocurre nada.

En la evaluación de la tercera expresión se declara una variable Serie que representa una subserie de la anterior, para lo que se establecen las fechas de inicio y fin entre las cuales la Serie existirá. El conjunto temporal que representa los instantes de tiempo en los que la Serie toma valor, es el mismo que en la Serie original. Tampoco se calculará la ristra de valores de la Serie, solo se anotan los parámetros usados en su definición.

2.3.1. Que hay dentro de un TimeSet y dentro de una Serie

Tol permite al programador forzar el cálculo de una variable perezosa mediante funciones propias de estos tipos de datos. La función `Succ` del tipo `TimeSet` devuelve el `Date` sucesor o predecesor de otro dado como parámetro. La función no desencadena un cálculo masivo de la variable perezosa, solo calcula el valor que debe retornar en función del parámetro pasado en la expresión. De esta manera el lenguaje no se demora calculando el conjunto de todos los instantes de tiempo que representa un `TimeSet`:

```
TimeSet losMartes = WD(2);
Date proximoMartes = Succ(Now, losMartes);
```

Crea una variable proximoMartes que contiene el sucesor en el conjunto losMartes del instante de tiempo actual, Now. Succ admite un tercer parámetro entero, que al tomar valores negativos fuerza el cálculo de los valores predecesores. El 2º Martes predecesor del instante actual se calcula así:

```
Date hace2Martes = Succ(Now, losMartes, -2);
```

Asociadas al tipo de datos Serie, la función SerDat devuelve el valor que toma una serie en un momento dado del TimeSet en la que se define. Al igual que ocurre con la función Succ de TimeSet, ésta tampoco desencadena un cálculo masivo del valor de la Serie. Solo se obtiene el valor solicitado:

```
Real valProximoMartes = SerDat(ser1, proximoMartes);
Real valHace2Martes = SerDat(ser1, hace2Martes);
```

2.4. Constantes y Variables del Sistema

Existe una amplia variedad de variables globales en Tol, precreadas por el intérprete durante la inicialización del lenguaje. Cada tipo de dato define sus constantes y variables globales en el método InitInstances del template BGraContens<Any>, si es Eager, o del template BGraObject<Any>, si es Lazy.

Los Tipos de Datos Eager cuentan con dos templates para crear constantes y variables del sistema: son los templates BGraConstant<Any> y BGraParameter<Any>. Por otro lado, los tipos Lazy no cuentan con ningún templates especializado para éste tipo de objetos evaluables.

Tol cuenta con el operador :=, u operador de reasignación. El hecho de que exista tal operador hace que sea importante diferenciar entre constantes y variables globales. En la práctica, hasta la version actual de Tol, la v1.1.3, no existe diferencia entre constantes y variables globales. Los dos templates de los tipos Eager son prácticamente idénticos, y no permiten diferenciar lo que es una constante de una variable. El operador := permite reasignar valores tanto a variables globales como a las supuestas constantes. Como ejemplo, es posible reasignar el valor de la constante True de Tol, simplemente usando el operador de reasignación:

```
Real True := 0;
```

Como regla, todos los tipos permiten que sus variables, ya sean éstas creadas durante la inicialización de Tol, o bien creadas por el programador, sean reasignadas a través del operador de reasignación. La excepción de la regla es el tipo TimeSet. El operador :=, implementado en la función EvPutValue, en spfuninst.cpp, impide llevar a cabo reasignaciones en variables de éste tipo de datos.[3]

2.5. Operadores y Funciones Built-in

Desde el punto de vista sintáctico, los operadores puede ser unarios o binarios (ver **??**), y actuan sobre uno o dos operandos respectivamente. Desde el punto de vista de la implementación, tanto los operadores como las funciones Built-in, de las que se habla en el próximo apartado, son objetos evaluables de la clase BOperator.

La implementación de la clase BOperator y sus hijas se realiza en el fichero tol_boper.h, opr.cpp y oprimp.cpp donde también se definen un grupo de macros de preprocesamiento empleadas para crear los operadores y funciones Built-in de cada tipo de dato.

Dependiendo del tipo de dato de los operandos o argumentos del operador o función que se esta creando, se utiliza la macro DefIntOpr o DefExtOpr. La primera se utiliza para crear operadores o funciones built-in cuyos operandos o argumentos sean del mismo tipo que el objeto evaluable que se está creando, la segunda se utiliza para crear operadores o funciones Built-in cuyos argumentos pertenecen a distintos tipos.

En los dos siguientes apartados se muestran ejemplos de la creación de un operador y una función Built-in, ambos del tipo de Datos Real.

[3] No hay nada que impida modificar éste comportamiento en la función mencionada, permitiendo así entonces que todas las variables puedan ser reasignadas. El Lenguaje Tol fué, durante sus oscuros comienzos, un lenguaje funcional, pero con el tiempo se le han añadido características sintácticas propias de lenguajes imperativos, como ocurre con éste operador. En el actual Equipo de Desarrollo de Tol existen fundados ánimos para crear un nuevo lenguaje funcional, partiendo de una nueva especificación formal, que cubra los mismos objetivos que Tol y que invite al cambio a su actual comunidad de programadores.

2.5.1. Creación de un Operador

El operador + del tipo de datos Real, se crea utilizando la macro DefIntOpr, debido a que los dos operandos que emplea son también del tipo Real. El siguiente código, extraido del fichero datgra.cpp, en el directorio tol/btol/real_type, crea el operador + entre variables de tipo Real.

```
//---------------------------------------------------------------------
DeclareContensClass(BDat, BDatTemporary, BDatSum2);
DefIntOpr(1, BDatSum2, "+", 2, 2,
  "x1 + x2 {Real x1, Real x2}",
  I2("Returns the summe of both real numbers.",
     "Devuelve la suma de ambos números reales."),
  BOperClassify::RealArythmetic_);
//---------------------------------------------------------------------
void BDatSum2::CalcContens()
{
  contens_ = Dat(Arg(1)) + Dat(Arg(2));
}
```

En el código anterior se usan 2 macros de preprocesador y se implementa un método.

La primera macro, DeclareContensClass, definida en el fichero de templates tol_bgencon.h, crea la clase BDatSum2 heredando del comportamiento de la clase BDatTemporary, y declara dentro de BDatSum2 el método CalcContens(), método invocado para evaluar cada Operador o función Built-in. También crea la función evaluadora del operador, de manera que pueda ser invocado trás el proceso de análisis semántico. Ésta última se crea usando la macro DeclareEvaluator(BDatSum2), invocada dentro de la declaración de DeclareContensClass.

La segunda macro, DefIntOpr, crea un objecto de la clase BInternalOperator, necesario para que éste sea añadido a la API del tipo de datos Real, al inicializarse Tol. La macro crea el objeto a partir de una llamada al método __delay_init, usado masivamente en el proceso de inicialización de Tol para crear los operadores y funciones Built-in de todos los tipos de datos.

Resumiendo la utilidad de estas macros, DefIntOpr crea el código que enlaza al evaluador de expresiones con el operator +. DeclareContensClass crea la clase que encapsula el comportamiento que tendrá el operator +.

2.5.2. Creación de una función Built-in

La función SerDat del tipo de datos Real, permite obtener el valor de una Serie para una fecha dada. Su prototipo es:

```
Real SerDat(Serie ser, Date fecha)
```

Recibe una Serie en el parámetro ser y retorna el valor de esa Serie en el instante de tiempo indicado en el parámetro Date fecha. La implementación de la función es muy similar a la implementación del operator + visto en el apartado anterior. El codigo de implementación reside en el fichero datgrav.cpp, en el directorio tol/btol/real_type, y es el siguiente:

```
//-------------------------------------------------------
DeclareContensClass(BDat, BDatTemporary, BDatOfSerie);
DefExtOpr(1, BDatOfSerie, "SerDat", 2, 2, "Serie Date",
  I2("(Serie ser, Date date)",
     "(Serie ser, Date fecha)"),
  I2("Returns the value of a serie in a a date.",
     "Devuelve el valor de una serie en una fecha."),
  BOperClassify::TimeSeriesAlgebra_);
//-------------------------------------------------------
void BDatOfSerie::CalcContens()
```

```
{
   BUserTimeSerie& ser = *Tsr(Arg(1));
   BDate           dte = Date(Arg(2));
   BDat            dat = ser[dte];
   contens_            = dat;
}
```

La macro DeclareContensClass, igual que ocurría en el operator +, crea la clase que encapsula el comportamiento de la función Built-in. El comportamiento se implementa en el método CalcContens de dicha clase, definido a continuación de las macros.

La macro DefExtOpr crea el BOperator que permite que la función sea invocada, junto con sus parámetros, desde el Evaluador de Expresiones de Tol. El código del BOperator encapsula el tratamiento de los argumentos de la función.

2.6. Variables de Usuarios

Cada uno de los tipos de datos de Tol esta representado por una instancia de la clase BGrammar. En ésta documentación se usa indistintamente el termino Tipo de Dato o Gramática.

Cada Gramática implementa un método que permite crear objetos del tipo de datos al que representa. Es el método New de la clase BGrammar. Éste método hace uso del atributo newRenamed_, un puntero a función inicializado en el momento de creación de cada gramática, y distinto para cada tipo de dato:

1. En el tipo de dato **Real**, newRenamed_ es BRenContens<BDat>::New
2. En el tipo de dato **Complex**, newRenamed_ es BRenContens<BCmp>::New
3. En el tipo de dato **Matrix**, newRenamed_ es BRenContens<BMat>::New
4. En el tipo de dato **Date**, newRenamed_ es BRenContens<BDate>::New
5. En el tipo de dato **TimeSet**, newRenamed_ es BTmsRenamed::New
6. En el tipo de dato **Serie**, newRenamed_ es BTsrRenamed ::New
7. En el tipo de dato **Set**, newRenamed_ es BRenContens<BSet>::New
8. En el tipo de dato **Polyn**, newRenamed_ es BRenContens<BPol>::New
9. En el tipo de dato **Ratio**, newRenamed_ es BRenContens<BRat>::New
10. En el tipo de dato **Text**, newRenamed_ es BRenContens<BText>::New
11. En el tipo de dato **Code**, newRenamed_ es BRenContens<BCode>::New

Todas las gramáticas, a excepción de **TimeSet** y **Serie**, utilizan el método New del Template BRenContens<Any> (codificado en btol/bgrammar/tol_bgencon.h) para crear variables de Tol. Así pues, una variable Tol de Usuario está representada en memoria como un objeto BRenContens<Any>.

En el proceso de creación de éste objeto, el método BRenContens<Any>::New crea un objeto de la clase Any. Esta clase Any representa los atributos específicos de cada tipo de dato, de manera que al crear una variable **Real** de Tol, se estará creando una instancia de la clase BDat, que representa el comportamiento y los atributos de dicha variable Tol. Lo mismo ocurre con el tipo **Complex**, cuya representación la realiza la clase BComplex, o con el tipo **Set**, cuyo representación está a cargo de BSet.

2.7. Funciones de Usuario

Las funciones de usuario (explicadas en **??**), creadas por los programadores en Tol, son un caso especial de objeto evaluable. Una función de usuario presenta el aspecto visto en **??** para la función de Fibonacci, reimplementada en **??**.

Las funciones de usuario, a diferencia de los operadores y funciones Built-in de cada tipo de dato, no se crean en la inicialización del intérprete, sino mediante la evaluación del código escrito por el programador de Tol. El intérprete evalua el código que define a la función creada por el programador, creando a partir de entonces el objeto evaluable que representará tal función.

2.7.1. El operador `#F#` de cada tipo de dato

Las funciones de usuarios son variables del tipo de datos `Code`.

Cada vez que el interprete de Tol se encuentra con una declaración de función en el código fuente realiza un tratamiento especial que desemboca en la creación de una variable de tipo `Code`.

El tratamiento especial comienza en el Parser, el cual realiza unas sustituciones en los tokens leídos en el código fuente para crear un árbol sintáctico cuyo símbolo raiz es `#F#`. Este símbolo representa un operador para el evaluador de expresiones. Dicho operador se crea automáticamente en todos los tipos de datos en el proceso de inicialización del lenguaje.

Tal creación tiene lugar en la función `InitCommonInstances`, invocada una vez por cada tipo de dato, desde el método `InitGrammars`. Dentro de la función `InitCommonInstances` se crea un objeto `BCodeCreator`, el cual contiene el método llamado por el Evaluador al encontrar el token `#F#` en el árbol sintáctico.

Así, cada vez que el programador de Tol ejecuta el código que crea una función de usuario, algo que podría ser, por ejemplo:

```
Real fQuad(Real param) { param * param };
```

El parser mete un símbolo `#F#` en el árbol sintáctico que representa tal función, y el evaluador de expresiones invoca la función `Evaluate` del objeto `BCodeCreator` del tipo de dato de la función, en éste caso, `Real`.

2.7.2. El objeto evaluable

El método `Evaluate` de `BCodeCreator` verifica que el resto del árbol sintáctico de la función de usuario corresponda con una declaración correcta de parámetros y con una definición de función. Para ello actua la clase `BUserFunctionCreator`, cuyo método `Evaluate` comprueba lo anterior y devuelve un objeto `BUserFunction`.

Una vez se ha creado el objeto `BUserFunction`, éste se encapsula dentro de un `BUserFunCode`, las variables del tipo de datos `Code`, y se añade bien a la tabla de símbolos o bien a la pila local, para que esté disponible en el ámbito en el que se ha creado.

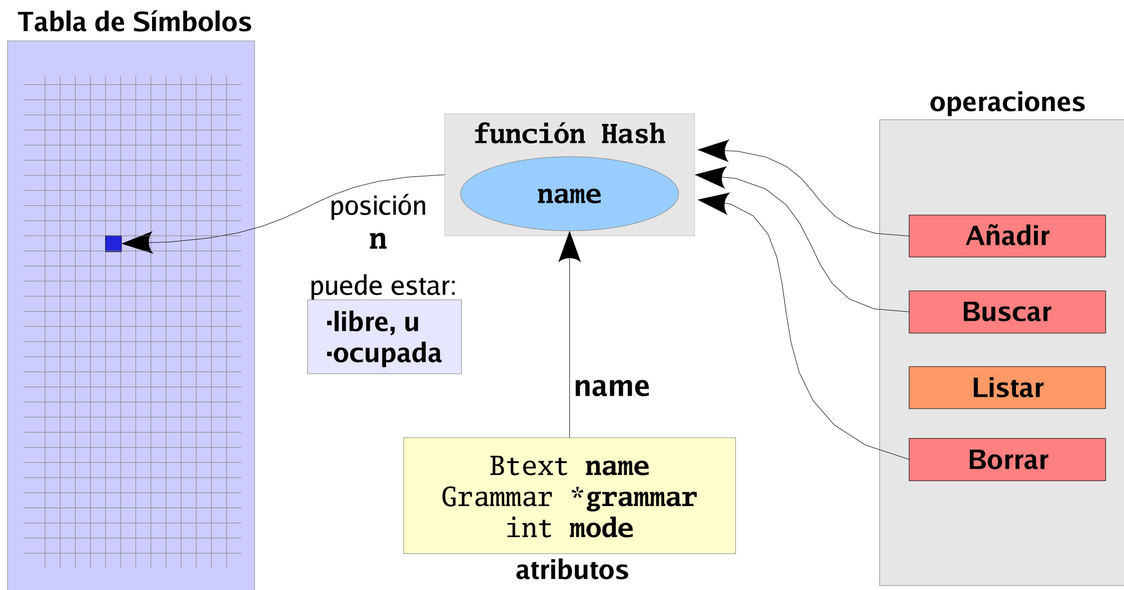
3. La Tabla de Símbolos

Un intérprete o compilador necesita guardar y usar la información de los objetos que van apareciendo en el código fuente. Esta información se introduce en una estructura de datos del lenguaje denominada Tabla de Símbolos.

Tol almacena en la Tabla de Símbolos, en adelante TS, una referencia por cada objeto creado por el Evaluador en el ámbito global. Los objeto almacenados en la Tabla de Símbolos presentan diferencias en cualquiera de los atributos Nombre, Tipo de Dato, o Tipo de Objeto:

Nombre Es el nombre que recibe en el código fuente el objeto. Puede ser el nombre de una función Built-in, el nombre de una constante, el nombre de una variable proporcionada por Tol, o puede ser el nombre de una variable o función creada por el usuario. Si dos objetos coinciden en el nombre,

Figura 1: Funciones Básicas de la Tablas de Símbolos



pueden coexistir dentro de la TS siempre que el *Tipo de Dato* al que pertecen o el *Tipo de Objeto* que representan sea distinto.

Tipo de Dato Es una referencia al objeto `BGrammar` que representa el tipo de dato del objeto que se ha creado. Puede ser cualquier de los tipos de Tol: `Anything`, `Code`, `Complex`, `Date`, `Matrix`, `Polyn`, `Ratio`, `Real`, `Serie`, `Set`, `Text` y `TimeSet`.

Tipo de Objecto Es una constante entera asociada a las macros de preprocesador `BNOMODE`, `BOBJECTMODE`, `BOPERATORMODE`, `BUSERFUNCMODE` y `BSTRUCTMODE`.

Durante la inicialización de Tol y hasta que éste comienza a evaluar el código fuente de un programa introducido por el usuario, se introducen objetos en la Tabla de Símbolos de cada Tipo de Datos. Son las variables, constantes y funciones Built-in que el usuario tendrá a su disposición para comenzar a programar.

Podemos considerar cada entrada en la Tabla de Símbolos como una 4-tupla de la forma (nombre, tipo de dato, tipo de objeto, objeto). Cada vez que se detecta un objeto en el código fuente, se busca en la Tabla de Símbolos para comprobar si ya existe. Si el objeto es nuevo, y pertenece al ámbito global, se introduce en la Tabla. Si pertence al ámbito local, el almacenamiento se realiza en la Pila Local (ver sección **??**).

La información contenida en la Tabla se utilizará principalmente en el análisis semántico, es decir, en el momento en que Tol comprueba si la declaración de un objeto es consistente con el uso que se le está dando.

3.1. Operaciones Básicas

Sobre la Tabla de Símbolos se pueden realizar operaciones de Inserción, Búsqueda, y Borrado. También permite obtener una lista con los objetos contenidos en ella. Ésta última función es usada ampliamente en las interfaces de usuario de Tol, Tolbase y Tolcon, para obtener una lista de las funciones y variables creadas para cada tipo de dato.

En las operaciones de Inserción, Búsqueda y Borrado, intervienen los tres atributos mencionados antes: *Nombre*, *Tipo de Dato*, y *Tipo de Objeto*. El primero de éstos, el *Nombre* del Objeto, interviene en el paso previo que la TS, aplica a las tres operaciones citadas, la función *Hash*:

- Aplicación de la función Hash al nombre del objeto, obteniendo una posición en la TS (ver la sección **??**).

Después de aplicar la función Hash al nombre del objeto se realizan distintas acciones en función de la operación a realizar:

Añadir un Objeto Si la operación a ejecutar es Añadir un Objeto a la TS, trás aplicar la función `Hash` es necesario comprobar si la dirección de retorno está libre, es decir, si la posición en la tabla que devuelve la función `Hash` esta vacía o no. Si está vacía, el objeto se añade en esa posición. Si la posición está ocupada, se comprueban el Tipo de Datos (método `BGrammar()`) y el Tipo de Objeto (método `Mode()`), para determinar si el objeto que ocupa la posición es del mismo Tipo de Datos y del mismo Tipo de Objeto. De ser así, la operación es rechazada y el objeto no se añade.

Buscar_un_Objeto Si se trata de una búsqueda, es necesario encontrar la posición que ocupa el objeto en la TS. La información disponible para ello es exclusivamente el nombre del objeto buscado. Sin embargo el contexto semántico de éste en el código fuente determina los otros dos atributos.
La búsqueda en la TS se realiza siempre que aparece en el código fuente un operador o identificador, ya sea para ejecutar la acción asociada al primero, o para crear o recuperar el segundo.

Borrar un Objeto La operación de borrado requiere que el objeto a borrar de la TS coincida en Nombre, Tipo de Dato y Tipo de Objeto con el objeto encontrado por la función Hash.

La figura 1 muestra un esqueña de las operaciones que se realizan en la TS y el cálculo de la posición a través de la función Hash. En las siguientes secciones se amplía la descripción de la función Hash y las operaciones básicas.

3.2. Función Hash

Tol utiliza multiples tablas Hash para la implementación de su Tabla de Símbolos debido al rendimiento que esta estructura de datos ofrece.

Una Tabla Hash se caracteriza básicamente por tener un espacio de almacenamiento de datos y una función de cálculo de la posición en dicho espacio. Existen distintos tipos de tablas Hash. La implementación clásica utiliza un área de overflow adicional al área de almacenamiento, donde se guardan los objetos que han colisionado en el área de almacenamiento principal.

La implementación de la Tabla Hash realizada en Tol no utiliza el área de overflow, en cambio esta preparada para utilizar más de un espacio de almacenamiento. La Tabla de Símbolos de Tol controla la densidad de elementos en cada Tabla Hash, de modo que si detecta un porcentaje de ocupación elevado crea un nuevo espacio de almacenamiento. De ahí la primera afirmación con la que comienza esta sección. Podemos ver la TS de Tol como un array de Tablas Hash.

El objetivo de una Tabla Hash es ofrecer una estructura de datos y una función de cálculo que permitan un acceso directo por clave al elemento almacenado. Para lograrlo se utiliza una función que aplicada al dato de entrada de la búsqueda retorna la posición del objeto asociado. El uso de Tablas Hash cobra especial sentido cuando el número de objetos con distinta clave que puedne existir es muy superior a las necesidades de almacenamiento requeridas.

La Tabla Hash de Tol esta implementada en la clase `BHashedArray` (ver subsección **??**), por lo que la función Hash forma parte de su comportamiento.

La función Hash de Tol

El objetivo general de toda función Hash es calcular la dirección en el área de almacenamiento que corresponde con la clave recibida como argumento. El argumento utilizado en la función Hash de Tol es el nombre del objeto. El prototipo de la función es el siguiente:

```
void Hash(const BText& name, BInt& h, BInt iter)
```

El parámetro `name` es la clave a la que se aplicará la función de conversión para obtener la posición. El parámetro `h` es la posición devuelta por la función. Y el parámetro `iter` representa el valor de la iteración en el cálculo de la posición en la tabla para una clave name dada.

Debido a que Tol permite sobrecarga de variables, funciones Built-in y funciones de usuario, pueden existir múltiples objetos con el mismo nombre, por lo que en tal caso, la posición devuelta puede estar ya ocupada con un objeto distinto al que pretendemos insertar o distinto al que buscamos, haciéndose necesario invocar la función sucesivamente.

Cuando el parámetro `iter` es 0, es decir, en la primera iteración, la función Hash aplica un algoritmo al parámetro name para obtener una posición `h`. Si el parámetro `iter` es mayor que 0, la función Hash aplica un algoritmo distinto, y más rápido, solamente a la dirección `h` previamente calculada. Es decir, a partir de la 2ª llamada a la función Hash, ésta se aplica exclusivamente sobre el parámetro `h`. Lo que implica recalcular una posición alternativa a la anterior.

El algoritmo

En la primera iteración, la función Hash aplica un algoritmo a los 64 primeros caracteres del identificador del objeto, es decir, al argumento `name` pasado a la función. Teniendo en cuenta que los identificadores en Tol pueden tener hasta 256 caracteres, se deduce que solo los 64 primeros son significativos. Si utilizamos identificadores con nombres de longitud superior a 64 caracteres deberíamos tener en cuenta que al menos los 64 primeros sean siempre distintos, para generar el menor número de colisiones posibles en la Tabla.

El algoritmo que se aplica pretende generar una posición distinta para cada nombre de entrada. En la práctica, distintos argumentos `name` pueden generar la misma posición en la tabla. Esto se debe a que el algorítmo devuelve siempre una posición en un área más pequeña que el área que ocuparían todos los posibles nombres.

Si se produce una colisión, es decir, un nombre dado genera una posición que ya esta ocupada por otro objeto, se vuelve a invocar a la función Hash, la cual aplica una formula más rápida, prácticamente inmediata, a la posición devuelta en el anterior cálculo. Este proceso se repite hasta encontrar el objeto buscado o, si se esta añadiendo un objeto, hasta encontrar un hueco libre.

3.3. Implementación de la TS

La implementación de la Tabla de Símbolos, su estructura y comportamiento, viene dado por la clase `BHashedTable`. La figura 2 muestra las relaciones de herencia entre las distintas clases implicadas, desde `BHashedTable` a `BDeadObject`.

El núcleo de Tol *inicializa*, *añade*, *busca*, *elimina*, y *obtiene una lista de objetos* de la Tabla de Símbolos utilizando una instancia de la clase `BHashedTable`.

Cada objeto creado por el Evaluador en el ámbito global se añade a la Tabla, usando el método `Add` de `BHashedTable`, y éste queda dentro, referenciado en el atributo `object_` de una instancia de `BObjectAddress`.

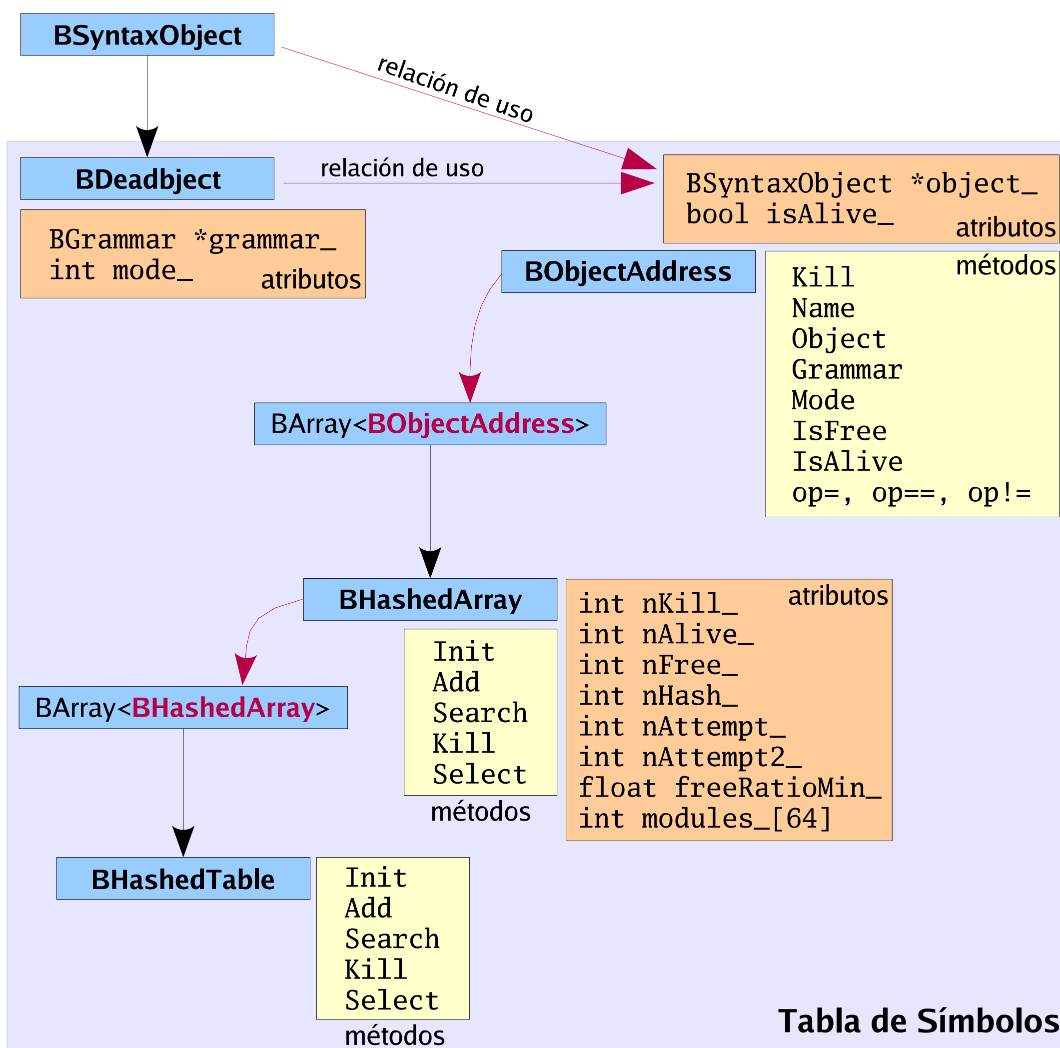
3.3.1. Clases C++

Tol se relaciona con la TS utilizando una instancia de `BHashedTable`. Esta instancia es un array de objetos `BHashedArray` (tablas hash), cada uno de los cuales es a su vez un array de instancias de la clase `BObjectAddress`. Cada objeto de la clase `BObjectAddress` mantiene dos atributos: `object_` e `isAlive_`. El primero se usa para referenciar al objeto insertado en la TS y el segundo para indicar si el objeto almacenado esta *vivo* o fué borrado en una operación anterior y por lo tanto ya no existe.

La figura 3 muestra una imagen conceptual en la que se representan la forma de la TS y las clases relacionadas: `BHashedTable`, `BHashedArray`, `BObjectAddress`, `BDeadObject` y `BSyntaxObject`.

La clase `BHashedTable` ofrece al Evaluador la interfaz necesaria para almacenar, recuperar y borrar los objetos que aparecen en el código fuente Tol en ejecución. La clase `BHashedArray` posee la misma interfaz que `BHashedTable`, como se puede ver en la figura 2. La TS (o lo que es lo mismo, la instancia

Figura 2: Relación de Clases en la Tabla de Símbolos



de BHashedTable) va recorriendo cada una de las instancias de BHashedArray que la forman, ejecutando sobre cada una de ellas las operaciones que se han solicitado sobre ésta, hasta tener éxito.

Como se puede ver en la figura 3, cada instancia de BHashedTable esta formada por varias instancias de BHashedArray. Para cada operación solicitada sobre la TS se toman de manera ordenada, y una a una, las instancia de BHashedArray y se opera sobre cada una de ellas, hasta que la operación tiene éxito.

Por ejemplo, si se solicita almacenar un objeto en la TS, el método Add de BHashedTable recorre cada instancia de BHashedArray intentando añadir éste. Si la operación Add de BHashedArray logra añadir el objeto, la TS devuelve al núcleo la posición que ocupa éste. Si el método Add termina el recorrido de las instancias BHashedArray sin añadir el objeto, se puede deber a que no ha encontrado espacio suficiente o a que el objeto ya existia. Si es el primer caso, la TS creará una nueba instancia de BHashedArray, es decir, una nueba tabla Hash, y añadirá a ésta el objeto. Si por el contrario, el objeto ya existía en alguna de las tablas el método Add de BHashedTable devolverá -1 al núcleo, indicando que no se ha podido añadir el objeto.

A continuación se describe las clases C++ que intervienen en la implementación de la Tabla de Símbolos.

BHashedTable Es la representante del concepto Tabla de Símbolos. Hereda de la implementación del Template BArray<Any>, para la clase BHashedArray, e implementa los métodos de almacenamiento, recuperación, borrado y listado de objetos en la TS.

Implementa también el método Init utilizado para inicializar la TS en el proceso de inicio del Lenguaje.

Método Add Intenta añadir el objeto evaluable (ver sección **??**) que recibe como argumento, devolviendo un entero distinto de -1 si ha tenido éxito y -1 en caso contrario.

Método_Kill Intenta borrar de la TS el objeto evaluable que recibe como argumento, devolviendo la posición que ocupaba éste, si tiene éxito, o -1 en caso contrario. Un objeto BDeadObject con el mismo nombre (atributo name_ heredado de BSyntaxObject), tipo de dato (atributo grammar_) y tipo de objeto (atributo mode_), es creado en sustitución de éste. Este BDeadObject ocupara a partir de entonces el atributo object_ (del objeto ObjectAddress) que ocupaba previamente el objeto retirado. El atributo isAlive_ (también de BObjectAddress) pasará a ser false.

Método Search Intenta devolver el objeto evaluable cuyo nombre, tipo de dato y tipo de objeto coinciden con los argumentos proporcionados. El método Search busca ordenadamente en cada instancia de BHashedArray.

BHashedTable se declara en el fichero tol_btxthash.h y se implementa en txthash.cpp, en el directorio tol/btol/bgrammar.

BHashedArray La TS esta compuesta por instancias de esta clase. Representa la implementación de una Tabla Hash. Ésta clase es una implementación del Template BArray<Any>, donde Any es la clase BObjectAddress. A los atributos propios de BArray<Any> se añaden aquellos que permiten conocer el rendimiento de cada tabla. La clase contabiliza con los siguientes atributos el rendimiento de cada instancia:

int nKill_ Lleva la cuenta del número de objetos BDeadObject contenidos dentro de la tabla. Éstos son referenciados por el atributo object_ de BObjectAddress cada vez que un objeto es eliminado de la TS.

int nAlive_ Lleva la cuenta del número de objetos vivos contenidos dentro de la tabla. Es decir, de aquellos BObjectAddress de la tabla cuyo atributo object_ no es null y apunta a un objeto evaluable distinto de BDeadObject.

int nFree_ Lleva la cuenta del número de posiciones libres en la tabla. Se inicaliza siempre al número de huecos disponibles y se va decrementando cada vez que se añade un objeto. Se utiliza en el método Add para determinar si dado un número de huecos libres no se aceptan más inserciones en esta tabla. De ser así, el método Add de BHashedArray retorna -3 al método Add de BHashedTable, el cual inicia la creación de una nueva instancia de BHashedArray.

int nHash_ Lleva la cuenta del número de veces que se ha invocado una operación de adición o búsqueda de un elemento en la tabla. En cada operación de adición o búsqueda de un elemento se puede invocar mas de una vez a la función `Hash`. (atención: éste atributo, pues, no contabiliza el número de veces que se ha invocado la función `Hash`).

int nAttempt_ Contabiliza el número de veces que se ha invocado la función `Hash`. Ésta es llamada en las operaciones de búsqueda (para acceder a un objeto o eliminarlo de la TS) y adición. Se utiliza para calcular información estadística sobre el rendimiento de la tabla, concretamente para el cálculo de la media del número de veces que ha sido necesario invocar a la función `Hash` en más de una ocasión.

int nAttempt2_ Al igual que el anterior, se utiliza para calcular información estadística sobre el rendimiento de la tabla, concretamente para el cálculo de la desviación típica del número de veces que ha sido necesario invocar a la función `Hash` en más de una ocasión.

double freeRatioMin_ Indica el porcentaje mínimo de huecos libres a partir del cual será necesario crear una nueva tabla. Se utiliza para calcular el tamaño del `BArray<BObjectAddress>`, en el método `Init` de `BHashedArray`.

int modules_[64] Inicializado en función del tamaño de la tabla, y utilizado en la función `Hash` para calcular la posición de un objeto a partir de su nombre. Si el nombre del objeto tiene más de 64 caracteres, los restantes no se utilizan para calcular su posición. Lo que implica que identificadores iguales en los primeros 64 caracteres generan la misma posición en la tabla.

El comportamiento de `BHashedArray` cuenta con métodos para inicializar la tabla, añadir objetos, buscarlos y eliminarlos de la tabla. Los métodos que representan estas tres últimas operaciones están sobrecargados para recibir unos argumentos de la TS y operar posteriormente con objetos `BObjectAddress`.

El método más importante de ésta clase es la función Hash que retorna la posición de un objeto en la tabla a partir de su nombre (ver sección **??**). A continuación se describen ligeramente éstos métodos:

Método Hash El método `Hash` retorna la posición que debe ocupar un objeto.
Declara tres parámetros, el primero es el nombre del objeto al que se va a aplicar la función, el último es la iteración que representa la llamada, y el segundo es la posición en la tabla que retorna la función. El método esta declarado con un valor de retorno `void`, pero su valor de retorno se aplica al segundo argumento.
Si el argumento iteración es igual a 0, el método calcula el valor del segundo argumento basándose en el argumento nombre. Si el valor del argumento iteración es mayor que 0, el método calcula el valor del segundo argumento a partir del valor anterior de éste.

Método público Search Recibe el valor de sus tres argumentos, tipo de dato, nombre, y tipo de objeto, del método Search de `BHasedTable`, el cual es invocado directamente desde el analizador semántico para buscar el identificador encontrado en el árbol sintáctico. Retorna el objeto evaluable, `BSyntaxObject`, cuyos atributos satisfacen la búsqueda.[4]
El método `Search` lleva a cabo su trabajo creando un objeto `BDeadObject` a partir de los argumentos dados, insertando el `BDeadObject` en un nuevo objeto `BObjectAddress` e invocando posteriormente al método privado Search.

Método privado Search Recibe un `BObjectAddress` cuyo atributo `object_` apunta a un objeto `BDeadObject`.
Calcula la posición que debe ocupar el objeto con el nombre dado por el `BObjectAddress` y si ésta está ocupada comprueba si el objeto que ocupa la posición es el objeto con los atributos buscados. Si el objeto que ocupa la posición devuelta por la función `Hash` no coincide con el objeto buscado, se vuelve a aplicar la función `Hash`, hasta que se encuentre el objeto, o bien hasta que falle la búsqueda.
La búsqueda falla cuando la posición que devuelve la función `Hash` ya ha sido examinada. El método privado `Search` retorna la posición del objeto buscado, o -1 si no ha tenido éxito.

[4]Ver los detalles de la clase BObjectAddress y su método operator== para entender los criterios por los cuales se establece que se ha encontrado un objeto en la tabla.

Método público Add Recibe un objeto BSyntaxObject, y devuelve la posición que ocupa después de haber sido añadido con éxito. Si no hay éxito en la operación retorna -1. Este método es invocado exclusivamente por el método Add de BHashedTable. Su funcionalidad consiste únicamente en encapsular el BSyntaxObject pasado como parámetro en un objeto BObjectAddress y pasar éste como parámetro al invocar al método privado Add.

Método privado Add Recibe un argumento BObjectAddress y devuelve la posición que ocupa el objeto en la tabla si se ha añadido con éxito o -1 en caso contrario. Calcula la posición que debe ocupar en la tabla a través de la función Hash y verifica si esta libre. De ser así, inserta el elemento y retorna su posición. Si la posición estuviera ocupada por otro objeto, estos se comparan con el método operator== de BObjectAddress. Si el objeto fuera considerado como un duplicado, Add devolverá -1. Si por el contrario el objeto en la posición previamente calcula fuera distinto al que se intenta almacenar, se volverá a aplicar la función Hash para calcular una posición alternativa.

Método público Kill Al igual que el método público Add, recibe un BSyntaxObject directamente desde el método Kill de BHashedTable. Lo encapsula en un BObjectAddress y se lo pasa al método Kill privado.

Método privado Kill Recibe un BObjectAddress del método público. Utiliza el método Search privado para buscar la posición que ocupa el objeto que se quiere eliminar. Si Search devuelve un valor mayor que -1 se elimina el objeto que ocupa esa posición en la tabla, utilizando para ello el método Kill de BObjectAddress.

BObjectAddress Cada hueco de cada tabla Hash contenida en la TS esta ocupado por un objeto de la clase BObjectAddress. Cada instancia de esta clase representa una celda de cada tabla Hash, y cada instancia de BObjectAddress tiene dos atributos:

BSyntaxObject *object_ Utilizado para referenciar al objeto evaluable que ocupa la posición de ésta posición en la tabla Hash. Puede tomar tres tipos de valores distintos.

- Puede ser Nulo (null)
- Puede apuntar a un objeto BDeadObject, es decir, un objeto que existió pero que ha sido borrado.
- Puede apuntar a un objeto BSyntaxObject de alguno de los tipos de datos de Tol.

bool isAlive_ Si es verdadero, el atributo object_ apunta a un objeto vivo, es decir, a un objeto evaluable de alguno de los tipos de datos de Tol. Por el contrario, si su valor es falso, el atributo object_ apunta a un objeto que ya no existe, representado por un BDeadObject.

La clase BObjectAddress tiene dos grupos de funciones. Por un lado el grupo de las funciones que permiten extraer información de los objetos referenciados por el atributo object_, y por otro lado aquellas que modifican el estado de cada celda de la tabla hash, o verifican su contenido.

En el primero grupo están las funciones Name(), Grammar(), Mode() y Object(), que retornan respecticamente el nombre del objeto (ya sea este un BSyntaxObject de algún tipo de dato, o un BDeadObject), el Tipo de Dato del objeto, el Tipo de Objeto (BOBJECTMODE, BOPERATORMODE...) y el objeto en sí referenciado por el atributo object_.

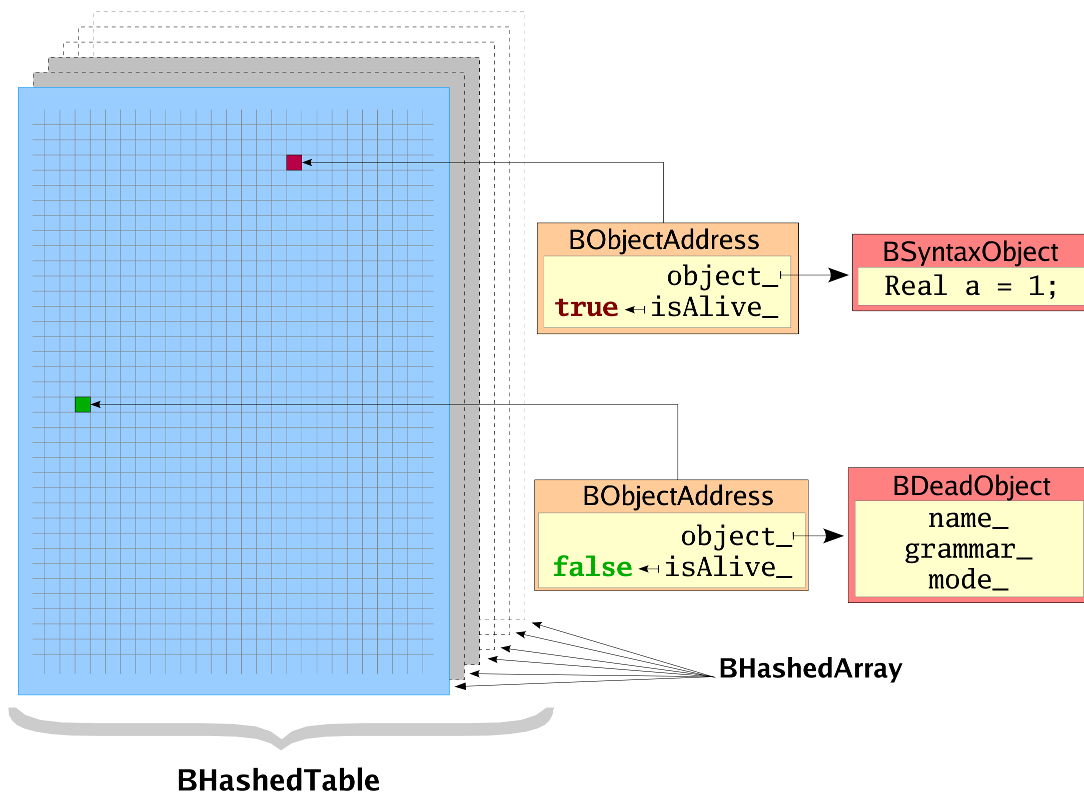
En el segundo grupo se encuentran los siguientes métodos:

Método Kill Elimina el objeto referenciado por el atributo object_ de la tabla Hash, y sustituye éste por un nuevo objeto BDeadObject que guardara información sobre el objeto eliminiado.[5] El atributo isAlive_ se establece false indicando que el objeto cuyos datos recuerda el BDeadObject representan a un objeto que ya no está almacenado en la tabla.

Método IsFree Retorna un valor buleano, cierto si el atributo object_ es nulo y falso en otro caso.

[5] La creación del objeto BDeadObject es necesaria para no dejar la posición vacía. De dejar esta posición vacía, podría perderse las referencias a otros objetos añadidos en la tabla que hubieran colisionado con éste que se esta eliminando (ver subsección **??**).

Figura 3: Direcciones vivas y direcciones muertas en la TS



Método `IsAlive` Retorna un valor buleano, cierto si el atributo `isAlive_` es `true` y el atributo object no es nulo.

Método `operator=` Introduce un objeto en esta celda de la tabla Hash, actualizando el atributo `object_` con el nuevo objeto. El objeto entrante es un objeto vivo reciéntemente creado por el evaluador de Tol. Puede ser una variable o una función de cualquier tipo de dato. El atributo `isAlive_` pasa ser `true`, pues `object_` referencia a un objeto vivo.

Método `operator==` Determina si un objeto `BObjectAddress` es igual al objeto `BObjectAddress` pasado como argumento. Para que un objeto almacenado en una celda de la tabla Hash se considere igual a otro, tienen que tener el mismo nombre, el mismo tipo de dato, y ser el mismo tipo de objeto. En la práctica el manejo de distintos tipos de objetos en la misma TS y la posibilidad de sobrecargar los nombres de los objetos de Tol, introducen un cierto nivel de dificultad en éste método.

BDeadObject Ésta clase se utiliza solo en la operación de eliminación de un objeto evaluable de la tabla de símbolos. Cada vez que un objeto evaluable es eliminado de alguna tabla hash de la tabla de símbolos, se crea una instancia de ésta clase que ocupará el atributo `object_` de la celda donde antes estaba el objeto eliminado. Ver subsección **??**.

La clase `BDeadObject` hereda de `BSyntaxObject` con el propósito de aplicar las capacidades polimórficas del atributo `BSyntaxObject *object_` de la clase `BObjectAddress`.

Cada objeto `BDeadObject` guarda en sus atributos datos relevantes del objeto que ocupaba previamente la celda. Define dos atributos adicionales a los heredados de `BSyntaxObject`, de éstos últimos solo utiliza uno, el nombre. Los dos atributos que añade son:

BGrammar *grammar_ Referencia al objeto BGrammar, representante del Tipo de Dato del objeto que previamente ocupaba la celda donde ahora está éste objeto.

int mode_ Contiene el entero correspondiente al Tipo de Objeto que previamente ocupaba la celda donde ahora está éste objeto. Puede tomar los siguientes valores: BOBJECTMODE, BOPERATORMODE y BUSERFUNMODE.

3.4. Objetos contenidos en la TS

Tol almacena en la Tabla de Símbolos todos los objetos evaluables creados en el ámbito global. Como se explicó en la sección **??**, existe una gran variedad de objetos evaluables.

Una posible clasificación éstos tiene en cuenta la naturaleza de los objetos, formando tres grupos diferenciados: Variables, Funciones Built-in y Funciones de Usuario. Otra posible clasificación consiste en diferenciarlos por el Tipo de Datos al que pertenecen.

El nombre de los objetos, junto con su Tipo de Dato y su Tipo de Objeto (o modo) forman, como se ha explicado al comienzo de éste capítulo, los tres datos clave de los objetos almacenados en las celdas de la TS.

Así pues, un objeto en la Tabla de Símbolos se diferencia de otro por el contenido de estos tres campos que se combinan entre sí:

Tipo de Dato	Tipo de Objeto	Nombre
Anything Code Complex Date Matrix Polyn Ratio Real Serie Set Text TimeSet	BOBJECTMODE BOPERATORMODE BUSERFUNMODE	<cualquiera>

No es posible encontrar dos objetos con el mismo **Tipo de Dato**, el mismo **Tipo de Objeto** y el mismo **Nombre**. Pero existe una excepción: objetos que con el mismo nombre tienen el Tipo de Dato "Code", y el Tipo de Objeto "BUSERFUNMODE". En ésta situación especial, el objeto referenciado en la celda de la tabla Hash contiene los datos que van a diferenciar a un objeto de otro. Es decir, es necesario realizar introspección sobre el objeto cuando éste pertenezca al Tipo de Dato Code y su Tipo de Objeto sea BUSERFUNMODE.

Esta situación especial viene dada por el hecho de que las Funciones de Usuario son objetos evaluables de tipo Code, y que Tol permite la Sobrecarga de nombres de Funciones de Usuario, siempre que éstas devuelvan distintos Tipos de Datos. Es decir, es posible crear una función llamada "f" que retorne un tipo Real, y otra con el mismo nombre "f" que retorne un tipo Text:

```
Real f (...declaración de parámetros...)
  { ... sentencias de la función ... };
Text f (...declaración de parámetros...)
  { ... sentencias de la función ... };
```

La Tabla de Símbolos almacenará estas dos funciones, y las dos serán dos objetos evaluables con el Tipo de Objeto BUSERFUNMODE, y el Tipo de Dato Code.

La TS implementa código adicional en los métodos operator== y Kill, de la clase BObjectAddress, para poder manejar esta situación excepcional.[6]

[6]Es posible mejorar la implementación actual de la TS. En la práctica existe un problema debido al elevado número de colisiones

3.5. Inicialización de la TS

La Tabla de Símbolos de Tol esta representada por el atributo `static BHashedTable *hashVar_` de la clase `BGrammar`. Éste atributo es inicializado en la carga de Tol, desde la función `InitGrammars()` de language.cpp, donde se invoca al método
`static BGrammar::InitHashedArray()`.

El método `InitHashedArray()` crea la instancia de la clase `BHashedTable` y la inicializa invocando explicitamente al método `Init` con los parámetros 1000 y 0.25:

```
HashVar().Init(10000,0.25);
```

El método `Init` de `BHashedTable` crea espacio para 100 tablas Hash (tipo `BHashedArray`) pero inicialmente solo se utiliza una. Cuando una operación de inserción falla por falta de espacio para el nuevo elemento, se crea una nueva tabla Hash, desde el método `Add` de `BHashedTable`.

Trás la inicialización de la Tabla de Símbolos, se inicia la creación de las instancias de cada Tipo de Datos, es decir, la creación de las Variables y Funciones Built-in propias de cada Tipo de Datos. Éste proceso comienza, también, en la función InitGrammars() de language.cpp.

3.5.1. Inserción de objetos propios de Tol

Una vez creada la Tabla de Símbolos, y antes de que el Lenguaje termine su inicialización, permitiendo al usuario interactuar con el intérprete, cada Tipo de Datos crea sus instancias, es decir, las variables que ofrece al programador.

Las variables que ofrece Tol por defecto se crean a través del método `InitInstances`, heredado en cada Tipo de Dato bien del Template `BGraContens<Any>` o bien del Template `BGraObject<Any>`.

Por otro lado, las funciones Built-in se añaden como objetos de tipo `BOPERATORMODE` al arrancar Tol. El proceso se desencadena a través del uso de la función `__delay_init`, descrita en la sección **??**.

Durante el arranque de Tol la función `__delay_init` crea las funciones Built-In como objetos `BOperator` (`BInternalOperator` o `BExternalOperator`), lo que hace que su atributo “Tipo de Objeto” sea `BOPERATORMODE`. En el proceso de construcción de las funciones Built-in como objetos `BOperator`, éstas entran en la Tabla de Símbolos (ver método constructor de la clase `BOperator`).

3.6. Funciones para el manejo de la TS

3.6.1. Búsqueda de un objeto

El proceso de búsqueda de un objeto en la Tabla Hash depende de los tres atributos mencionados en los apartados anteriores: Nombre, Tipo de Dato y Tipo de Objeto. Se inicia siempre como resultado del análisis semántico, considerándose parte de la actividad del Evaluador de Expresiones (ver capítulo **??**).

La función que inicia el proceso de búsqueda de un objeto evaluable dentro de la Tabla de Símbolos es el método `Search` perteneciente a la clase `BHashedTable`.

El algortimo desencade las siguientes acciones dentro de la TS:

1. Se Inicia la búsqueda del objeto que posee el Nombre, Tipo de Dato y Tipo de Objeto dados. La búsqueda se lleva a cabo secuencialmente en cada tabla Hash (instancias de `BHashedArray`) que forman la TS.
   La búsqueda en cada tabla Hash puede devolver un objeto evaluable, es decir un `BSyntaxObject`.
   Si la búsqueda tiene éxito, el proceso se detiene y se devuelve el objeto encontrado en la tabla BHash que se estaba procesando. Si no se ha encontrado el objeto se devuelve un puntero nulo al Evaluador.

---

producidas por la aplicación de la función Hash exclusivamente al nombre. La capacidad de sobrecarga de los nombres de los objetos en Tol hace necesario un cálculo adicional en la función Hash que evite las colisiones.

También es posible mejorar la implementación de las Funciones de Usuario, de manera que no sean objetos de tipo Code, sino objetos del Tipo de Datos devuelto por la función, pero con Tipo de Objeto BUSERFUNMODE. En la práctica ya tienen éste Tipo de Objeto, pero un problema en el diseño de los Tipos de Datos hace que éste dato no esté siempre disponible, lo que obliga a una extraña introspección en `BObjectAddress::operator==`.

2. La búsqueda en cada tabla Hash consiste en aplicar la función Hash al nombre del objeto buscado y comprabar la posición de retorno. Si la posición devuelta por la función Hash esta ya ocupada se compara el objeto referenciado en la celda con los argumentos dados en la búsqueda, es decir, el Tipo de Dato y el Tipo de Objeto. Este trabajo de comparación lo realiza el método `operator==` de `BObjectAddress`.

3. El método `BObjectAddress::operator==` compara los argumentos con los que se invocó en (1) al método Search, ahora encapsulados en un objeto `BObjectAddress`, con el objeto encontrado en la posición de la tabla devuelta por la función Hash.
   Si los dos objetos coinciden en nombre, tipo de dato y tipo de objeto, entonces los dos son iguales. Salvo que el tipo de dato sea Code y el tipo de objeto sea BUSERFUNMODE, en cuyo caso es necesario inspeccionar el objeto almacenado en la tabla de símbolos.

La figura 4 ilustra la actividad que tiene lugar en el Evaluador de Expresiones y la posterior llamada al método Search de la TS.

Caso especial

El interprete de Tol puede encontrar referencias a funciones de usuario en el código que esta ejecutando. En algunas ocasiones es posible deducir del contexto el Tipo de Dato que la función de usuario debe devolver. En otras situaciones es imposible obtener esa información del contexto.

Si el Evaluador es capaz de obtener el Tipo de Dato de retorno de una función de usuario, entrega esa información a la función de búsqueda de la Tabla de Simbolos, indicando además que el tipo de objeto buscado es un `BUSERFUNMODE`.

El método de comparación de dos `BObjectAddress`, es decir, `operator==`, tiene en cuenta esta situación especial para realizar una inspección del objeto almacenado en la tabla Hash.

3.6.2. Adición de un objeto

Los objetos evaluables entran en la TS utilizando el método `static BGrammar::AddObject`.

El comportamiento de la TS en la adición es el mismo en el caso de las Variables y las Función Built-in, sin embargo es ligeramente distinto al añadir Funciones de Usuario.

El método de inserción es el mismo, tanto en la TS (representado por `BHashedTable::Add`) como en las Tablas Hash (representado por `BHashedArray::Add`). Pero el método de comparación del objeto que ocupa una celda dada y el objeto que representa la nueva Función de Usuario, requiere código adicional que haga una inspección extra en el objeto de la celda.

A continuación se describe la inserción de Variables y Funciones de Usuario. Considérese la inserción de Funciones Built-in igual a la inserción de Variables.

La figura 5muestra una ilustración de la inserción de Variables y Funciones de Usuario, con el mismo nombre, en la Tabla de Símbolos.

Adición de variables

En el caso de las variables, se almacenan siempre con el Tipo de Objeto `BOBJECTMODE`. Variando el Nombre y el Tipo de Dato. Por ejemplo:

```
Real a = 1;
Text a = "Hola";
```

Son dos objetos evaluables, con Tipo de Objeto `BOBJECTMODE`, con el mismo Nombre, y con distinto Tipo de Dato:

Sentencia	Nombre	Tipo de Dato	Tipo de Objeto
`Real a = 1`	`a`	`Real`	`BOBJECTMODE`
`Text a = "Hola"`	`a`	`Text`	`BOBJECTMODE`

Figura 4: Búsqueda en la Tabla de Símbolos

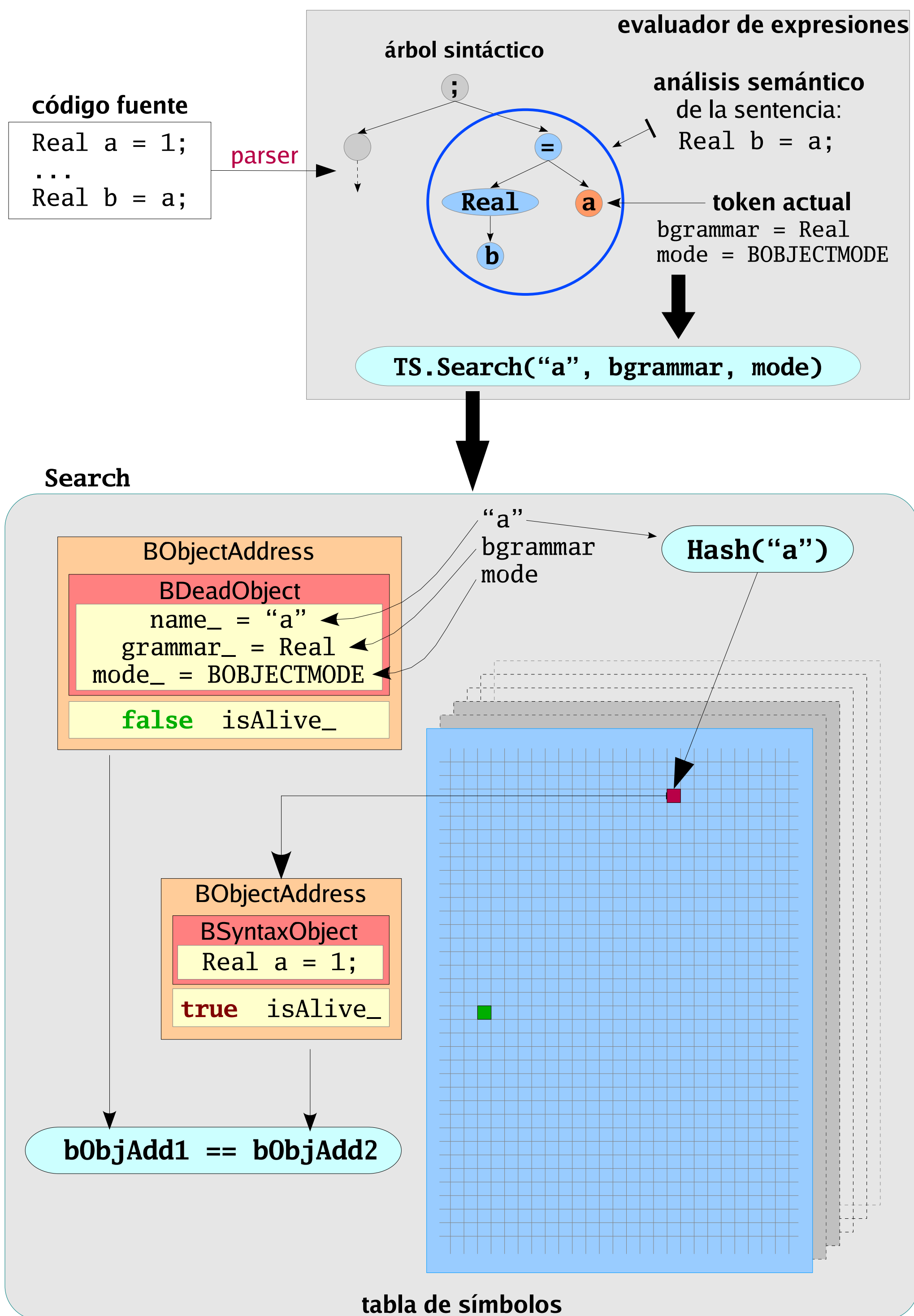
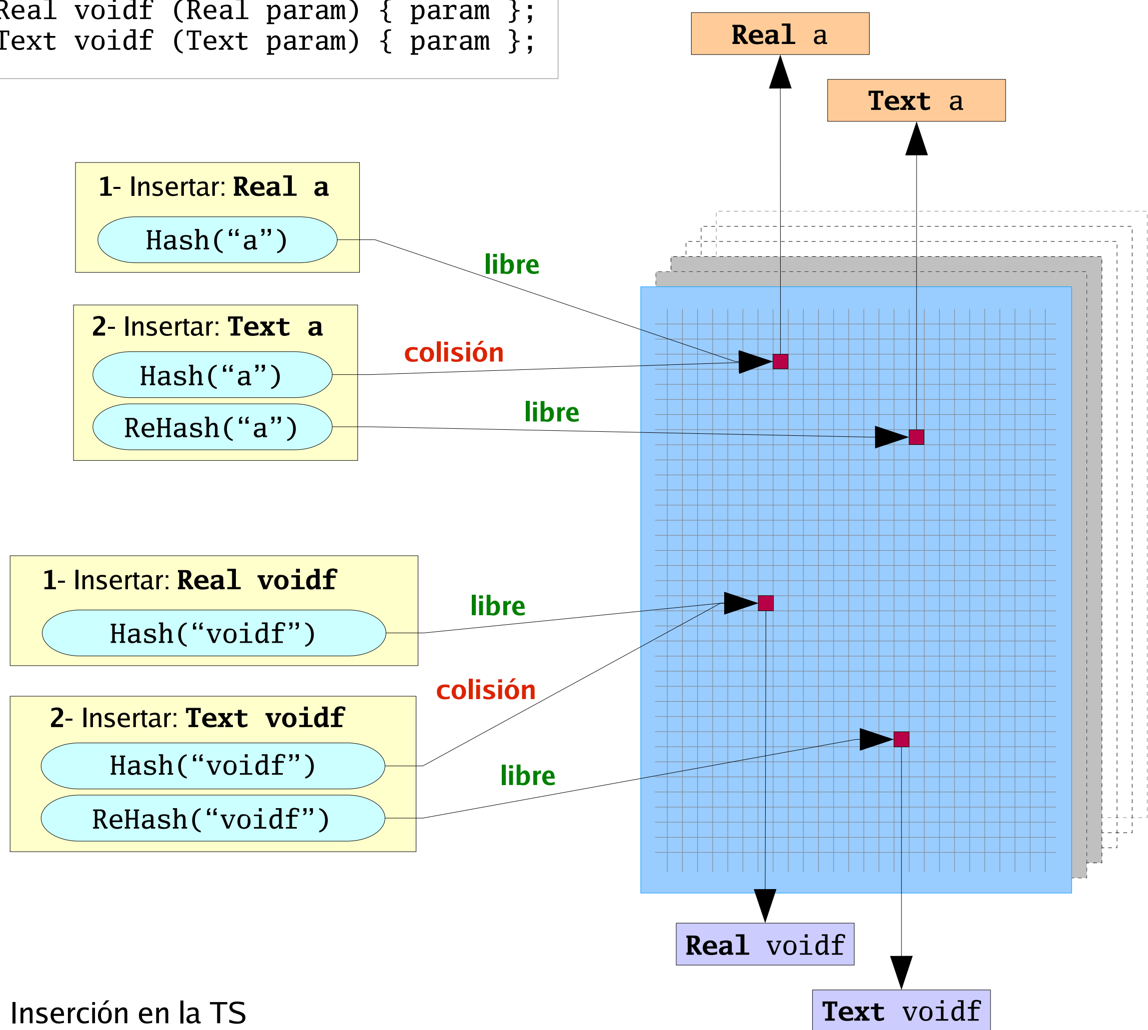


Figura 5: Inserción de Variables y Funciones de Usuario, con el mismo nombre, en la TS



El proceso de inserción de los dos objetos anteriores en la TS, implica el siguiente proceso[7]:

1. Se encuentra “Real a = 1” en el árbol sintáctico ->se busca “Real a” en la TS y no se encuentra ->se procede a añadir a la TS.
2. Aplicación de la función Hash al nombre de variable: “a”. Se obtiene una posición vacía en la Tabla Hash en uso. Se inserta el BSyntaxObject “Real a = 1” en la celda (BObjectAddress).
3. Se encuentra ’Text a = “Hola”’ en el árbol sintáctico ->se busca “Text a” en la TS y no se encuentra ->se procede a añadir a la TS.
4. Aplicación de la función Hash al nombre de variable: “a”. La función Hash devuelve una posición ya ocupada por la variable “Real a”. La TS debe verificar si el objeto almacenado en la celda es el mismo que el que se intenta insertar.
5. El método operator== compara los nombres, tipos de datos y tipos de objetos.
   El nombre y el tipo de objeto de ambas variables coincide, tal y como muestra la tabla anterior, pero el Tipo de Dato es distinto, por lo que se consideran objetos distintos.
6. Se vuelve a aplicar la función Hash para calcular una nueva posición a partir de la anterior. La nueva posición esta vacía, por lo que se inserta el objeto “Text a” en dicha posición.

Adición de funciones de usuario

La interpretación del siguiente código implica la creación de dos objetos evaluables, Funciones de Usuario, y su inserción en la Tabla de Símbolos:

```
Real func (Real param) { param };
Text func (Text param) { param};
```

Las dos funciones son objetos evaluables de tipo BUSERFUNMODE, con el mismo nombre, pero con distinto Tipo de Datos de retorno. Sin embargo, el Tipo de Dato asociado al objeto en la celda es Code, por lo que la comparación en el caso de colisión con otra Función de Usuario con el mismo nombre es distinta a la que ocurre en el caso de Variables y Funciones Built-in.

Sentencia	Nombre	Tipo de Dato	Tipo de Objeto
`Real func (...) { ... }`	`func`	`Code`	`BUSERFUNMODE`
`Text func (...) { ... }`	`func`	`Code`	`BUSERFUNMODE`

Como se ve en la tabla, los tres datos Nombre, Tipo de Dato y Tipo de Objeto de las dos Funciones son iguales. ¿Cómo se distinguen entre sí? -Veámos que sucede en la TS al insertar estos dos objetos[8]:

1. Se encuentra “`Real func (...) { ... }`” en el árbol sintáctico ->se busca una función que se llame “func” y retorne un Real y no se encuentra ->se procede a añadir a la TS.
2. Aplicación de la función Hash al nombre de la primera función. Se obtiene una posición vacía en la Tabla Hash en uso. Se inserta el `BSyntaxObject` que representa la función (un objeto `BUserFunCode`) con nombre “func”.
3. Se encuentra “`Text func (...) { ... }`” en el árbol sintáctico ->se busca una función que se llame “func” y retorne un Text y no se encuentra ->se procede a añadir a la TS.
4. Aplicación de la función Hash al nombre de la segunda función. La función Hash devuelve la posición ocupada por la función de usuario anterior. La TS debe verificar si los dos objetos evaluables son iguales.

[7] Suponiendo que no existe otro objeto evaluable con el nombre “a”.
[8] Suponiendo que no existe ningún objeto evaluable llamado “func” en la Tabla de Símbolos.

5. El método `operator==` compara los nombres, y verifica el Tipo de Objeto. Al ser éste `BUSERFUNMODE` es necesario realizar una inspección en el objeto con el que se ha colisionado en la celda.
   Es necesario verificar si la Función de Usuario referenciada en la celda esta viva para analizar el Tipo de Dato de retorno. En él caso particular del ejemplo, la función está viva, y se determina que no coincide con la que se quiere añadir (el paso 3 lo habría detectado).

6. Se vuelve a aplicar la función Hash para calcular una nueva posición a partir de la anterior. La nueva posición está vacía, por lo que se inserta el `BSyntaxObject` que representa la función (otro objeto `BUserFunCode`) con nombre fun.

3.6.3. Borrado de un objeto

De las tres operaciones que permite una Tabla Hash, Inserción, Búsqueda y Borrado, ésta última es la que más penaliza el funcionamiento de la estructura de datos.

Para entender el problema del borrado de elementos de una Tabla Hash, analicemos la situación fijándonos en las operaciones de inserción de los 4 objetos de la subsección anterior:

```
Real a = 1;
Text a = "Hola";
Real voidf (Real param) { param };
Text voidf (Text param) { param };
```

Trás ejecutar esas lineas de código la TS contendrá los 4 objetos.

El acceso a la variable de tipo `Real` y a la función `voidf` que retorna un `Real` no produce ninguna colisión, pues fueron las primeras que se insertaron con esos nombres. En cambio el acceso a la variable `Text` y a la función que retorna un `Text` produce una colisión con los correspondientes objetos del tipo `Real`, que se subsanan aplicando de nuevo la función Hash.

Si una vez insertados los cuatro objetos en la TS borrámos la variable Real a y la función Real voidf, quedarán dos huecos vacíos. Estos huecos no deben quedarse como huecos libres, es decir, si al borrar los dos objetos se marcan las posiciones como vacías, los objetos Text a y Text voidf no se encontrarán en los siguientes accesos, pues los algorítmos de la TS determinarán, al encontrar los huecos libres, que no hay más objetos con los nombres buscados.

Cuando se accede a la TS para buscar un elemento o para añadir uno nuevo se verifica en que estado se encuentra la posición devuelta por la función Hash. Si la posición esta vacía y la operación es de inserción, se inserta en dicha celda. Si la operación es de búsqueda, se determina entonces que no existe el elemento buscado.

Posiciones vacías con BDeadObject

Tol utiliza objetos de la clase `BDeadObject` para ocupar las posiciones de los objetos borrados.

Cuando se borra la variable Real a, la TS inserta en la celda que ocupaba dicha variable un objeto de la clase `BDeadObject`, con el mismo nombre, el mismo Tipo de Datos, y el mismo Tipo de Objeto que el que se va a borrar.

Una vez finalizada la operación, una búsqueda del objeto borrado encontrará un `BDeadObject`, que indica a la TS que el objeto buscado ya no existe en la tabla.

Si el objeto buscado es uno que previamente colisionó con el borrado, como es el caso de `Text a`, la TS encontrará primero el `BDeadObject` que representa al objeto borrado y continuará buscando. Ahora el objeto borrado no deja un hueco vacío, por lo que la TS sabe que debe continuar buscando.

4. El Evaluador de Tol

El conjunto de pasos que lleva a cabo el lenguaje desde el momento en el que recibe el árbol sintáctico del Parser y hasta que las variables y funciones son creadas es conocido como el Evaluador. El Evaluador de Tol es la parte del núcleo encargada de la interpretación semántica de las sentencias escritas en Tol.

Realiza el Análisis Semántico de las ramas del árbol, y delega en los módulos del Entorno de Ejecución para crear y buscar variables, operadores, y funciones.

El Evaluador es la columna vertebral a partir de la cual se inician los pasos que llevan a la ejecución de las sentencias del árbol sintáctico, se considera la "receta" de pasos que el intérprete aplica para ejecutar cada línea de código escrita por el programador de Tol.

4.1. Componentes

El Evaluador no está localizado en exclusiva en un método concreto, sino esparcido entre varios métodos de distintas clases. El más importante está localizado en la clase que representa el Núcleo de Tol: `BGrammar`. El resto forman parte de otras clases, y son invocados desde el principal dependiendo del objeto sintáctico que se vaya a evaluar.

El método raiz del Evaluador lleva a cabo el Análisis Semántico de las ramas del Árbol Sintáctico. Éste delega en otros métodos implementados en la jerarquía de clases `BOperator`, y en las instancias de la clase `BSpecialFunction`, que implementan entre otros, el operador de reasignación "`:=`" y el operador de campo de un conjunto "`->`".

Los métodos del Evaluador relacionados con BOperator son el `BEqualOperator::Evaluate`, que se encarga de las expresiones de asignación, `BStandardOperator::Evaluate`, encargado de la evaluación de funciones Built-in y el resto de operadores unarios y binarios, y `BUserFunction::Evaluator`, que evalua las llamadas a funciones de usuario. Todos ellos implementados en el fichero oprimp.cpp del directorio tol/btol/bgrammar.

4.2. Análisis Semántico

Cada sentencia que se evalua tiene un tipo de dato asociado como resultado. El proceso de análisis semántico se encarga de verificar la concordancia entre los tipos de datos de las variables, funciones y operadores que intervienen en cada expresión. Las comprobaciones se llevan a cabo a lo largo de todos los métodos de evaluación que componen el Evaluador.

El proceso se inicia mediante la lectura de tokens del árbol sintáctico. El Evaluador le pide al Parser el siguiente token del árbol sintáctico mediante la función `static treToken(List*)`, que retorna el primer token en el árbol pasado como parámetro. En función del tipo de token leído el Evaluador se enfrenta a distintos tipos de expresiones, que serán evaluadas de distinta forma.

El método `Evaluate(BList*)` de la clase `BGrammar,` implementado en graimp.cpp, en el directorio tol/btol/bgrammar, es el principal método en el proceso de evaluación.

Éste método lee el tipo de token del árbol sintáctico que recibe como parámetro, comprueba el tipo de Token, y en función de él continua con el proceso de evaluación. Es un método recursivo, utilizado tanto con expresiones completas (las separadas entre sí por tokens SEPARATOR: ";" o ",") como con subexpresiones. Así, en la sentencia de asignación vista hace varios apartados, de creación de la variable `Real a` con valor "1", el método `Evaluate` interviene en dos ocasiones.

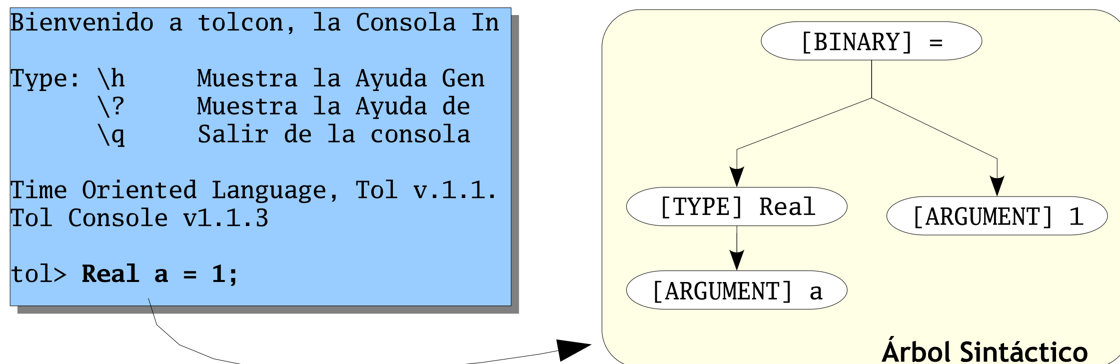
A continuación se describe una traza de evaluación de dicha sentencia:

Paso 1 Trás obtener el Árbol Sintáctico, se invoca al Evaluador: `BGrammar::Evaluate(BList*)` que lee el token de la raiz, **`[BINARY]=`**
Busca el operador (que coincide con `BEqualOperator`) y ejecuta su método `Evaluate`

Paso 2 `BEqualOperator::Evaluate` comprueba que la parte izquiera de la expresión, es decir, la rama izquierda del árbol sintáctico (la que corresponde con el tipo y nombre de la variable) está correctamente formada, e invoca a la función `CreateObject` (representante del Entorno de Ejecución, encargada de la creación de variables), indicando el tipo de dato de la expresión y el nombre que debe tener la variable que se va a crear.

Paso 3 `CreateObject` verifica que no exista una variable con el mismo nombre y tipo de dato. Si ya existe emitirá un mensaje de error y devolverá el control a `BGrammar::Evaluate`, que continuará su ejecución en la siguiente sentencia. Si no existe, invoca a `BGrammar::Evaluate`

Figura 6: Sentencia y árbol Sintáctico



con la rama derecha del árbol. Es la 2ª llamada en la evaluación de la sentencia "`Real a = 1;`"

Paso 4 `BGrammar::Evaluate(BList*)` lee el token de la raiz, **`[ARGUMENT] 1`**
Invoca al método `BGrammar::FindOperand("1")` (del Entorno de Ejecución) que buscará una variable con el nombre "1" en el ámbito de ejecución global (en el caso actual). Si no la encuentra, invocará al método `FindConstant` del tipo de datos `Real`, responsable de la creación de variables de tipo `Real` a partir de un literal.
Así pues, se ejecutará el método `BGraContens<BDat>:FindConstant("1")` y retornará un objeto `BGraContens<BDat>` con un atributo `contens_` de tipo `BDat` y valor `1`.

Paso 5 El retorno del `BGraContens<BDat>` con valor `1` llega a `BGrammar::Evaluate`, que lo retorna a `CreateObject`. Éste crea un nuevo objeto `BRenContens<BDat>`, que encapsula el anterior `BGraContens<BDat>` con el valor `1`. A continuación le pone el nombre y lo añade, bien a la Tabla Hash, si está en ámbito global, o bien a la Pila Local, si está en ámbito local.

En esta secuencia de pasos el Evaluador ha intervenido dos veces. La primera en el paso 1 y la segunda en el paso 4. El resto de acciones se inician desde el Evaluador, y deben devolver a éste un objeto evaluable, es decir, un `BSyntaxObject`, como resultado de la interpretación de la sentencia leída del árbol sintáctico. En el caso de la sentencia analizada, el resultado es un `BSyntaxObject`, cuyo tipo de dato es `Real` y cuyo valor es `1`.

4.3. Evaluación de funciones de usuario

Las funciones de usuario son internamente objetos de tipo Code. La evaluación de éstas se divide en dos pasos:

- Evaluación de la declaración/definición de la función
- Evaluación de la llamada a la función

La evaluación de cualquier declaración/definición de una función de usuario se realiza utilizando el objeto `BCodeCreator` (otoken "`#F#`" en el árbol sintáctico) que existe por cada tipo de dato. Éste `BCodeCreator` se instancia en la función `InitCommonInstance(BGrammar *gra)` del fichero language.cpp. Es decir, la creación del operador `#F#` es parte del proceso de creación de cada tipo de dato de Tol.

El operador `#F#` no es visible en el código Tol empleado por el usuario, debido a que el Filtro (un submódulo del Parser) realiza varias sustituciones previas al Análisis Sintáctico, entre las que se encuentra la que afecta a éste operador. El siguiente código Tol:

```
Real quad(Real param) { param * param };
```

Es traducido por el Filtro en éste otro:

```
Real quad(Real param) #F# { param * param };
```

Las dos sentencias anteriores son equivalentes, aunque si escribimos directamente la segunda obtendremos un error sintáctico, es decir, Tol solo admite la primera.

4.3.1. Evaluación de la declaración y definición

Partiendo del siguiente código fuente:

```
Real quad (Real param) { param * param };
```

El parser genera un árbol sintáctico como éste:

```
----> [BINARY]#F#
 |----> [TYPE]Real
 | |----> [FUNCTION]quad
 |   |----> [TYPE](Real)
 |     |----> [ARGUMENT]param
 |----> [BINARY]{*}
   |----> [ARGUMENT]param
   |----> [ARGUMENT]param
```

El evaluador determina que se trata de la declaración y definición de una función leyendo el token binario #F#. La evaluación se realiza en los siguientes pasos:

1. El evaluador lee el token binario `#F#` y busca el operador asociado.
2. El operador `#F#` intenta crear un objeto `BUserFunction` que contendrá la declaración de la función (nombre y lista de parámetros) y la definición (cuerpo).
    a) Se analiza la rama izquierda del árbol para obtener el nombre de la función y el tipo de dato que devuelve. Se instancia un objeto `BUserFunction` con estos datos.
    b) Se establecen los atributos `declaration_` y `definition_` del objeto `BUserFunction`, copiando en ellos la rama izquierda y derecha del árbol respectivamente.
3. Si el objeto `BUserFunction` se ha creado correctamente:
    a) Se añade el nombre del fichero fuente en el que se ha creado la función de usuario
    b) Se crea un objeto `BGraContens<BCode>` con el `BUserFunction` dentro, y se le pone el nombre de la función.
    c) Se solitica al núcleo que almacene la nueva instancia `BGraContens<BCode>` que se usará para localizar la nueva función creada cuando sea invocada.

El objeto evaluable creado es, según 3.c) del tipo de datos `Code`.

4.3.2. Evaluación de la llamada

La evaluación de una llamada a una función de usuario debe partir de una sentencia Tol cuyo tipo de datos de retorno sea el mismo que el devuelto por la función. La función anterior puede ser invocada de la siguiente manera:

```
Real a = quad(4);
16.000000
```

Esta sentencia produce el siguiente árbol sintáctico:

```
----> [BINARY]=
 |----> [TYPE]Real
 | |----> [ARGUMENT]a
 |----> [FUNCTION]quad
   |----> [ARGUMENT](4)
```

La evaluación del árbol se realiza en los siguientes pasos:

1. Llamada a `BGrammar::Evaluate(BList*)` con el árbol anterior
2. Al leer el token "=", se delega en `BEqualOperator::Evaluate`, que verifica que la parte izquierda de la expresión, es decir el tipo y nombre de la variable, son correctas. A continuación `BEqualOperator::Evaluate` delega en `CreateObject` para crear la variable con el resultado de la evaluación de la parte derecha del árbol.
3. Llamada a `BGrammar::Evaluate(BList*)` con la rama derecha de la expresión, es decir, la llamada a la función, que delega en `BUserFunction::Evaluate`.
4. Llamada a `BUserFunction::Evaluate` que delega en `BStandardOperator::Evaluate`
5. Llamada a `BStandardOperator::Evaluate`, que evalua los parámetros de la llamada a la función y crea una lista con ellos, que utiliza para invocar al evaluador de la función, `BUserFunction::Evaluator`. La evaluación de los parámetros en éste método vuelve a invocar a `BGrammar::Evaluate(BList*)`, lo que permite pasar expresiones en los parámetros de llamada a una función, p. ej.: `Real b = quad(quad(a));`
6. Llamada a `BUserFunction::Evaluator`, que ejecuta la función y retorna un `BSyntaxObject` con el resultado.

Aunque es una descripción muy esquemática, permite observar que el método BGrammar::Evaluate(BList*) es el motor de la evaluación de expresiones en Tol. Desde éste se delega en otros para obtener el resultado final, que pueden volver a invocar al principal para evaluar el paso de parámetros.

5. Entorno de Ejecución

5.1. Gestión del Ámbito

Las sentencias de un programa Tol pueden estar en el ámbito global o dentro de un ámbito local. Las variables y funciones creadas en el ámbito global se almacenan en la Tabla de Símbolos, mientras que el resto de variables se almacena en los registros de activación locales, lo que en Tol se llama el LocalScope, la Pila Local.

El ámbito de ejecución (controlado con el atributo `static int level_` de la clase `BGrammar`) toma el valor `0` cuando el programa esta en el ámbito global y toma un valor `>0` cuando está en algún ámbito local. El cambio de ámbito, del global a un ámbito local, o entre ámbitos locales hacia uno más interior, se produce cuando el Evaluador encuentra un símbolo de apertura de llaves: "`{`".

En el siguiente extracto de código, Tol entra dos veces en un nivel de ejecución local:

```
Real double(Real p) { p*p };
Real a = 2;
Real b = { double(Real a = 3) };
Real c = 3;
```

Por un lado el ámbito de ejecución dentro de la función `double`, y por otro lado el ámbito creado en la parte derecha de la sentencia de asignación de la variable `b`, son ámbitos locales cuyo nivel de ejecución es 1 (`BGrammar::level_ == 1`).

Dejando a un lado la utilidad del código mostrado en el ejemplo, éste sirve para demostrar que el ámbito dentro de las llaves es mayor que el ámbito fuera de ellas, pues de lo contrario habría un conflicto entre la variable `Real a = 2;` y el parámetro de la llamada a la función double, declarado como `Real a = 3;`

El siguiente ejemplo muestra el uso de código en el ámbito global y en el ámbito local, existiendo 2 niveles para éste último:

```
Real fLocal1 (Real p) {
  Real fLocal2 (Real p) {
    Real x = p^p;
    x
  }
  Real x = double(p);
  fLocal2(x)
};
Real salida = fLocal1(123);
```

La última sentencia y la declaración de la función `fLocal1` pertenecen al ámbito global. El código dentro de la función `fLocal1` está en el ámbito local con `level_ == 1`, así como la declaración de la función `fLocal2`. El código de la función `fLocal2` pertenece al ámbito local con `level_ == 2`.

5.1.1. Implementación

La variable que controla el nivel de ejecución en el que se encuentra el intérprete en cada momento es la variable `static int level_` de la clase `BGrammar`. Esta variable se incrementa y decrementa en dos métodos:

1. En el método raíz del evaluador, `BGrammar::Evaluate`, (graimp.cpp)
2. En el Evaluador de funciones de usuario (método `BUserFunction::Evaluator`, en oprimp.cpp)

Solo dentro de estos dos métodos se llevan a cabo los cambios de ámbito de ejecución.[9]

Control del ámbito en el método raíz del Evaluador

Cada vez que se lee la raiz del árbol sintáctico en el método principal del Evaluador, BGrammar::Evaluate(BList*), se comprueba si el token leído representa un cambio de ámbito. De ser así, la dicho token debe forzar un aumento del ámbito antes de continuar la evaluación, y una disminución trás terminarla.

En el siguiente ejemplo de código Tol, se muestra dos sentencias que crean dos variables Real. La 2ª sentencia crea un ámbito local en el lado derecho de la expresión. El grupo de sentencias del ámbito local están separadas entre sí por el token ";" siendo éste la raiz del árbol sintáctico del contenido de dicho ámbito. Como se puede observar en el árbol sintáctico de las dos sentencias, el nodo principal del grupo de sentencias del ámbito local tiene asociados sendos tokens de inicio y cierre del ámbito:

```
Real a = -2;
Real b = { Real res = If(a>0, a, Abs(a)); res };
Real c = 4;
```

[9] La excepción a esta regla se encuentra en las funciones `DBSeriesXXX` implementadas en dbspool.cpp, en el directorio tol/btol/bdb, que pueden devolver variables de tipo Serie al ámbito global para lo que fuerzan un cambio de ámbito al crear éstas.

Genera el árbol sintáctico siguiente:

```
----> [SEPARATOR];
  |----> [BINARY]=
  | |----> [TYPE]Real
  | | |----> [ARGUMENT]a
  | |----> [MONARY] -
  |   |----> [ARGUMENT]2
  |----> [BINARY]=
  | |----> [TYPE]Real
  | | |----> [ARGUMENT]b
  | |----> [SEPARATOR]{;}                <--- "{" y "}" delimitan el ámbito
  |   |----> [BINARY]=
  |   | |----> [TYPE]Real
  |   | | |----> [ARGUMENT]res
  |   | |----> [FUNCTION]If
  |   |   |----> [BINARY]>
  |   |   | |----> [ARGUMENT]a
  |   |   | |----> [ARGUMENT]0
  |   |   |----> [ARGUMENT]a
  |   |   |----> [FUNCTION]Abs
  |   |     |----> [ARGUMENT](a)
  |   |----> [ARGUMENT]res
  |----> [BINARY]=
    |----> [TYPE]Real
    | |----> [ARGUMENT]c
    |----> [ARGUMENT]4
```

Cuando el Evaluador lee el token ";", raiz de las sentencias del ámbito local, debe aumentar la profundidad del ámbito, para a continuación continuar con la evaluación del resto del árbol. Al terminar éste, debe decrementar de la profundidad para continuar con la siguiente sentencia.

Control del ámbito en el Evaluador de Funciones de Usuario

Cuando el Evaluador de Sentencias, detecta una llamada a una función, pasa a evaluarla, realizando las siguientes tareas:

1. Registro de la función en la lista de funciones activas (en el atributo `BUserFunction::activeFunctions_`)
2. Creación de la lista de parámetros usados para llamar a la función (en el método `BStandardOperator::Evaluate`).
3. Inserción de los parámetros en el LocalScope, y evaluación de la función (en el método `BUserFunction::Evaluator`)

De estos tres pasos, el último añade los parámetros de la función creados en el paso 2 al registro de activación de la función. Antes de llevarse a cabo esta acción, el Evaluador cambia el ámbito de ejecución, posteriormente se meten los parámetros de la función en el ámbito de la función, y finalmente se invoca al evaluador pasándole como árbol sintáctico el atributo `definition_` de la `BUserFunction`, el cual contiene el código que define a la función.

5.1.2. Ámbito Estático y Ámbito Dinámico

Tol permite la ejecución de código utilizando bien ámbito estático o bien ámbito dinámico.

El primero es también conocido como ámbito léxico. La ejecución con ámbito estático define el ámbito de cada variable por la posición de ésta en el programa escrito. El programa es algo estático, por lo que el ámbito de cada variable o función se puede saber de antemano observando el código que se va a ejecutar.

Por el contrario, un programa en tiempo de ejecución es dinámico, por lo que una implementación del ámbito dinámico implica que las variables se resuelven por el momento en el que se crean al ejecutarse el código, independientemente de la posición que ocupen en el programa escrito.

El siguiente ejemplo ofrece distintos resultados en función del tipo de ámbito con el que se esté ejecutando Tol:

```
Real r = 4;
Real fun(Real q) {q+r};
Real b = { Real r = 1; fun(8) };
```

Utilizando ámbito estático, la variable `Real b` debe valer `12`, en cambio, usando ámbito dinámico, deberá valer `9`.

El ámbito estático dice que la función “`fun`” está declarada en el ámbito global, y su código interno tiene un ámbito local 1, por lo que la referencia a la variable “`r`” solo puede encontrarse dentro de su propio ámbito, o en alguno exterior a ella. Así, al no haberse renombrado la variable “`r`” ni como parámetro, ni como variable interna de la función, ésta debe tomar el valor `4` que corresponde con el valor de la variable global “`r`”.

El ámbito dinámico dice que cuando se invoca `fun(8)`, en un ámbito local `1`, previamente se ha renombrado la variable “`r`” con el valor `1`, por lo que la referencia a “`r`” que aparece en el código de la función fun debe referirse al valor `1`.

En el siguiente apartado se explica la implementación de la pila local, que puede gestionarse de dos modos distintos atendiendo a que tipo de ámbito este usando Tol. El tipo de ámbito se puede seleccionar al compilar el lenguaje. Ejecutando el script configure con la opción `--enable-DS` se activa el Scope Dinámico. Por defecto se compila con Scope Léxico.

5.2. La Pila Local

Las variables creadas en scope local se almacenan en la Pila Local, representada por el atributo BList *stack_ de la clase BGrammar. La inserción de las variables en la pila se lleva a cabo por el método AddObject, usado siempre que se crea una nueva variable. AddObject comprueba el atributo level_ para decidir si debe insertar en la Tabla de Símbolos o en la Pila Local.

Dependiendo del tipo de Scope que use Tol, la gestión de la pila tendrá unos requirimientos distintos. Con Scope Dinámico, la gestión de la Pila Local es más ligera que con Scope Léxico.

5.2.1. Control del nivel del ámbito en la Pila Local

Haya donde se modifique el valor del ámbito es necesario anotar el tamaño de la pila de cara a poder eliminar los elementos que se inserten en ésta trás abandonar el nivel del ámbito en el que se ha entrado.

El método DestroyStackUntil requiere conocer hasta donde deben eliminarse elementos de la Pila.

En la implementación de Tol, solo se modifica el ámbito en dos puntos, vistos en 5.1.1.

5.2.2. Control de los niveles de búsqueda en Scope Léxico

La Pila Local almacena las variables creadas en los ámbitos locales en la Pila Local. Otra estructura, llamada stack_level, perteneciente a BGrammar, lleva la cuenta del ámbito en el que se está en cada momento así como del tamaño de la pila justo en el momento del cambio. Esta anotación es empleada para resolver la búsqueda de variables con Scope Léxico.

En el ejemplo:

```
Real r = 4;
Real fun(Real q) {q+r};
Real b = { Real r = 1; fun(8) };
```

La referencia a la variable “r” que aparece en el código de “fun” no debe ser confundida con la variable r=1 creada antes de la llamada fun(8). Para ello interviene en la búsqueda la estructura stack_level. El método FindLocal de BGrammar, lleva a cabo la búsqueda. Dicho método aparece duplicado en el fichero graacc.cpp, en el directorio tol/btol/bgrammar. Cada método da soporte a cada uno de los dos tipos de Scope soportados. La macro de compilación __USE_DYNSCOPE__ se encarga de seleccionar uno u otro.[10]

6. Implementación de los Tipos de Datos

Tol tiene 11 tipos de datos. **`Real`**, **`Date`**, **`Set`**, **`Code`**, **`Text`**, **`Serie`**, **`TimeSet`**, **`Complex`**, **`Matrix`**, **`Ratio`**, y **`Polyn`**.

6.1. Introducción

Cada tipo de dato, dependiendo de la naturaleza de la información que esté representando, se implementa de manera distinta. Un `Real` y un `Text` almacenan información diferente. Lo mismo ocurre con un Número Complejo, con una Matriz o con cualquiera de los tipos que ofrece Tol. Cada uno representa, y por lo tanto almacena, información distinta. Sin embargo. todos estos distintos tipos de datos presenta una interfaz común con el núcleo del lenguaje.

El almacenamiento de las instancias de los tipos de datos se realiza en objetos de clases básicas. Podemos encontrar las siguientes clases básicas dedicadas al almacenamiento y representación básica de cada tipo de dato:

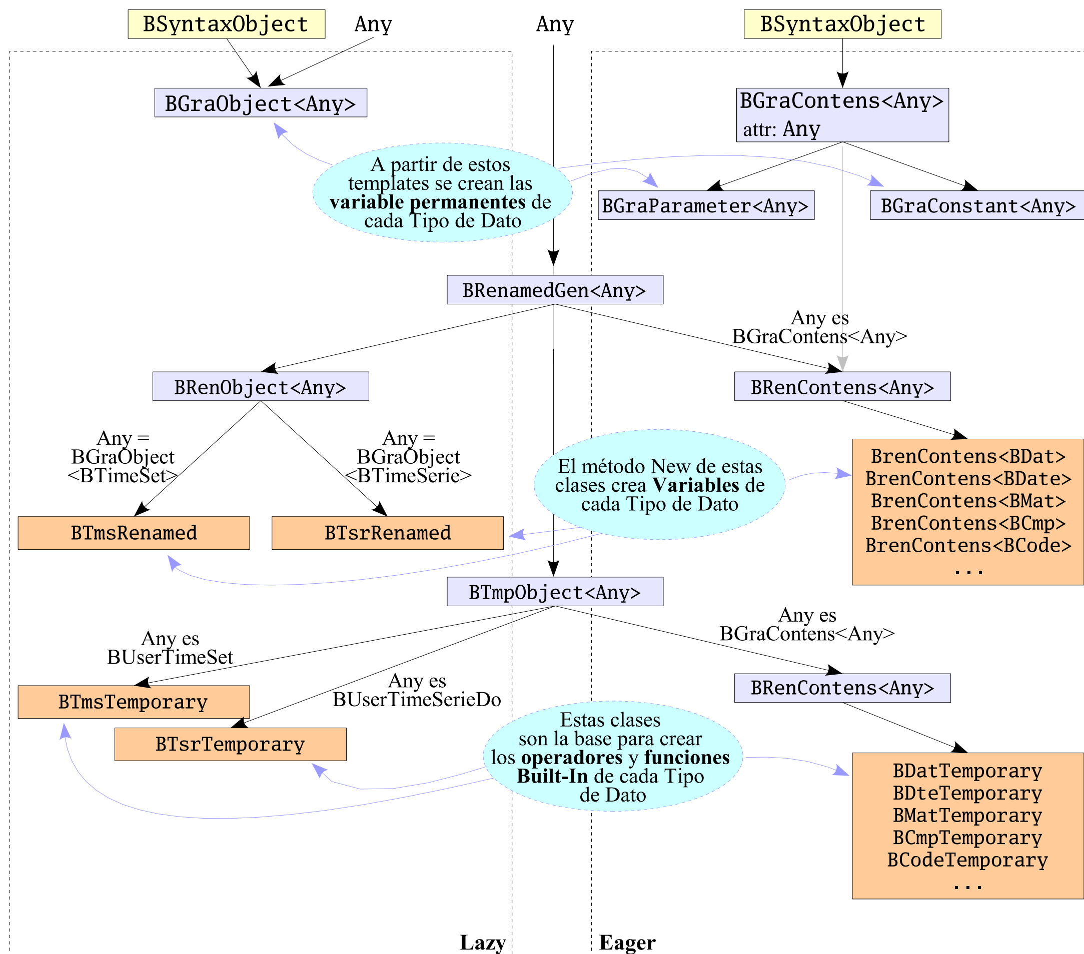
1. La clase `BDat`, representa el almacenamiento de las variables de tipo `Real`
2. La clase `BDate`, representa el almacenamiento de las variables de tipo `Date`
3. La clase `BSet`, representa el almacenamiento de las variables de tipo `Set` (conjunto)
4. La clase `BCode`, representa el almacenamiento de las variables de tipo `Code`
5. La clase `BText`, representa el almacenamiento de las variables de tipo `Text`
6. La clase `BTimeSerie`, representa el almacenamiento de las variables de tipo `Serie`
7. La clase `BTimeSet`, representa el almacenamiento de las variables de tipo `TimeSet`
8. La clase `BComplex`, representa el almacenamiento de las variables de tipo `Complex`
9. La clase `BMatrix`, representa el almacenamiento de las variables de tipo `Matrix`
10. La clase `BRational`, representa el almacenamiento de las variables de tipo `Ratio`
11. La clase `BPolyn`, representa el almacenamiento de las variables de tipo `Polyn`

Cada tipo de dato tiene que compartir una interfaz común que permita al núcleo gestionar las variables independientemente de la información que representen. Son necesarias dos interfaces para distinguir entre dos conceptos distintos:

- Interfaz de Gestión del Tipo de Dato o Gramática
- Interfaz de Gestión de las variables

[10]Por defecto Tol se compila con Scope Léxico. Para activar el Scope Dinámico es necesario ejecutar el script configure con la opción –enable-DS

Figura 7: Templates para la implementación de los Tipos de Datos



La primera interfaz representa al Gestor de cada Tipo de Dato. Cada gestor debe permitir, a través de sus métodos, crear variables del tipo de dato al que representa, acceder a ellas, ejecutar sus funciones, usar sus propios operadores, interpretar las constantes, y en definitiva llevar a cabo cualquier tarea relacionada con el Tipo de Dato al que representa.

La segunda interfaz, que hemos denominado “Interfaz de Gestión de Variables”, representa cada instancia de cada tipo de dato creado en Tol, es decir, cada variable creada, de manera que el núcleo de Tol puede manejar la variable, independientemente de su tipo, utilizando dicha interfaz. Está representada por la clase `BSyntaxObject`, explicada en 2.1.

Entre las dos interfaces se definen los Tipos de Datos. Éstos se definen mediante el uso de un grupo de templates que se comentan a continuación.

6.2. Templates

El grupo de Templates C++ que definen la forma interna de las Constantes, Variables del Sistema, Variables de Usuario y Funciones Built-in, de cada Tipo de Dato del lenguaje están definidos en los ficheros tol_bgencon.h, tol_bgentmp, y tol_bgenobj.h, en el directorio tol/btol/bgrammar.

La combinación de las clases básicas (BDat, BText...) y los templates da como resultado la implementación de los tipos de datos de Tol.

Según la figura 7 existen dos tipos de datos diferenciados por el momento en el que se evalua su

contenido: tipos de datos Eager y tipos de datos Lazy. Los únicos representantes de los últimos son los Conjuntos Temporales y las Series Temporales, es decir, los tipos TimeSet y Serie respectivamente.

Como muestran los globos azules de la figura, la instanciación de los templates será empleada para implementar todos los diferentes tipos de objetos evaluables de cada tipo de dato: Constantes, Variables, Operadores y Funciones Built-In.

6.3. Tipos de Datos perezosos (Lazy)

Los tipos de datos perezosos se implementan usando el template `BGraObject<Any>`.

La clase `Any` especifica la clase básica que se utilizará para representar el tipo de dato.

Tol solo cuenta con dos tipos lazy, cada uno utilizará su clase básica correspondiente para ser implementado. El tipo `TimeSet` esta implementado en los ficheros del directorio tol/btol/timeset_type. El fichero tol_btmsgra.h define una macro de preprocesador que indica que el mnemotécnico `BUserTimeSet` es realmente `BGraObject<BTimeSet>`, también define un grupo de funciones que convierten de `BSyntaxObject` a BUserTimeSet de manera que se pueda manejar la interfaz de `TimeSet` después de realizar el casting. Estas funciones son ampliamente utilizadas en la implementación de cada operador y función Built-In del tipo de dato, como se puede ver en los ficheros tmsgra.cpp y tmsgrav.cpp.

Por otro lado, el tipo `Serie` esta implementado en los ficheros del directorio tol/btol/serie_type. El fichero tol_btsrgra.h define una macro de preprocesador que indica que el mnemotécnico `BUserTimeSerie` es realmente `BGraObject<BTimeSerie>`, también define un grupo de funciones que convierten de `BSyntaxObject` a `BUserTimeSerie`, al igual que ocurre con `BUserTimeSet`.

6.4. Tipos de Datos impacientes (Eager)

Los tipos de datos Eager se implementan usando el template `BGraContens<Any>`.

La clase `Any` especifica la clase básica que se utilizará para representar al tipo de dato, y la llamada a `CastingDeclaration(Any)` en cada tipo de dato, completa la declaración de funciones para el casting de `BSyntaxObject` al tipo creado.

La macro `CastingDeclaration` esta definida en tol_bgencon.h, en tol/btol/bgrammar.

6.5. Declaración de los Tipos de Datos

6.5.1. Tipo Real

Directorio tol/btol/real_type
Fichero principal: tol_bdatgra.h
Clase Básica: `BDat` (en tol/bmath/mathobjects/tol_bdat.h)

```
#define BUserDat   BGraContens   <BDat>
#define BSystemDat BGraConstant  <BDat>
#define BParamDat  BGraParameter <BDat>
CastingsDeclaration(Dat);
```

6.5.2. Tipo Date

Directorio tol/btol/date_type
Fichero principal: tol_bdtegra.h
Clase Básica: `BDate` (en tol/bbasic/tol_bdate.h)

```
#define BUserDate   BGraContens   <BDate>
#define BSystemDate BGraConstant  <BDate>
#define BParamDate  BGraParameter <BDate>
#define BDteTemporary BTmpContens <BDate>
CastingsDeclaration(Date);
```

6.5.3. Tipo Set

Directorio tol/btol/set_type
Fichero principal: tol_bsetgra.h
Clase Básica: `BSet` (en tol/btol/set_type/tol_bset.h)

```
#define BUserSet       BGraContens  <BSet>
#define BSystemSet     BGraConstant <BSet>
#define BSetTemporary  BTmpContens  <BSet>
CastingsDeclaration(Set);
```

6.5.4. Tipo Code

Directorio tol/btol/code_type
Fichero principal: tol_bcodgra.h
Clase Básica: `BCode` (en tol/btol/code_type/tol_bcode.h)

```
#define BUserCode        BGraContens  <BCode>
#define BSystemCode      BGraConstant <BCode>
#define BCodeTemporary   BTmpContens  <BCode>
CastingsDeclaration(Code);
```

6.5.5. Tipo Text

Directorio tol/btol/text_type
Fichero principal: tol_btxtgra.h
Clase Básica: `BText` (en tol/bbasic/tol_btext.h)

```
#define BUserText      BGraContens   <BText>
#define BSystemText    BGraConstant  <BText>
#define BParamText     BGraParameter <BText>
#define BTxtTemporary  BTmpContens   <BText>
CastingsDeclaration(Text);
```

6.5.6. Tipo Serie

Directorio tol/btol/serie_type
Fichero principal: tol_btsrgra.h
Clase Básica: `BTimeSerie` (en tol/btol/serie_type/tol_btmser.h)

```
#define BUserTimeSerie BGraObject<BTimeSerie>
```

6.5.7. Tipo TimeSet

Directorio tol/btol/timeset_type
Fichero principal: tol_btmsgra.h
Clase Básica: `BTimeSet` (en tol/btol/timeset_type/tol_btmset.h)

```
#define BUserTimeSet BGraObject<BTimeSet>
```

6.5.8. Tipo Complex

Directorio tol/btol/complex_type
Fichero principal: tol_bcmpgra.h
Clase Básica: `BComplex` (en tol/bmath/tol_bcomplex.h)

```
#define BUserCmp      BGraContens  <BComplex>
#define BSystemCmp    BGraConstant <BComplex>
#define BCmpTemporary BTmpContens  <BComplex>
CastingsDeclaration(Cmp);
```

6.5.9. Tipo Matrix

Directorio tol/btol/matrix_type
Fichero principal: tol_bmatgra.h
Clase Básica: `BMat = BMatrix<BDat>` (en tol/bmath/tol_bmatrix.h)

```
#define BUserMat      BGraContens  <BMat>
#define BSystemMat    BGraConstant <BMat>
#define BMatTemporary BTmpContens  <BMat>
CastingsDeclaration(Mat);
```

6.5.10. Tipo Polyn

Directorio tol/btol/polynomial_type
Fichero principal: tol_bpolgra.h
Clase Básica: `BPolyn<BDat>` (en tol/bmath/tol_bpolyn.h)

```
#define BPol             BPolyn    <BDat>
#define BUserPol      BGraContens  <BPol>
#define BSystemPol    BGraConstant <BPol>
#define BPolTemporary BTmpContens  <BPol>
CastingsDeclaration(Pol);
```

6.5.11. Tipo Ratio

Directorio tol/btol/ratio_type
Fichero principal: tol_bratgra.h
Clase Básica: BRational<BDat>(en tol/btol/ratio_type/tol_bratio.h)

```
#define BRat          BRational    <BDat>
#define BUserRat      BGraContens  <BRat>
#define BSystemRat    BGraConstant <BRat>
#define BRatTemporary BTmpContens  <BRat>
CastingsDeclaration(Rat);
```