



UNIVERSIDAD DE OVIEDO

**ESCUELA POLITÉCNICA SUPERIOR
DE INGENIERÍA DE GIJÓN**

LENGUAJES Y SISTEMAS INFORMÁTICOS

PROYECTO FIN DE CARRERA N° 1022010

**DISEÑO Y CONSTRUCCIÓN DE UNA HERRAMIENTA PARA LA
GENERACIÓN AUTOMÁTICA DE SISTEMAS DE TIPOS
ORIENTADOS A OBJETOS**

DOCUMENTO INTERNO

MANUAL DE USUARIO DEL FRAMEWORK TYS



**DANIEL ZAPICO RODRÍGUEZ
MARZO 2003**

TUTOR: D. FRANCISCO ORTÍN SOLER

Tabla de contenidos

1	Objetivos	5
2	Arquitectura del sistema	5
3	Especificación de tipos: Entrada a TyCC	6
3.1	Sintaxis del fichero de entrada	6
3.1.1	Preámbulo	7
3.1.2	Declaración de tipos	7
3.2	Definición de Tipos	8
3.2.1	Parámetros de construcción de tipo	8
3.2.2	Atributos preestablecidos utilizando TyCC	9
3.2.3	Código destino incrustado	9
3.2.4	Definición de operaciones	10
3.3	Ejemplo de uso	15
4	Generador de comprobadores de tipos: TyCC	16
4.1	Manual de usuario	16
4.1.1	Invocación desde la línea de comandos	17
4.1.2	Parámetros de entrada	17
4.2	Funciones de TYCC	18
4.3	Clases generadas por TyCC	19
4.3.1	Clase Type	19
4.3.2	Clase BaseType	21
4.3.3	Clases representantes de los tipos (Type-1, Type-2, etc.)	23
4.3.4	Clase Type Reference	23
4.3.5	Clase NamedType	24
5	Interfaz de procedimientos para la gestión de tipo	25
5.1	Manual básico de usuario	25
5.1.1	Gestión de tipos: getTypeIF(String): TypeIF	25
5.1.2	Tipos con nombre: assignName(TypeIF t, String typeName): TypeIF	26
5.1.3	Tratamiento de los tipos desconocidos: getUnknownTypes():ArrayList	26
5.1.4	Consultor de tipos: isType(String s):boolean	26
5.1.5	TEL: Type Expression Lenguaje	26
5.1.6	Manejador de errores	28
5.2	Manual avanzado del API	28
5.2.1	Inicialización de la tabla de tipos	28
5.2.2	Construcción de tipos: builder(String id, String et, TermList e):TypeIF	29
5.2.3	Representación simbólica de las expresiones de tipo	29
5.2.4	Representación de las expresiones de tipo mediante objetos (Avanzada)	30
5.2.5	Creación de tipos	31
6	Construcción de un compilador usando el sistema TyS	31
6.1	Código fuente del compilador (Compiler Source Code)	32
6.1.1	Módulo Léxico	32
6.1.2	Módulo Sintáctico	34
6.1.3	Módulo semántico	47
6.2	Especificación de tipos (Type Specification)	54
6.2.1	Preámbulo	57
6.2.2	Sección de declaración de tipos	57
6.2.3	Definición de tipos	58
6.2.4	Generación del comprobador de tipos	65

6.2.5	API	66
6.3	Comprobación de adaptabilidad	66
6.3.1	Análisis léxico	67
6.3.2	Análisis sintáctico	67
6.3.3	Análisis semántico	67
6.3.4	Especificación de tipos	73
6.3.5	Conclusiones de la prueba de refactoring	78
6.4	Unificación de la comprobación de tipos estática y dinámica	79
6.4.1	Sentencia Write	79
6.4.2	Sentencia Read	80
6.4.3	Operaciones aritméticas	82
6.4.4	Operaciones módulo (resto)	83
6.4.5	Operaciones lógicas	83
6.4.6	Operaciones relacionales	83
6.4.7	Operaciones de igualdad	86
6.4.8	Operaciones unarias	87
6.4.9	Operaciones de cast explícito	87
6.5	Resultado de la refactorización	88
6.6	Compilador resultante (Compiler .OBJ)	88
6.7	Uso avanzado. Aplicaciones reflectivas	88
7	<i>Interacción con YACC</i>	89
7.1	Invocación desde la línea de comandos	89
7.2	Escritura del fichero de entrada	90
7.3	Interacción con BYACCJ	90
7.4	Ejemplo de uso	90
8	<i>Interacción con ANTLR. comprobación de tipos de un lenguaje OO</i>	91
8.1	Análisis léxico	91
8.2	Sintaxis de Dril	92
8.2.1	Uso y ubicación de operadores	92
8.2.2	Gramática de Drill	94
8.3	Análisis semántico	96
8.3.1	Definición de tipos de datos (Constructores de tipos)	96
8.3.2	Tipos predefinidos	96
8.3.3	Promociones implícitas	97
8.3.4	Promociones explícitas (cast forzado)	97
8.3.5	Herencia	97
8.3.6	Equivalencia de expresiones de tipo	97
8.3.7	Semántica de operadores	98
8.4	Especificación de tipos	100
8.5	Jerarquía de objetos generada	101
8.6	Prueba del analizador semántico	101
9	<i>BIBLIOGRAFÍA</i>	102

Índice de figuras

<i>Figura 1: Arquitectura del sistema TyS</i>	5
<i>Figura 2: Diagrama de clases, que representa la delegación de operaciones de la clase Type en la jerarquía de tipos coronada por BaseType.</i>	19
<i>Figura 3: Diagrama de clases generado por TyCC en colaboración con el API.</i>	21
<i>Figura 4: Diagrama de clases, que representa la acutación de los objetos de clase BaseType, como objetos delegados de Type.</i>	22
<i>Figura 5: Diagrama de clases que representa la referencias a tipos.</i>	24
<i>Figura 6: Diagrama de clases que representa los tipos con nombre.</i>	24
<i>Figura 7: Representación simbólica de las expresiones de tipo.</i>	29
<i>Figura 8: Representación de expresiones de tipo mediante objetos.</i>	30
<i>Figura 9: Arbol sintáctico asociado al programa ejemplo.</i>	40
<i>Figura 10: Arbol sintáctico simplificado asociado al programa ejemplo.</i>	41
<i>Figura 11: Diagrama de clases de las sentencias del lenguaje Frog.</i>	43
<i>Figura 12: Diagrama de clases de las expresiones del lenguaje Frog.</i>	44
<i>Figura 13: Colaboración Visitor que estructura el AST para el lenguaje Frog.</i>	45
<i>Figura 14: Diagrama de clases que modela el patrón Visitor para Frog.</i>	46
<i>Figura 15: Diagrama de objetos, que representa el arbol sintáctico del ejemplo.</i>	47
<i>Figura 16: Diagrama de clases. Jeararquía de clases generada por TyCC para Frog.</i>	65
<i>Figura 17: Diagrama de clases. Jerarquía de tipos generada por TyCC para Frog.</i>	66
<i>Figura 18: Diagrama de clases generado por TyCC para Frog tras añadir dos nuevos tipos.</i>	88
<i>Figura 19: Uso de YACCPT</i>	89
<i>Figura 20: Diagrama de clases. Jerarquía de tipos generada por TyCC para el lenguaje Drill.</i>	101

1 OBJETIVOS

El objetivo de este proyecto es proporcionar un método lo más automatizado posible para la generación de comprobadores de tipos, con la idea de ayudar en la construcción de analizadores semánticos.

Para ello se ha realizado un completo *framework*, el sistema TyS.

Normalmente la construcción de analizadores semántico se hace *ad hoc*, ya sea con las herramientas existentes o diseñado completamente por el usuario.

El *framework* TyS (*TypeChecking System*) automatiza aquellas partes que son genéricas en un comprobador de tipos, proporcionand un interfaz de uso muy reducido y fácil de usar, y un método la generación de comprobadores de tipos específicos a partir de una especificación .

2 ARQUITECTURA DEL SISTEMA

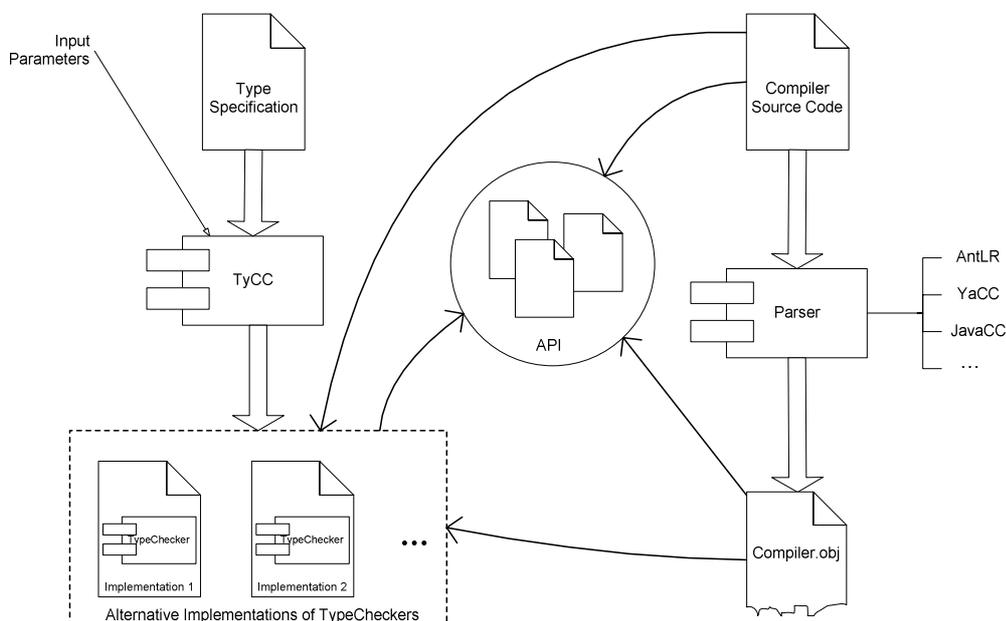


Figura 1: Arquitectura del sistema TyS

La figura precedente nos muestra el mapa de componentes que conforman el sistema TyS.

De manera muy sucinta, el sistema es capaz de integrar una especificación formal de tipos proporcionada por el usuario con el resto de las partes de un procesador del lenguaje, proporcionando de esta manera una separación de incumbencias muy conveniente. Además proporciona para este cometido, un entorno orientado a objetos, en el que se ha hecho especial énfasis en un cuidado diseño para el que se han aplicado diversos patrones de software.

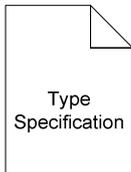
Las partes principales del sistema son un API y el programa TyCC,

TyCC recibe una especificación formal de tipos como fichero de entrada (*Type Specification*) y a partir de ahí y colaborando con un API, genera un comprobador de tipos que puede ser usado en tiempo de compilación o dinámicamente.

En función de los parámetros de entrada (*input parameters*) es capaz de generar distintas implementaciones de comprobadores de tipos (*alternative implementations of TypeCheckers*). Todo ello se integrará con el código fuente del compilador que se desea

diseñar (*compiler source code*), y con ayuda de un compilador adecuado (*Parser*) que puede o no apoyarse en diversas herramientas de generación automática de analizadores generara el compilador adecuado (*compiler.obj*)

3 ESPECIFICACIÓN DE TIPOS: ENTRADA A TYCC



En este punto se detalla el uso del fichero de entrada a la aplicación TyCC. Al final del mismo se incluye un ejemplo de uso. La definición los tipos que formaran parte del lenguaje de programación que se quiere diseñar se realiza en un fichero de texto que sigue una serie de reglas.

Este fichero sirve como entrada a la aplicación TyCC que realizará un análisis de la validez de dicho fichero, y si todo es correcto generará un fichero con una implementación particular de un comprobador de tipos.

En este fichero se definen tanto los tipos como las operaciones que estos pueden realizar. Es posible indicar las promociones implícitas entre los tipos de una manera sencilla y rápida. Si se quieren especificar operaciones particulares sobre un tipo determinado, se declara y define éste, de manera similar a la definición de una clase JAVA.

3.1 Sintaxis del fichero de entrada

El fichero de entrada sigue la siguiente estructura:

```
Fichero_de_entrada ::= (Preámbulo)?
                    Declaracion_Tipos
                    Definición_de_clases
                    ;

Preámbulo ::= CodigoIncrustado

Declaracion_Tipos ::= TYPES '=' '{' {cadena_promociones}'
                    ;
Cadena_promociones ::= ID {'<' ID}
                    ;
Definición_de_Clases ::= { Definición_de_clase }
                    ;
Definición_de_Clase ::= CLASS ID['(' Arg {',' Arg} ')']
                    '{'
                    { DefiniciónMetodo |CodigoIncrustado }
                    '}'
                    ;
CodigoIncrustado ::= "%{" .* "%}"
                    ;

DefiniciónMétodo ::= ReturnType ID MethodArgs Body
                    ;
ReturnType ::= Type
                    ;

MethodArgs ::= '(' [Type ID {',' Type ID}] ')'
                    ;
Type ::= ID {'[' ']' }
                    ;
Body {' .* '}'
                    ;
```

3.1.1 Preámbulo

En esta zona separada por las marcas `%{` y `%}` se puede meter todo el código que se desee. Esta zona se copiará al comienzo del fichero *Type.java* que genera la aplicación TyCC.

Es el lugar donde apropiado para introducir cláusulas *import*.

```
%{  
import java.lang.*;  
%}
```

3.1.2 Declaración de tipos

En esta sección se declaran todos los tipos y se indican las posibles promociones implícitas entre tipos. De esta forma se genera automáticamente el código para las conversiones automáticas entre tipos.

Por ejemplo suponiendo que se va a definir un lenguaje con 4 tipos *Void*, *Char*, *Integer* y *Real* y que existe una promoción implícita de *Integer* a *Real* y de *Char* a cualquiera de los dos anteriores. El tipo *Void* no se promociona a ningún tipo

Esto se expresaría en la entrada a TyCC con una declaración de tipos de la siguiente manera.

```
Types = { Char < Integer < Real Void }
```

De esta manera tan sencilla se ha especificado que el lenguaje consta de 4 tipos y las promociones implícitas entre 3 de ellos.

Como se ha visto es necesario poner en la sección de declaración de tipos todos los tipos que formarán parte del lenguaje aunque no haya ninguna promoción entre ellos.

En el caso de que se quiera definir una conversión implícita para un tipo determinado una conversión implícita más sofisticada se puede realizar definiendo para este tipo, la operación *implicitCast (Type):Type*

Por ejemplo si se quieren añadir clases a este lenguaje, y entre clases puede haber coerción implícita si se trata de clases que está más arriba en el árbol de herencia (*Upcasting*). Esta característica no podríamos expresarla en la sección de declaración de tipos, así que definimos una operación *implicitCast()* para clases. Esto es:

```
Types = { Char < Integer < Real Void Class }
```

Como vemos no queda definida la coerción implícita entre clases. Esto se realizará definiendo la operación *implicitCast()*.

```
class Class {  
...  
Type implicitCast(Type t) {  
    If t is a base class of this class do de casting  
}  
...  
}
```

De la sintaxis exacta para definir operaciones hablaremos en 3.2.4. Y un ejemplo de especificación con tipos se encuentra en 2.3.

3.2 Definición de Tipos

El cometido de la entrada a TyCC es definir tipos de datos. Para ello se definen los tipos de datos en forma de clases con una sintaxis muy similar a la del lenguaje Java.

Así pues además de un preámbulo y una declaración de tipos, un fichero de entrada a TyCC consta de una sección de definición de clases (tipos).

Las clases se definen poniendo la palabra reservada `class` seguido del nombre del tipo que vamos a definir, opcionalmente seguido por los parámetros componente entre paréntesis. A continuación entre llaves una serie de definiciones de métodos o código incrustado.

```
Definición_de_Clase ::= CLASS ID[ '(' Arg { ',' Arg } ') ' ]
                        '{ '
                        { DefiniciónMetodo | CódigoIncrustado }
                        '}'
```

Dado que este tipo se mapeará directamente en una clase en el lenguaje de implementación destino; será necesario usar el léxico adecuado para los identificadores de clase. Por ejemplo si el lenguaje destino va a ser Java, el nombre del tipo deberá tener la primera letra en mayúsculas.

3.2.1 Parámetros de construcción de tipo

Indican en el caso de tipos compuestos de datos, o constructores de tipos la clase de dato de que están compuestos. Por ejemplo un tipo de datos arreglo (*array*), tiene una componente de clase *Tipo*, es decir un arreglo es un arreglo de “algo”.

Representar la estructura de un tipo, es mostrar la disposición de sus componentes. Lo que se quiera representar depende de los deseos del usuario, y es jugando con estos argumentos como se puede conseguir.

Las componentes de un tipo de datos no tienen porque ser únicamente tipos de datos, también pueden ser datos en “bruto” (*raw data*). Por ejemplo en la definición de un arreglo si se desea que la dimensión forme parte de su representación entonces se debe indicar que dicho tipo contiene además un elemento que indica su dimensión.

La forma de indicar datos en bruto será mediante cadenas de caracteres.

Si no se desease que la dimensión formase parte de la representación de un tipo de dato arreglo, entonces no se podría distinguir entre un arreglo u otro de un mismo tipo. Es decir los datos componentes de un tipo sirven para preservar su identidad ante instancias del mismo tipo.

Estos datos pueden ser de 4 tipos, y cada uno de ellos tiene una palabra reservada.

1. *Tipos de datos*. Palabra reservada *Type*
2. *Literales de caracteres*. Palabra reservada *String*
3. *Listas de tipos de datos*. Palabra reservada *Type+*
4. *Listas de literales de caracteres*. Palabra reservada *String+*

Diversos ejemplos

Un tipo puntero:

```
class Pointer (Type) { ... }
```

Un tipo arreglo en el que queremos que la dimensión forme parte de su representación.

```
class Array (Type, String) { ... }
```

Un tipo registro

```
class Record (Type+, String+) { ... } //First arg type of the
                                     fields, second their name
```

Un tipo función

```
class Function (Type+, Type) { ... } //First arg, function args,
                                     second type of return
```

Después de esta cabecera las clases constarán de secciones opcionales de código incrustado de lenguaje destino (el lenguaje en el que finalmente se generará la jerarquía de clases), y definición de métodos. El cuerpo de los métodos será también en el lenguaje destino.

3.2.2 Atributos preestablecidos utilizando TyCC

Si se utiliza la aplicación TyCC, existen en las clases una serie de métodos y atributos preestablecidos.

Se utilizan dos vectores de objetos para almacenar los datos componentes de un tipo que representan su estructura.

- *typeFields* que almacena los objetos de clase *Type*
- *dataFields* que almacena los objetos de clase *String*. Para almacenar datos en bruto.

Como se mostraba en el punto anterior los componentes de un tipo se indican en el momento de la construcción. A partir de una expresión de tipo se van construyendo todas las componentes y estas son pasadas al objeto tipo durante su construcción.

Estos dos atributos sirven precisamente para almacenar estos componentes. Los objetos de clase *Type* son almacenados consecutivamente en el vector *typeFields* y los objetos de tipo *String* en el vector *dataFields*. El almacenamiento se produce en posiciones crecientes del vector correspondiente según se haya especificado en la cabecera de la clase. Unos ejemplos clarificarán esta explicación.

```
class A(Type1, String2, Type3+, Type4, String5+,Type6)
```

Después de la construcción el contenido de los atributos es el siguiente:

```
typeFields[0]      <- Type1
typeFields[1..n]  <- type31..type3n
typeFields[n+1]   <- Type6
dataFields[0]     <- String2
datafields[1..m]  <- String61..String6m
```

3.2.3 Código destino incrustado

En el interior de una clase se puede incrustar trozos de código fuente del lenguaje destino. La forma de incrustar código es parecida a la que se mostró en el preámbulo, pero fuera de la definición de un método pero en el interior de una clase.

En esa zona se puede escribir el código fuente que se desee delimitado por las marcas `%{` y `%}`. Este código no será tratado por TyCC, sino que se escribirá directamente en el fichero de salida.

Por ejemplo si se desea añadir una clase *Integer* información sobre su rango pero que no aparezca en la representación de su estructura. Lo que se debe hacer es tratarlo como información a incrustar. Simplemente se pueden añadir dos constantes enteras, que indiquen el límite inferior y superior. Esto en Java sería:

```
Class Integer {
    . . .
    %{ //Embbeded code
    final static int low = +32767
    final static int high= -32768
    %}
    . . .
}
```

3.2.4 Definición de operaciones

Este punto trata de la característica más interesante de esta mecanismo de especificación de tipos: la posibilidad de añadir operaciones en forma de métodos de clase.

La sintaxis de definición de los métodos es similar a la del lenguaje Java

Primeramente se escribe la cabecera del método y después el cuerpo entre llaves.

```
DefiniciónMétodo ::= Type ID '(' MethodArgs ')' '{' .* '}'
                    ;
MethodArgs ::= [Type ID {',' Type ID}]
                    ;
Type ::= ID { '[' ']' }
```

En la cabecera primeramente se declara el tipo de retorno, seguido del nombre del método. Después los argumentos que puede recibir el método y a continuación entre llaves el código del mismo.

3.2.4.1 Tipo de retorno

El tipo de retorno se declara como un identificador al que opcionalmente le pueden seguir una serie de corchetes

```
Type ::= ID { '[' ']' } ;
```

Los identificadores de los tipos siguen la siguiente sintaxis.

```
alpha ::= a..z | A..z ;
alnum ::= alpha | '0' .. '9' ;
ID ::= [alpha][{ alnum | '.' } alnum] ;
```

Es decir ejemplo de identificadores de tipo válido serían: `Integer`, `hola22`, `I.a.b`

De esta manera se pueden emplear tipos que formen parte de un paquete (*package*) o de otra clase. Pues en el propio identificador va incluido el operador de ámbito.

Los corchetes se usarán si se desea devolver objetos arreglo.

3.2.4.2 Nombrado de métodos

El nombre de un método tiene que ser un identificador válido estilo Java.

```

alpha ::= a..z | A..z
alnum ::= alpha | '0' .. '9'
ID ::= [alpha]{ alnum }

```

Se permite la sobrecarga de métodos. Es decir que dos métodos pueden tener el mismo nombre y diferenciarse únicamente en sus argumentos. Esta sobrecarga se hace al estilo Java [Eckel98], ya que todos los métodos van a ser virtuales. No se permite diferenciación en el tipo de retorno como en C++ [Stroustrup94b]

Hay una serie de métodos que están ya preestablecidos, unos se pueden redefinir y otros son reservados del sistema, por lo que no se pueden utilizar. De todo ello se habla en 3.2.4.5.

3.2.4.3 Argumentos de método

Todo método puede tener una lista de argumentos.

```

MethodArgs ::= [Type ID {',' Type ID}] ;
Type ::= ID { '[' ']' } ;

```

Los tipos que se pueden tomar como parámetro son los mismos que puede devolver el método (*ver 3.2.4.1*). Es decir los identificadores de los tipos siguen la siguiente sintaxis.

```

Type ::= ID { '[' ']' } ;
ID ::= [alpha][{ alnum | '.' } alnum] ;
alpha ::= a..z | A..z ;
alnum ::= alpha | '0' .. '9' ;

```

De esta manera se pueden emplear tipos que formen parte de un *package* o de otra clase, ya que en el propio identificador va incluido el operador de ámbito.

Al igual que con los tipos de retorno los corchetes se usarán si se desea devolver objetos arreglo.

3.2.4.4 Cuerpo de los métodos

El cuerpo de los métodos se escribe de la siguiente manera: delimitado por dos llaves se puede escribir el código fuente en el lenguaje que se desee.

```

Body '{' .* '}'

```

Se se tienen en cuenta los comentarios estilo C/C++. Es decir que algo así

```

Metodo {
    . . .
    // }

```

no sería el tomado como el fin del método.

Lo que se escriba entre esas dos llaves será copiado directamente en un método correspondiente de la clase tipo que se está definiendo.

Hay que hacer una aclaración. Se están definiendo métodos para clases que representan tipos, pero el interfaz que se va a mostrar al usuario es en forma de una única clase *Type*. Esta clase es la que se encarga de delegar responsabilidades en el resto de clases que representan tipos. Así pues estas clases no son visibles para el usuario.

Pueden darse situaciones un poco confusas, sobre todo si se desean hacer referencias a la instancia de la clase que ejecuta el método. Por ejemplo, se intenta definir la función *majorType()* entre tipos integrales, en un lenguaje formado por enteros y reales

```
Class Integer {
    Type majorType (Type t) {
        if (t.getType() == "Real") return t;
        return this // !!WRONG
    }
}
```

Lo lógico sería escribir el método tal y como se muestra en el código precedente. Sin embargo es incorrecto, pues se pide que se devuelva un objeto de clase *Type*, y se está escribiendo el cuerpo del método de un objeto de clase *Integer*. El problema es que la clase *Integer* y todas las que se definan estarán ocultas al usuario y siempre que se quieran usar los objetos tipo será a través de la clase *Type*.

La única manera que tiene el usuario de acceder al objeto *Type* que recubre a la instancia de la clase que está definiendo es acceder a la tabla de tipos.

Esto es:

```
Class Integer {
    Type majorType (Type t, Type.TypeTablesAdapter tt) {
        if (t.getType() == "Real") return t;
        return tt.getType("Integer"); //OK
    }
}
```

O si se usa la tabla de tipos en forma de variable de clase que proporciona la clase *Type*

```
Class Integer {
    Type majorType (Type t) {
        if (t.getType() == "Real") return t;
        return Type.typesTable.getType("Integer"); //OK
    }
}
```

3.2.4.5 Métodos preestablecidos

Existen una serie de métodos de la clase *Type* que están preestablecidos, es decir que siempre se ha de contar con la existencia de esos métodos. Hay dos tipos de métodos los métodos predefinidos y los reservados.

3.2.4.5.1 Metodos predefinidos

Los métodos predefinidos son métodos que el sistema provee, pero que pueden ser sobrescritos. Si se define una implementación de un método predefinido, el sistema la acepta y descarta su implementación.

Los métodos predefinidos y su comportamiento son los siguientes.

- *implicitCast (Type):Type*
- *equalsTo(Type):boolean*

3.2.4.5.2 Metodos reservados

Se llaman de esta manera aquellos métodos que el sistema proporciona, pero que de ninguna manera pueden ser redefinidos. Hay que aceptar la implementación que el sistema proporciona. Los métodos reservados y su comportamiento son los siguientes.

- *getName():String*.
- *getType():String*
- *getTypeField(int n):Type*
- *getDataField(int n):String*
- *isReference():boolean*
- *createAlias(String):Type* no debe ser usado por el usuario.
- *isNamedType():boolean*
- *isPredefinedType(String):boolean*
- *builder (String id, TermList e):Type* no debe ser usado por el usuario, Tipos preestablecidos

Existen cuatro de clases de objetos tipo que se van a generar siempre. Estas clases son fundamentales para el funcionamiento del *framework* que conforma el comprobador de tipos que se va a generar.

A continuación hablaremos de estas tres clases y como las debe tratar el usuario.

3.2.4.6 **BaseType**

Es la clase Base de todos los objetos tipo. Es generada automáticamente por la aplicación TyCC.

Automáticamente se generan dos métodos en *BaseType*: *implicitCast()* y *equalsTo()*

- *implicitCast(Type):Type* se implementa en base a la cadena de promociones que el usuario haya indicado en la sección de declaración de tipos. Es decir garantiza una promoción implícita entre los tipos que estén en la misma cadena de promociones. Y que no la habrá entre tipos que no se encuentren en dicha cadena.
- *equalsTo(Type):boolean* implementa la equivalencia entre tipos. Se proporciona un a implementación de equivalencia nominal. Es decir que dos tipos serán iguales si se llaman exactamente igual. Aunque tengan la misma estructura, no se considerarán iguales si tienen nombre distinto.

Cualquiera de estas dos operaciones puede ser redefinida por el usuario, tanto en la clase *BaseType*, si se quiere actuar sobre el comportamiento global de todos los objetos. O ser redefinida en alguna de las clases hija, para un matiz de comportamiento más específico.

La clase *BaseType* no debe aparecer en la sección de declaración de tipos. Sin embargo se pueden definir operaciones sobre ella. Para definir operaciones sobre la clase *BaseType* lo hacemos como si se tratase de cualquier otra clase.

La clase *BaseType* tiene una característica muy interesante que proporciona al *framework* su flexibilidad y su gran comodidad de uso.

Todas las operaciones que hayan sido definidas por el usuario y que no hayan sido definidas en la clase *BaseType*, son implementadas como operaciones base en *BaseType* de forma que si son invocadas su comportamiento sea el alzamiento de una excepción indicando operación no permitida entre tipos. De esta manera si no definimos una operación para un tipo determinado, es que esta operación no puede realizarse. Al no estar definida llamará a la de la clase base que lanzará una excepción.

Esta característica ahorra un tiempo precioso de implementación de comportamiento indefinido, a la vez que se controlan también dichos comportamientos indefinidos.

3.2.4.7 TypeReference

Esta clase no es manejada directamente por el usuario, sino que es un objeto de esta clase es asignado por el *framework* un objeto tipo cuando se ha pedido un tipo que no existe.

TypeReference sirve como suplente de la clase requerida a la hora de formar parte de una estructura. Es un tipo por derecho propio. Tiene definidas las operaciones de todos los tipos como un comportamiento de error. Es decir un intento de invocar cualquier operación de tipos sobre *TypeReference* alzará una excepción indicando que no se pueden realizar operaciones sobre referencias a tipos.

Esto es así excepto los métodos *getName()* y *getType()* que devuelven el nombre que se ha querido dar a la expresión de tipo y su tipo, que en el caso de tipos con nombre coincide con la etiqueta devuelta por *getName()*.

Es posible saber si un objeto tipo es una referencia con el método preestablecido *isReference()*

Esta clase no puede aparecer en la sección de declaración de tipos ni se pueden definir operaciones sobre ella. Tampoco debe ser usada directamente, sino que debe dejarse su creación y gestión en manos del *framework*.

3.2.4.8 NamedType

Esta clase tampoco es manejada directamente por el usuario, sino que un objeto de esta clase es asignado por el *framework* un objeto tipo cuando se ha resuelto una referencia al mismo, o cuando se quiere asignar un nombre a un tipo.

NamedType es una redirección al objeto *Type* que representa. Su único cometido es darle un nombre a una expresión de tipo. Es un tipo por derecho propio. Tiene definidas las operaciones de todos los tipos como una delegación al objeto *Type* que representa. Excepto los métodos *getName()* y *getType()* que devuelven el nombre que se ha querido dar a la expresión de tipo y su tipo, que en el caso de tipos con nombre coincide con la etiqueta devuelta por *getName()*. Cualquier llamada a un método de *NamedType* es delegada en su método homólogo en el objeto *Type* correspondiente.

Es posible saber si un objeto tipo es un tipo con nombre consultando el método preestablecido *isNamedType()*

Esta clase no puede aparecer en la sección de declaración de tipos ni se pueden definir operaciones sobre ella. Tampoco debe ser usada directamente sino que debe dejarse su creación y gestión en manos del *framework*.

3.2.4.9 Type

Esta clase es el interfaz a la jerarquía de clases representantes de los tipos que ha definido el usuario. Todos los objetos tipo se manejarán a través de esta clase.

Cada objeto de la clase *Type* tiene asociado un objeto de tipo *BaseType*, que es el objeto tipo en sí. La clase *Type* tiene implementadas todas las operaciones que haya definido el usuario, y las preestablecidas. Cuando se invoca un método de comprobación de tipos en *Type*, esta clase delega su ejecución en el objeto *BaseType* correspondiente.

De esta manera tenemos un comportamiento polimórfico, usando objetos de un mismo tipo.

Esta clase no puede aparecer en la sección de declaración de tipos ni se pueden definir operaciones sobre ella. Es la única clase que el usuario debe manejar directamente. Es la representante de los objetos tipo, por lo que debe ser usada en las operaciones de comprobación de tipos como argumento de métodos, tipo devuelto, etc. Asimismo cuando se quiera crear un tipo, la tabla de tipos lo devolverá encapsulado en un objeto *Type*.

3.3 Ejemplo de uso

Un pequeño ejemplo clarificará más estos conceptos.

Se parte de un lenguaje muy sencillo con sólo 3 tipos de dato: entero, real y puntero, representados respectivamente por las clases *Integer*, *Double* y *Pointer*.

De entero y real hay una promoción implícita, del puntero a resto de tipos ninguna.

Se quiere definir una operación aritmética básica (suma y resta) entre tipos y el resto.

La aritmética es posible entre números enteros y reales, y entre enteros y punteros. El resto es posible sólo entre enteros.

El fichero de especificación sería de esta manera:

```
%{
import java.io.*;
%}
```

En el peámbulo se ha introducido código para poder usar las funciones de E/S de Java.

```
Types = { Integer < Double Pointer}
```

En la sección de declaración de tipos únicamente es necesario indicar la cadena de promociones indicando el orden de conversión con un signo '<'. Los tipos que no se promocionan es necesario indicarlos también.

```
class Integer{
  Type aritmetical(Type t){
    System.out.println("aritmetical OK: MyInteger" +t.getName()+ " -> "+
t.getName());;

    return t;
  }

  Type mod(Type t) {
    if (t.getType().equals("Integer")) return t;
    throw new TypeException("mod only available in Integers");
  }
}
```

Las operaciones aritmeticas, se definen con la operación *aritmética()*, para los enteros se devuelve simplemente el tipo del segundo operando, pues se puede realizar con cualquier tipo, y el entero será siempre el menor de los dos. El módulo sólo será posible entre enteros, de manera que se define también en los enteros y se comprueba que el segundo operando sea entero.

```
class Double{
  Type aritmética(Type t) {
    if (t.getType().equals("Pointer") )
      throw new TypeException("Can't use binOP between Double & Pointer");
    System.out.println("aritmética OK: Double" + t.getName() + " -> Double");

    return Type.typesTable.getType("Double");
  }
}
```

Para los tipos reales es necesario comprobar que el segundo operando es promocionable a real, también se puede comprobar que el segundo tipo no sea un puntero

```
class Pointer (Type){

  Type aritmética(Type t){
    if (t.getType().equals("Integer") ){
      System.out.println("aritmética OK: Pointer " + t.getName() + " -> Pointer");
      return Type.typesTable.getType("Pointer");
    }
    else throw new TypeException("Can't use aritmética operations between pointers
and"
                                + t.getName());
  }
  Type getOf() { return getTypeField(0); }
}
```

Por último para los punteros debe comprobarse que el segundo operando sea un tipo entero.

Como puede verse no se redefine el método *mod()* en los otros tipos, no es necesario. Cuando se define una operación por defecto TyCC genera en la clase base una operación idéntica que lanza una excepción indicando que dicha operación no es posible en el tipo indicado.

Si se desea ver el fichero de entrada por completo, así como la jerarquía de clases generada puede consultarse en el directorio *calc* del CDROM que acompaña la documentación del proyecto.

4 GENERADOR DE COMPROBADORES DE TIPOS: TYCC

4.1 Manual de usuario

TyCC es la aplicación que permite que a partir de la especificación completa de los tipos de un lenguaje, según se ha visto en el punto anterior, se genere una implementación a medida del usuario de un comprobador de tipos.

La aplicación TyCC es un programa de línea de comandos. En la implementación para Windows funciona desde una consola MS-DOS.

4.1.1 Invocación desde la línea de comandos

Su uso es muy simple: se escribe el nombre de la aplicación (TyCC) opcionalmente puede llevar algún parámetro de entrada seguido del nombre del fichero que contiene la especificación de tipos

```
TyCC [parámetros] <fichero de entrada>
```

Esto produce que TyCC realice un análisis sintáctico sobre el fichero de entrada, y si es correcto generará los ficheros de salida con la jerarquía de tipos especificada en el fichero de entrada. Así como el código del interfaz de acceso a los objetos tipo al usuario la clase *Type*.

En el caso de generar lenguaje Java el fichero de salida es invariablemente *Type.java*, Ya que es necesario que el fichero se llame igual que la clase pública que se define en él. Pero la primera letra en minúsculas.

En el caso de generar otro lenguaje como C++ es posible especificar otro fichero de salida.

Si invocamos TyCC sin ningún argumento nos imprimirá toda la lista de sus posibles parámetros de entrada.

```
C:> tycc [ENTER]
usage:
  TyCC [options] <input_file>

options:
  -baseError=<class_name> // name of the base class
                          // for TypeException.(Optional)
  -language=<lang_name>   // name of output language.
                          // JAVA by default. (Mandatory)
  -outHeader=<file_name>  // output header file. (Optional)
  -outImpl=<file_name>    // output implementation file.
                          // Type.Java by default (Mandatory)
  -API=<api_location>     // path of the API (Mandatory)
  -TEParser=<class_name> // name of the type expression
                          // compiler class. (Mandatory)
```

4.1.2 Parámetros de entrada

En la última versión de TyCC hay 3 parámetros posibles

-baseErrorClass=<clase base>

Se especifica la clase base de que puede heredar la clase *TypeException*, que es la clase base de la jerarquía de excepciones que utiliza el comprobador de tipos. Esto es útil si se quiere integrar el comprobador de tipos con ANTLR para habilitar la recuperación de errores ya que para hacerlo posible ANTLR necesita que todas las excepciones que se alcen deriven de *ParserException*. Así pues si se escribe

```
--baseErrorClass=ParserException.
```

TypeException derivaría ahora de *ParserException* y la integración es inmediata. Obviamente en la parte derecha de este parámetro las mayúsculas y minúsculas pueden ser importantes, pues se trata del nombre de una clase.

El uso de este parámetro implica que se va a generar una clase *TypeException* en la localización del paquete donde resida el API, por lo que se debe comprobar que se tienen los correspondientes permisos de escritura.

-language=<lenguaje destino>

Especifica el lenguaje que se desea utilizar. En esta versión sólo hay implementación de salida en Java. Ejemplos válidos de este parámetro serían.

`--language=JAVA, --lenguaje=C++, etc.`

-outHeader=<nombre de fichero>

Se especifica el nombre de un fichero de salida que contendrá las declaraciones de las clases que genere TyCC. Como sólo está implementada la generación de código en Java, este parámetro no es aplicable.

-outImpl=<nombre de fichero>

Se especifica el nombre del fichero de salida que contendrá la implementación de las clases que genere TyCC. Esta opción se anula si *language=JAVA*, ya que la única clase pública será *Type* y por tanto debe estar definida en un fichero llamado *Type.java*

-API=<path>

Se especifica el path completo donde se haya el la implementación del API.

Este parámetro es obligatorio si se usa la opción *-baseError*, ya que indica donde se debe generar el fichero *TypeException.java*

--TEParser=<nombre de clase>

Se especifica el nombre la clase que el *framework* utilizará para analizar la representación textual de las expresiones de tipo y generar a partir de ella una representación en forma de objetos. Debe ser el nombre de una clase que cumpla el interfaz *TEParserIF*. Por defecto se utiliza la clase *es.uniovi.lsi.typeChecking.TEParser* que es una implementación realizada de *TEParserIF* para el lenguaje de expresiones de tipo *TEL*.

4.2 Funciones de TYCC

TyCC analiza la entrada y construye a partir de la especificación de tipos una jerarquía de objetos que los representan y con un único interfaz que da visibilidad a esos objetos en forma de una única clase *Type*.

TyCC realiza un análisis sintáctico de la entrada, y hay situaciones en que alertará al usuario. En concreto:

- Escritura incorrecta de alguna palabra reservada. En las palabras reservadas TyCC es sensible a las mayúsculas.
- Omisión de la sección de declaración de tipos. La sección de declaración de tipos es necesaria siempre. Deben aparecer todos los tipos para los que luego se quieran definir operaciones. Aunque no es necesario definir operaciones en todos los tipos que en ella aparecen. La clase *BaseType* no debe aparecer en la sección de declaración de tipos. Aunque también se pueden definir operaciones para *BaseType*.
- Se cuida también de que las clases preestablecidas no aparezcan en la sección de declaración de tipos y que, excepto sobre *BaseType*, no se pueda definir operaciones sobre ellas. Es decir que *NamedType* *TypeReference* no aparecen ni en la sección de declaración de tipos ni se definen como clases. Son generadas automáticamente por TyCC

- Errores en la escritura de promociones. TyCC detecta incoherencias en las cadenas de promociones, así como repeticiones en la especificación de tipos.
Por ejemplo :
Types = { int < real < double double < real }
esto es absurdo y daría un error.
- Errores de sintaxis en la definición de las clases.
- Se permite la sobrecarga de métodos, pero se detecta también si se intenta definir dos veces un mismo método. El análisis de si dos métodos son iguales, se hace en función de su nombre tipo de retorno y signatura, sin tener en cuenta los nombres de los argumentos si los hubiere.
- Se cuida de que no se intenten redefinir métodos reservados. Y se realizan las operaciones necesarias para que el usuario pueda redefinir los métodos predefinidos.
- El cuerpo de los métodos y el código incrustado se copia tal cual en la salida. Por lo que no se realiza ningún análisis sobre él. Será el compilador del código destino el que se encargue de verificar su corrección.

4.3 Clases generadas por TyCC

4.3.1 Clase Type

Es el interfaz que se ofrece al usuario para acceder a la jerarquía de tipos.

Lo tradicional al utilizar una jerarquía de clases representando tipos es que todos hereden de una clase base y después utilizar composición *ver Composite* [GOF], La composición aquí es utilizada a través de un nivel extra delegación. La clase *Type* delega cada operación o en el método homologo de su atributo *type*, que es un objeto de la clase *BaseType*, de la que derivan todos los tipos definidos en el fichero de especificación.

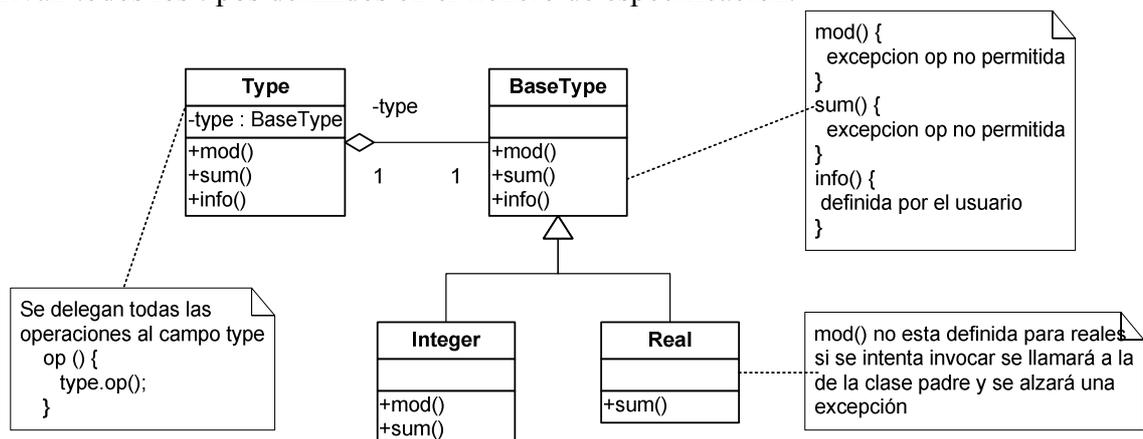


Figura 2: Diagrama de clases, que representa la delegación de operaciones de la clase *Type* en la jerarquía de tipos coronada por *BaseType*.

Esta delegación plantea unas ventajas enormes.

Primeramente el usuario sólo manejará una clase la clase *Type*, el resto de clases tienen nivel de *package*.

Al usar esta delegación (o indirección), se permite directamente el uso de alias en las expresiones de tipo (de gran utilidad en los tipos definidos por el usuario). Se permite que un

tipo tenga agregados otros tipos que aún no han sido definidos, con la colaboración de la clase *NamedType* (ver 3.2.4.8 en página 14). Esto es posible gracias a la mencionada indirección que hace la clase *Type*.

Hay que decir que esta indirección plantea una desventaja. El usuario define el código de las operaciones en el fichero de entrada. Este código se copia tal cual en las clases representantes de los tipos, que luego son direccionadas por *Type*, es por ello que cualquier referencia a la instancia de clase en el código de estos métodos será a la clase *BaseType* que realmente contiene al método, no a *Type*, que no es más que un envoltorio.

Por ello operaciones que serían muy sencillas como:

```
Class Integer {
    . . .
    Type unaryNot() {
        return this
    }
    . . .
}
```

no son posibles, pues se devolvería una referencia al objeto *Integer* no al objeto *Type* que contiene al objeto *Integer*.

Para ello hay dos soluciones:

1. Añadir a todos los objetos tipo un campo *thisType*, que referenciase a su envoltorio (a *Type*).
2. En las operaciones que haga falta una referencia al propio objeto, usar la tabla de tipos para obtener una referencia.

Se eligió la segunda opción, aunque es más tediosa y propensa a errores, pues la primera, puede plantear inconsistencias en la jerarquía de tipos al tratarse de un campo que en ocasiones se debería modificar una vez el tipo está almacenado. Por tanto no se podría suponer el almacenamiento de objetos como constantes. En algunos lenguajes como C++ tendríamos problemas de coherencia. Además el uso de *this* y *thisType*, es en opinión de los autores demasiado truculento así como poco elegante.

Adoptando entonces la segunda solución la operación anterior se definiría así:

```
Class Integer {
    . . .
    Type unaryNot(TypeTable typeTable) {
        return typeTable.getType("Integer");
    }
    . . .
}
```

La jerarquía exacta es la que se muestra en la Figura 3.

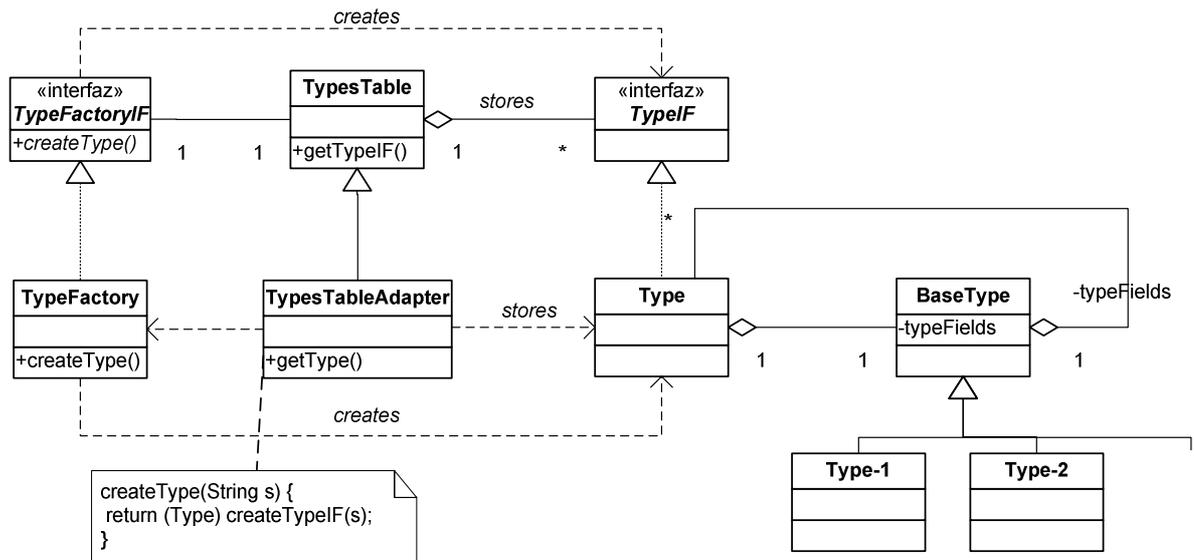


Figura 3: Diagrama de clases generado por TyCC en colaboración con el API.

Como se puede observar la clase *Type* que se va a generar implementa el interfaz *TypeIF*. Se define una clase adaptadora de la tabla de tipos por comodidad para el usuario para poder usar directamente las clases *Type*.

Asimismo, se define una clase que implementa el interfaz *TypeFactoryIF*, que servirá para poder crear los objetos *Type*. Esta clase no se usará directamente sino a través de la tabla de tipos.

En la implementación en Java, la clase *TypeFactory* es una clase anónima. Simplemente se usa para inicializar la tabla de tipos, y la clase *TypesTableAdapter* es una clase anidada dentro de la clase *Type*. Además existe una variable de clase de tipo *TypesTableAdapter*, o mejor dicho *Type.TypesTableAdapter*, en la clase *Type* que se puede usar como tabla de tipos.

En los siguientes puntos se detalla el papel de cada clase de la jerarquía.

4.3.2 Clase BaseType

Se define una clase base de la jerarquía de tipos. De esta clase colgaran como hijas todas las clases tipo que defina el usuario como parte de su lenguaje y dos clases predefinidas: *NamedType* y *TypeReference*

La clase tiene un vector de elementos de clase *Type* con lo que el composite [GOF94] es inmediato. Esta clase puede usarse activamente; definiendo operaciones, que tendrán un comportamiento similar en todos los tipos; o diferenciando comportamientos por medio de la introspección, que proporciona el sistema generado por TyCC, aunque es preferible dejar esta diferenciación en manos del polimorfismo siempre que sea posible.

Todas las operaciones que hayan sido definidas por el usuario y que no lo hayan sido

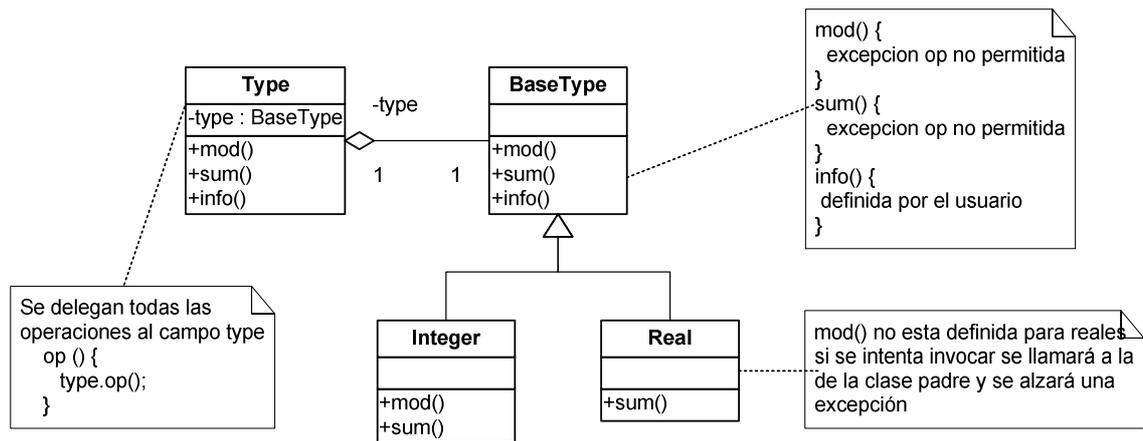


Figura 4: Diagrama de clases, que representa la acutación de los objetos de clase *BaseType*, como objetos delegados de *Type*.

en la clase *BaseType*, se generan en *BaseType* de forma que lancen una excepción indicando operación no permitida entre tipos. De esta manera si no se define una operación para un tipo determinado, es que esta operación no puede realizarse. Al no estar definida llamará a la de la clase base que lanzará una excepción.

En la Figura 4 se tiene el ejemplo de una jerarquía de dos tipos real y entero, con tres únicas operaciones.

1. *mod()* que realiza la operación resto entre dos tipos. Por tanto sólo válida entre tipos de datos que representen reales.
2. *sum()* que realiza la operación de adición entre dos tipos de datos. Válida para cualquiera de los dos tipos.
3. *info()* que devuelve información sobre los tipos de datos. Nuevamente válida para los dos tipos de datos.

Cada operación definida en la clase *Type* es delegada en la operación homóloga de su campo *type*, que es un campo de la clase *BaseType*. Un tipo *Integer*, por tanto es una clase *Type*, cuyo campo *Type* es de tipo *Integer*, de esta forma se llamarán a las operaciones de *Integer* polimórficamente.

La operación *mod()* únicamente la se define en los enteros, pues en los reales no tiene sentido. TyCC automáticamente crea un método idéntico en *BaseType*, que lanza una excepción en caso de que ser invocado. De esta forma cuando se invoque esta operación sobre un tipo no entero, al no estar redefinida en ese tipo se llamará a la operación definida en el padre que lanzará una excepción. Si el tipo resulta ser entero, el polimorfismo se encarga de que el método invocado sea el del tipo entero.

La operación *sum()* está definida en ambos tipos, de forma que cuando se invoque el polimorfismo se encarga de llamar al método adecuado.

La operación *info()*, no está definida en ninguno de los dos tipos, pero forma parte de los dos, pues se define en la clase padre. De esta forma es una operación común a toda la jerarquía de tipos. Si se desea una especialización para un tipo en particular, TyCC permite que también se redefina esta operación en alguno de los tipos derivados.

Hay que tener cuidado con estas características, pues funcionan en una sola dirección. Es decir que no son conmutativas. Por ejemplo sean dos clases *A* y *B*, y una operación *Op(Type)*

t). que sólo es posible en la clase *A*. Entonces dicha operación se define sólo en la clase *A*. Pero ¿y si el diseñador es un poco vago y la define de esta manera?

```
Type Op(Type t) {
    return typeTable.getType("A");
}
```

Al no definirse en la clase base, automáticamente TyCC genera un método que hace que lance una excepción indicando que no es posible realizar la operación *Op()* en esos tipos. Exactamente algo así:

```
Type OpType(t) {
    throw typeException("operation not allowed between " + getType());
}
```

Sean ahora dos variables *a* y *b* de tipos *A* y *B* respectivamente. Si se invoca *b.Op(a)* saltará una excepción. Perfecto, es el comportamiento esperado. Pero si se invoca *a.Op(b)*, no saltará ninguna excepción, pues se invoca la operación *Op* sobre *a*, que la tiene definida. Esto puede ser lo que se pretende o no, si la operación *Op* es conmutativa. El *framework* que proporciona TyCC no genera operaciones conmutativas. Es decir, no es lo mismo *a.op(b)*, que *b.op(a)*.

En el caso anterior si se tratase de una operación conmutativa sería necesario definir el método en *A* como sigue:

```
Type Op(Type t) {
    if (t.getType()=="B") throw new TypeException("Operación no permitida
    sobre el tipo B");
    return t;
}
```

4.3.3 Clases representantes de los tipos (Type-1, Type-2, etc.)

En este marco se encuadran las clases que representan los tipos que el usuario quiere definir para su compilador. Por tipos se entienden tipos predefinidos, constructores de tipos, tipos con nombre, etc.

Se genera una clase por cada tipo que el usuario quiera definir, todas derivadas de la clase *BaseType*.

Los constructores de tipos pueden usar el vector *typeFields* de elementos *Type* que heredan de *BaseType* para realizar la composición de tipos, y otro vector *dataFields* de elementos *String* que pueden usar para guardar datos relativos a la estructura de los tipos. También pueden tener miembros extra que el usuario quiera definir para un tipo en particular, pero que no formarán parte de la representación del tipo.

4.3.4 Clase Type Reference

Como ya se ha visto es posible manejar tipos que aún no hayan sido definidos, esto se hace usando la clase *TypeReference*. Esta clase actúa como un tipo más, pero cualquier intento de invocar un método de tipo hace que lance una excepción que indica la imposibilidad de realizar operaciones sobre referencias a tipos.

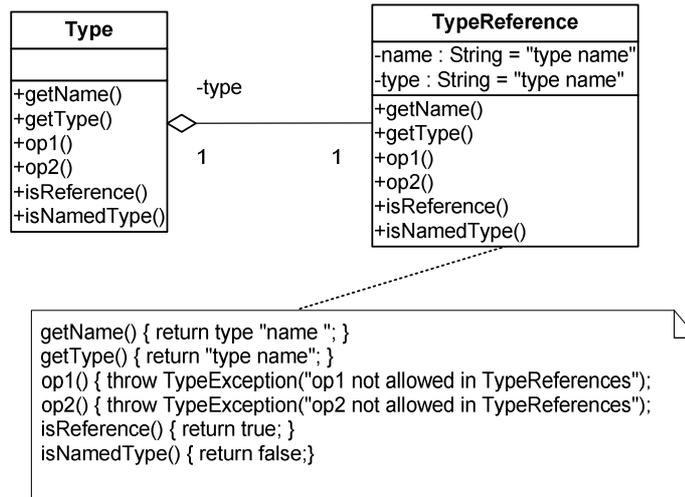


Figura 5: Diagrama de clases que representa la referencias a tipos.

Los únicos métodos que tienen sentido en este clase son los métodos *getType()* y *getName()*, que devuelven el nombre del tipo al que referencia. En este caso *getType()* y *getName()* devuelven exactamente el mismo nombre, pues los nombres de expresiones de tipo no tienen argumentos de construcción, y se ha tomado el convenio de que *getType()* devuelve la clase a la que pertenece la expresión de tipo. Se entiende que esta clase será el nombre que le quiere dar el usuario.

4.3.5 Clase NamedType

Una vez que una referencia a un tipo ha sido resuelta, se crea una instancia de la clase *NamedType* que representa el nombre de la expresión de tipo que referenciábamos. La clase *NamedType* tiene un comportamiento similar a la clase *Type* en el sentido de que delega su comportamiento a la clase *Type* a la que se refiere.

Este nivel de indirección es necesario para garantizar la unicidad de los objetos representantes de las expresiones de tipo.

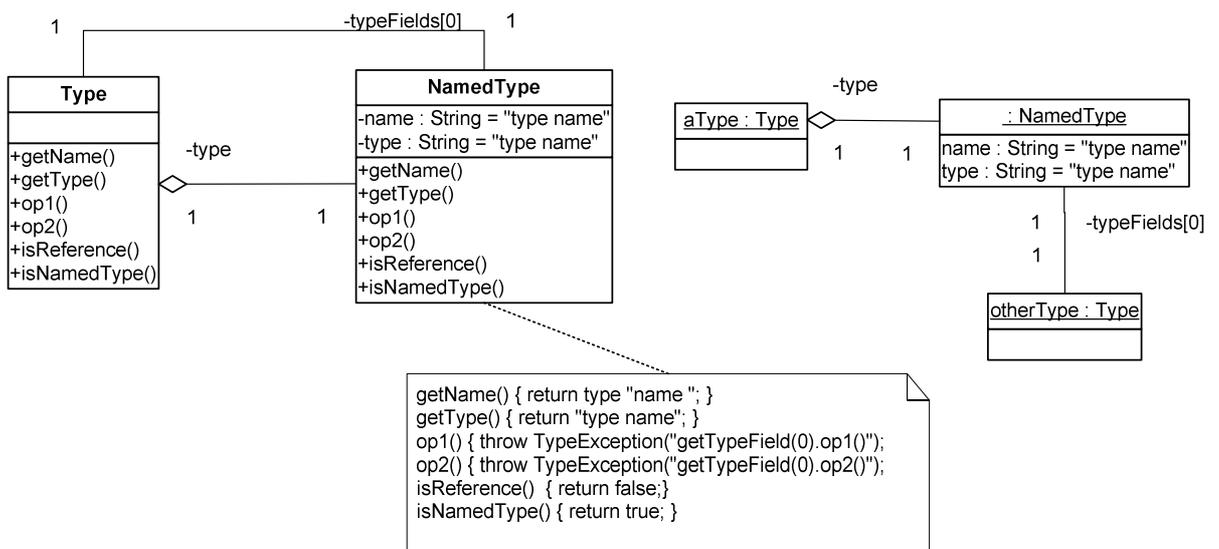


Figura 6: Diagrama de clases que representa los tipos con nombre.

Cada objeto de la clase *Type* tiene un miembro de tipo *BaseType* que es el objeto para el cual hace de *proxy*. Pero se ha dicho que la visión que se da al usuario es la de que sólo existe una clase *Type*, por tanto si se quiere dar un nombre a una expresión de tipo definida por el usuario ésta será de clase *Type*, por lo que no podemos usar un objeto de clase *Type* para envolverle, pues *Type* y *BaseType* no están en el mismo árbol de herencia.

Tampoco es buena idea crear una instancia de *Type* y asignarle la expresión de tipo que indirecciona el otro objeto, pues entonces la nueva expresión de tipo no sería un sinónimo exacto.

Así pues la solución es crear una nueva clase predefinida derivada de *BaseType* que sirva de adaptador entre las dos clases *Type*, la que da el nombre y la expresión de tipo a la que se quiere dar un alias: la clase *NamedType*.

El uso de *NamedType* y *TypeReference* con la clase *Type* como manejador sugiere un comportamiento muy parecido al del patrón *DynamicBinding* [Grand99],

5 INTERFAZ DE PROCEDIMIENTOS PARA LA GESTIÓN DE TIPO

La otra parte fundamental del sistema es el API de gestión de objetos tipo.

Se trata de un interfaz en forma de llamadas a métodos de objetos que gestionan la jerarquía de tipos.

Mediante este interfaz el usuario tendrá acceso a la tabla de tipos y podrá pedir objetos tipo. No se tiene que preocupar ni de su creación ni de su eliminación. Se dividirá esta explicación de la utilización del API en dos partes un uso básico, para usuarios que hacen uso de el de forma transparente a través de la aplicación TyCC, y un uso avanzado para los usuarios que quieren utilizarla como otro componente más de su diseño.

5.1 Manual básico de usuario

TyS proporciona al programador un interfaz de una tabla de tipos que puede gestionar cualquier objeto que cumpla el interfaz *TypeIF*.

5.1.1 Gestión de tipos: `getTypeIF(String): TypeIF`

Es el método principal de esta tabla de tipos. Recibe como parámetro una cadena de caracteres con la traducción de la expresión de tipo que representa la estructura del tipo a construir a un lenguaje para representar expresiones de tipo.

Lo normal es adaptar este método a la implementación de *TypeIF* que se vaya a utilizar. Por ejemplo TyCC genera una clase derivada de la tabla de tipos que adapta el interfaz de este método de la siguiente manera.

```
Type getType(String s) {
    return (Type) getTypeIF(s);
}
```

El *downcasting* es seguro pues la tabla de tipos almacena objetos de tipo *Type* que siguen el interfaz *TypeIF*.

5.1.2 Tipos con nombre: `assignName(TypeIF t, String typeName): TypeIF`

Este método asigna el nombre `typeName` a al tipo `t`, crea un tipo con nombre y lo retorna. Si se desea asignar un nombre a un tipo se debe utilizar este método no se debe asignar directamente.

Este método se apoya en el método `createAlias()` del interfaz `TypeIF`.

Además si el nombre que se quiere asignar antes era un tipo desconocido, en el sentido de que era un nombre de un tipo que antes se había requerido y no se había encontrado, por lo que se había creado una referencia, se considera que la referencia está resuelta y se llama al método `resolveReference()` del interfaz `TypeIF`, que se encargará de resolver las referencias como se estime más oportuno. También se borra el tipo de la lista de tipos desconocidos.

5.1.3 Tratamiento de los tipos desconocidos: `getUnknownTypes():ArrayList`

Cuando se pide un tipo, mediante el método `getTypeIF()` y este no está en la tabla de tipos, se intentará construir. Si este tipo o alguno de sus tipos componentes no se ha definido todavía durante el proceso de compilación, en lugar de dar un error la tabla de tipos lo trata como un tipo desconocido. Esto se averigua porque se espera que el método `builder()` de la tabla de tipos, considere a este tipo una referencia. Es decir que el método `isReference()` del tipo recién creado devuelva `true`. Todos los tipos desconocidos se van anotando en una lista. Y el usuario puede consultarla en cualquier momento llamando al método `getUnknownTypes()`.

Cuando las referencias se van resolviendo estos tipos son borrados de la lista.

5.1.4 Consultor de tipos: `isType(String s):boolean`

Para saber si una cadena de caracteres se corresponde con algún tipo se puede usar este método consultor: `isType()` devuelve `true` si `s` es el nombre de algún tipo contenido en la tabla de tipos y `false` en caso contrario.

5.1.5 TEL: Type Expression Lenguaje

En este API se ha definido un lenguaje para expresiones de tipo: TEL (*Type Expression Language*) basado en una notación funcional.

TEL representa a los tipos con sus componentes entre paréntesis. Estos componentes, pueden ser a su vez datos entre comillas u otros tipos.

Es decir.

Tipo -> *identificador* seguido opcionalmente de `(' <lista de componentes separados por comas> ')`

Componente -> *Tipo* o *dato* entre comillas o `(' <lista_unica_componentes separados por comas> ')`

Si en la especificación de un tipo se ha indicado que no tiene componentes no es necesario poner los paréntesis.

Los datos entre comillas sirven para representar datos que forman parte de la estructura del tipo. pero que no son tipos.

Las listas únicas de componentes son o bien listas de tipos separados por comas o listas de cadenas de caracteres separados por coma. Se usarán para representar expresiones de tipo que

pueden llevar un número indefinido de argumentos de estilo tipo o de estilo cadena de caracteres.

La gramática que define el lenguaje TEL es la siguiente:

$\langle \text{TypeExpression} \rangle ::= \text{IDENTIFIER} \langle \text{OptionalSignature} \rangle$

$\langle \text{OptionalSignature} \rangle ::= \langle \text{Signature} \rangle$
| λ

$\langle \text{Signature} \rangle ::= \text{'('} \langle \text{ParameterList} \rangle \text{'}'$

$\langle \text{ParameterList} \rangle ::= \langle \text{Parameter} \rangle \langle \text{MoreParameterList} \rangle$

$\langle \text{MoreParameterList} \rangle ::= \text{' ; ' } \langle \text{Parameter} \rangle \langle \text{MoreParameterList} \rangle$
| λ

$\langle \text{Parameter} \rangle ::= \langle \text{TypeExpression} \rangle$
| *STRING*
| $\text{'('} \langle \text{TupleList} \rangle \text{'}'$

$\langle \text{TupleList} \rangle ::= \text{TypeList}$
| *StringList*

TypeList ::= *TypeExpression*
| *TypeExpression* ' ; ' *TypeList*

StringList ::= *STRING*
| *STRING* ' ; ' *StringList*

Ejemplos:

Se quiere representar:

Puntero a entero : `Puntero(Entero)`

Entero: `Entero`

Array de cuatro dimensiones de Array de tres dimensiones de Char :

`Array(Array(Char, "3"), "4")`

Registro { Entero x; Real y; }. Hay varias posibilidades, representarlo directamente:

`Registro((Entero, Real), ("X", "Y"))`

Como en una clase registro a priori no se sabe cuantos campos se van a definir es necesario aquí utilizar las lista de componentes únicos.

Otra posibilidad que puede utilizar el programador para usar una clase registro es usar una clase tipo intermedia.

Por ejemplo:

`Campo(Nombre, Tipo)`

El ejemplo anterior quedaría

`Registro ((Campo("X", Entero), Campo("Y", Real)))`

Por último la forma de representar una función podría ser

Funcion (Entero, Real, Puntero a char)→ Real

Funcion ((Entero, Real, Puntero(Char)), Real)

En este lenguaje se puede representar cualquier expresión de tipo. La máxima expresividad se lograría permitiendo que las listas no fueran de componentes únicos, sino que se permitiese cualquier permutación. Pero la mejora en la expresividad es mínima, se puede suplir de cualquiera de las dos forma indicadas en el ejemplo anterior de la definición de un registro, y a la hora de organizar los datos resulta más sencillo de esta manera.

De todas formas hay que recordar que el lenguaje de expresiones de tipo es configurable. El usuario es libre de utilizar el que más le convenga.

5.1.6 Manejador de errores

Basado en excepciones. Si ocurre cualquier comportamiento inadecuada se alza una excepcion de tipo *TypeException*.

Se puede hacer que *TypeException* derive de otra clase base que no se *Exception*, esto es útil por ejemplo para integrarlo con ANTLR.

Las excepciones se aíslan en un paquete a parte *es.uniovi.lsi.typeChecking.errorHandling* en el que se define la clase *TypeException* como derivada de *exception*. Este paquete será sobrescrito por el usuario haciendo que *TypeException* derive de la clase que desee.

La opción `--baseError` de TyCC realiza esta acción automáticamente.

5.2 Manual avanzado del API

A continuación se muestra aquellas aspectos del API que sólo interesan a los programadores que van a usarla como un componente aislado de la aplicación TyCC.

Al programador se le proporciona un tabla de tipos completa. Con la funcionalidad necesaria para crear, y gestionar los objetos tipo. En concreto gestionará tipos que siguen el interfaz *TypeIF*.

El programador tiene acceso directo únicamente a esta clase de tipos genérica por lo que se procede a detallar su interfaz.

5.2.1 Inicialización de la tabla de tipos

Es necesario indicar:

- El lenguaje de expresiones de tipo que se va a utilizar. Ello se hace pasándole al constructor una *abstract factory* con el compilador adecuado. Debe ser un compilador que cumpla el interfaz *TEParserIF*, definido en el API. Por ejemplo con TyCC se usa el lenguaje para expresiones de tipo TEL.
- Implementación *TermList* que se quiere utilizar para albergar los parámetros de construcción. Ello se hace pasándole al constructor una *abstract factory* con el compilador adecuado. Esta característica se ha dejado aparte del diseño del API, por considerarla únicamente para usos muy avanzados del API. Pero se anota como una posible ampliación.

- La clase de objetos que implemente *TypeIF* que se va a utilizar. Nuevamente con un *abstract factory*.

Todo ello se puede realizar en el constructor de la tabla de tipos

5.2.2 Construcción de tipos: `builder(String id, String et, TermList e):TypeIF`

builder(String

Cuando un tipo no se encuentra en la tabla de tipos en algún momento es necesario construirlo. El método *builder()* se encarga de crear el tipo cuyo nombre es *id*, cuya expresión de tipo viene determinada por la cadena de caracteres *et*. Y cuyos argumentos de construcción están en la lista de términos *e*.

La lista de términos simplemente debe seguir el interfaz *TermList*, definido en el API. Y el lenguaje en que está escrito la expresión de tipo recordemos que esta configurado de antemano.

Para la construcción del tipo se delega en el método *createType()* del interfaz *TypeFactoryIF()* que es un *abstract factory* para *TypeIF*. Y después si el tipo es una referencia se añade a la lista de tipos desconocidos. En cualquier caso el nuevo tipo es retornado al usuario.

5.2.3 Representación simbólica de las expresiones de tipo

Las expresiones de tipo se representan en forma de cadena de caracteres. Para ello es necesario definir un lenguaje que las refleje inequívocamente.

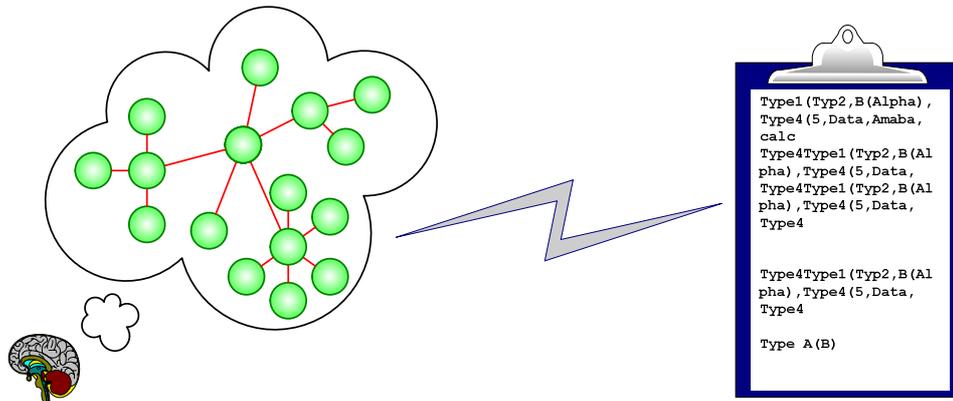


Figura 7: Representación simbólica de las expresiones de tipo.

Se define un interfaz *TParserIF*, con un único método *parse()*, que analiza una expresión de tipo en forma de cadena de caracteres en un lenguaje de representación determinado y colaborando con la tabla de tipos construye un objeto o grupo de objetos que represente ese tipo.

A la hora de inicializar la tabla de tipos lo único que se debe hacer es indicar el compilador que implemente el interfaz.

5.2.4 Representación de las expresiones de tipo mediante objetos (Avanzada)

Para que el *framework* pueda gestionar los tipos es necesario mapear las representaciones simbólicas de las expresiones de tipo en objetos.

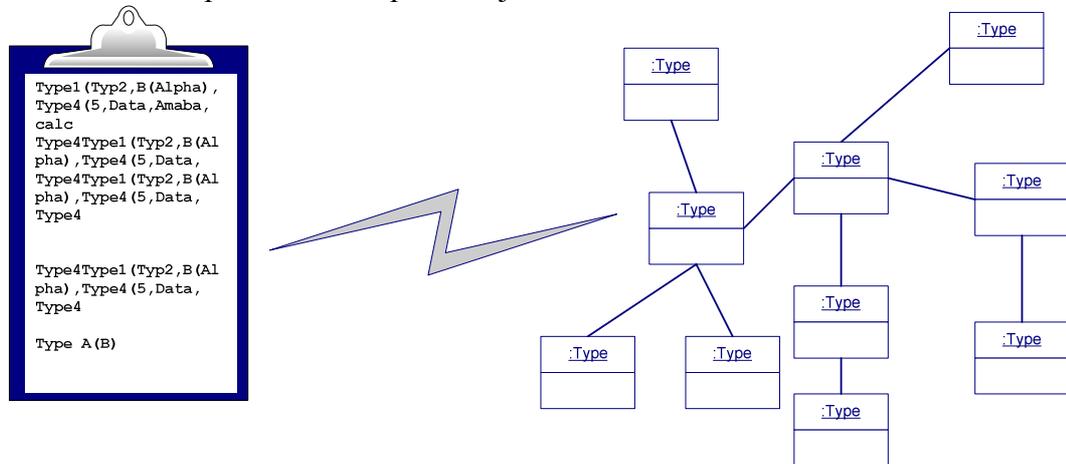


Figura 8: Representación de expresiones de tipo mediante objetos.

En definitiva se tiene que conseguir pasar de una representación textual de la expresión de tipo a un grupo de objetos conectados que reflejen esa expresión de tipo, pero con la que ya se puede operar.

El encargado de realizar esto es un objeto que cumpla el interfaz *TermListIF*. Este objeto es inicializado por un compilador que cumpla el interfaz *TEParserIF*. Y contiene las componentes, ya sean tipos o datos de un tipo que se va a construir. En este API se ha definido una implementación particular de *TermList* que se apoya en la clase *BaseType* definida por TyCC.

Su método *fillFields()* simplemente copia todos los componentes en los vectores de objetos *typeFields* y *dataFields* de la clase *BaseType* de manera consecutiva atendiendo a como se han definido en la especificación de tipos.

Vamos a poner algunos ejemplos que lo clarifican.

```
class Ej(Type1, String2, Type3, Type4+, Type5, String6+, String7)
```

Después de la construcción nuestra implementación de *TypeTermListIF* el contenido de los atributos es el siguiente:

```
typeFields[0] = Type1
typeFields[1] = Type2
typeFields[2..n+1] = type4..type4n
typeFields[n+2] = type5
dataFields[0] = String2
dataFields[1..m] = String61..String6m
dataFields[m+1] = String 7
```

5.2.5 Creación de tipos

Se implementa en el mismo *package* donde se define la clase *TermList* y el interfaz *Term*, cuatro implementaciones del interfaz *Term*, para el chequeado de la construcción de tipos: *TypeTerm*, *TypeTermList*, *StringTerm* y *StringTermList*.

Estas clases también juegan el papel de gestoras de representaciones de expresiones de tipo, como se mostró en el punto anterior.

Estas clases se integran perfectamente con la clase *Type*, generada por la aplicación *TyCC*, y sirven para chequear tipos cuyas componentes pueden estar basadas en otros tipos y datos de tipo *String*.

A continuación se detalla lo que realizan los métodos *check()* de cada clase.

- *TypeTerm* comprueba que la componente *i*-ésima del tipo que se va a construir sea un tipo.
- *StringTerm* comprueba que la componente *i*-ésima del tipo que se va a construir sea un dato.
- *TypeTermList* es una lista de objetos de tipo *TypeTerm*, para cada uno de sus elementos se comprueba que la componente *i*-ésima del tipo que se va a construir sea un tipo.
- *StringTermList* es una lista de objetos de tipo *StringTerm*, para cada uno de sus elementos se comprueba que la componente *i*-ésima del tipo que se va a construir sea un *String*.

6 CONSTRUCCIÓN DE UN COMPILADOR USANDO EL SISTEMA TYS

En este punto se detalla la manera en que se puede utilizar el *framework* TyS para la construcción de un compilador. Se pretende explicar de manera pragmática el resto de partes de la arquitectura del sistema. Para ello se detalla cada una de las fases de la construcción de un compilador haciéndolas coincidir en su momento con cada una de las componentes del sistema.

En concreto se construirá un intérprete con un diseño orientado a objetos avanzado con comprobación de tipos en tiempo de ejecución. Se mostrará paso a paso el proceso de construcción con el sistema TyS de manera que quedará patente su utilidad.

En el intérprete construido se conseguirá:

- a) Aislamiento entre la comprobación de tipos y el resto de componentes del intérprete.
- b) Utilización de la jerarquía de tipos en la interpretación para la optimización de operaciones de interpretado.
- c) Rápido desarrollo gracias a que una parte fundamental del mismo se hace utilizando el sistema TyS.
- d) Adaptabilidad a los cambios eficaz y rápida debido al aislamiento comentado en (a) y a la rapidez de desarrollo comentada en (b). Esto se ilustrará añadiendo funcionalidad al intérprete una vez terminado.

6.1 Código fuente del compilador (Compiler Source Code)

A la hora de construir un compilador usando el sistema TyS. Se puede separar de manera efectiva el análisis semántico del resto de fases de análisis.

Se puede utilizar nada más que el API de TyS como ayuda para realizar un comprobador de tipos e implementarlo manualmente, pero esto será sólo para usuarios muy específicos. En ese caso el usuario debe implementar una serie de interfaces para poder usar el API efectivamente. En concreto es necesario implementar el interfaz *TypeIF*, el interfaz *TEParserIF*, y *TypeFactoryIF* e *TEParserFactoryIF*. Haciendo esto se puede usar ya la tabla de tipos normalmente y se tienen mecanismos para evitar la duplicidad de instancias de un mismo objeto tipo, la construcción y chequeo de tipos complejos y la representación de expresiones de tipo por medio de un lenguaje.

Lo normal será utilizar la aplicación TyCC que mediante una especificación de tipos contenida en un fichero de entrada genera un comprobador de tipos, que utiliza automáticamente los métodos del API. De esta forma el uso del API queda oculto por la jerarquía de tipos generadas por TyCC, y la implementación que hace TyCC del interfaz *TypeIF*.

Según los parámetros de entrada de la aplicación TyCC se genera una implementación de *TypeIF*, *TypeFactoryIF*, *TEParserFactoryIF*, y se indica la implementación de *TEParserIF* que se va a utilizar.

Se va a construir un intérprete para un lenguaje inventado. El lenguaje *Frog*.

Frog contará con instrucciones de flujo simple *if-then-else* y *while-do*, y con tipos de datos numéricos. El lenguaje admite variables todas ellas numéricas de cinco tipos distintos. En él, se pueden realizar una serie de operaciones aritméticas, definir variables e interacción con la consola.

Como en cualquier compilador que se precie primero se define el léxico y la sintaxis del lenguaje que reconoce.

6.1.1 Módulo Léxico

Los lexemas admitidos por el lenguaje *Frog* son los siguientes:

ID ::= (A..Z | a..z) +

CONST_INT ::= (0..9) +

CONST_REAL ::= CONST_INT '.' CONST_INT +

Operadores

- *Identificadores* cualquier secuencia de letras del alfabeto inglés (no sirve la ñ).
- *Constantes enteras* cualquier secuencia de dígitos. En el léxico suele ser buena idea no considerar el signo de los números.
- *Constantes reales* números reales sin signo. Secuencias de dígitos con un punto por el medio.
- *Operadores* los siguientes caracteres o grupos de caracteres: ! + - / * % > < = ; , . == () && || >= <= != ==
- *Palabras reservadas*: if then else while do write read short int long real double

Con todo esto se construye un módulo léxico que a partir de un fichero de entrada va devolviendo una secuencia de tokens al módulo sintáctico.

6.1.1.1 Definición de tokens

Primeramente se definen unas constantes de clase para identificar los tokens que señalan el fin de fichero, los identificadores, los reales y los enteros y el resto de operadores.

A continuación se muestra una tabla con el mapeo de tokens con las componentes léxicas.

<i>Lexema</i>	<i>Token</i>
Fin de fichero	EOF
Identificador	IDENT
Constante entera	CONST_INT
Constante real	CONST_REAL
!	BNOT
=	ASSIGN
:	COLON
;	SEMI
,	COMMA
==	EQUALS
(LPAREN
)	RPAREN
!=	NOT_EQUALS
<	LT
<=	LTE
>	GT
>=	GTE
+	PLUS
-	MINUS
*	TIMES
/	DIV
{	LBRACE
}	RBRACE
	OR
&	AND
%	MOD

Lo normal sería implementarlas como variables de clase dentro del mismo *scanner*. O mejor, como hace ANTLR, definir un interfaz que contenga estas variables de clase, y hacer que la clase *Scanner*, la implemente. Esta última solución tiene la ventaja de que así otras clases tienen acceso a estas variables de clase simplemente implementando el interfaz.

6.1.1.2 Métodos públicos del Scanner

En el caso de implementar el *Scanner* automáticamente lo normal es diseñar un método *yylex()* que lea un token del fichero de entrada y devuelve un entero que equivale al *token* que se acaba de leer según el mapeo anterior.

También otro método *getLastToken()* devuelve el último token que se ha leído y no se avanza en la lectura del fichero.

Por último se deben proporcionar los métodos consultores necesarios para acceder al valor del lexema leído. Es decir en caso de un identificador el valor de ese identificador, etc.

6.1.1.3 Generación automática de un scanner

Usando herramientas de generación de analizadores léxicos se puede soslayar la implementación del *scanner* teniendo únicamente en cuenta los puntos anteriores. Como el objetivo de este ejemplo no es mostrar la construcción de un analizador léxico se utilizará la herramienta ANTLR que permite generar un scanner de forma automática a partir de la especificación de su gramática. Para ver el listado de la gramática léxica ANTLR fichero *Frog.g* en el directori *Frog* del CDROM.

6.1.2 Módulo Sintáctico

Para construir un módulo sintáctico se precisa tener en cuenta la estructura que seguirán las sentencias del lenguaje el tipo de expresiones que reconoce así como los operadores que se pueden aplicar a dichas expresiones.

Una vez tenido todo esto bien claro, se puede diseñar la gramática del lenguaje, y a partir de ahí la construcción del módulo sintáctico es automática.

Se empieza por la descripción de los operadores del lenguaje, a partir de ahí será más fácil comprender las expresiones y las sentencias para finalmente diseñar una gramática no ambigua que represente perfectamente el lenguaje.

6.1.2.1 Operadores del lenguaje

Los operadores de las expresiones de cualquier lenguaje poseen tres características fundamentales .

Precedencia Cuando en una secuencia se encuentran distintos operadores el orden de evaluación se resuelve atendiendo a la precedencia relativa que tengan entre sí los operadores. Se dice que un operador tiene precedencia más alta que otro, si a la hora de evaluarlos en una secuencia donde aparezcan los dos operadores se debe evaluar primero el que tiene la precedencia más alta. En el caso de que dos operadores tengan la misma precedencia se evaluará dependiendo de las características del lenguaje.

Asociatividad Forma de agrupar los operandos para su evaluación sobre los que actúa un operador, en caso de haber una secuencia en la que interviene más de una vez un mismo operador. Puede ser a izquierdas, a derechas, o no tener asociatividad, en este caso no se pueden dar secuencias en las que intervenga más de una vez un mismo operador

Aridad Es el número de operandos que admite el operador. Puede ser fijo, como el caso de las operaciones aritméticas o variable, como en el caso de las funciones.

A continuación se detalla el uso sintáctico de cada operador:

Operador	Nombre	Descripción
,	Secuencia	Secuencia de expresiones,
=	Asignación	Asignación, su resultado no es una expresión por lo que no se puede usar en otras expresiones ni en conjunción con otros operadores

&	And lógico	Realiza la operación <i>and</i> lógica sobre dos expresiones. Su resultado es otra expresión.
	Or lógico	Realiza la operación <i>or</i> lógica sobre dos expresiones. Su resultado es otra expresión
!	Not lógico	Realiza la operación <i>not</i> o negación lógica sobre una expresión Su resultado es otra expresión
<	Mayor	Realiza la operación de comparación “menor que” entre dos expresiones. Su resultado será una expresión indicando este resultado.
>	Menor	Realiza la operación de comparación “mayor que” entre dos expresiones. Su resultado será una expresión indicando este resultado.
<=	Menor o igual	Realiza la operación de comparación “menor o igual que” entre dos expresiones. Su resultado será una expresión indicando este resultado.
>=	Mayor o igual	Realiza la operación de comparación “mayor o igual que” entre dos expresiones. Su resultado será una expresión indicando este resultado.
==	Igual	Realiza la operación de comparación de igualdad entre dos expresiones. Su resultado será una expresión indicando este resultado.
!=	Distinto	Realiza la operación de comparación que indica si dos expresiones son distintas. Su resultado será una expresión indicando este resultado.
+	Suma	Realiza la suma entre dos expresiones devolviendo la expresión resultante.
-	Resta	Realiza la resta entre dos expresiones devolviendo la expresión resultante.
+	Mas	No tiene efecto. Devuelve la misma expresión.
-	Menos	Cambia el signo a una expresión. Y devuelve la expresión resultante.
/	Division	Realiza la división entre dos expresiones devolviendo la expresión resultante.
*	Producto	Realiza el producto entre dos expresiones devolviendo la expresión resultante.
%	Modulo	Realiza el módulo (resto) entre dos expresiones devolviendo la expresión resultante.
()	Agrupado	Agrupar expresiones, para interferir en el orden de evaluación. El resultado de la agrupación es la expresión que se evalúa entre los dos paréntesis
(Type)	Casting	Cambia el tipo de una expresión, su resultado es una expresión con el tipo resultante.

A continuación se muestra la tabla de precedencias y asociatividades. También se indica la ubicación del operador, informando si es infija, prefija, postfija, o si lleva una ubicación especial.

Los lugares más altos de la tabla muestran las precedencias más pequeñas, es decir a según se va bajando en la tabla va creciendo la precedencia de los operadores.

Operador	Aridad	Asociatividad	Ubicación
,	2	Izquierda	Infija

=	2	-	Infija
&	2	Izquierda	Infija
= ==	2	Izquierda	Infija
< > <= >=	2	Izquierda	Infija
+ -	2	Izquierda	Infija
/ * %	2	Izquierda	Infija
+ -	1	Izquierda	Prefija
!	1	Izquierda	Prefija
() casting	2	Izquierda	Encerrando la expresión de tipo y delante de la expresión a ahorrar
() agrupado	1	-	Encerrando las expresiones

6.1.2.2 Sentencias

A continuación se describe la sintaxis de las sentencias y su cometido

Las sentencias pueden ser: declaración de variables, asignación, escritura por pantalla, lectura de datos por teclado y control de flujo *if-then-else* y *while-do*, y la sentencia compuesta o bloque de sentencias

```

<sentencia> ::= <Definición_de_Variables> ';'
              | <Asignación> ';'
              | <Escritura_pantalla> ';'
              | <lectur_teclado> ';'
              | <sentencia_if>
              | <sentencia_while>
              | <bloque_sentencias>

```

Definición de variables

Se asigna a un tipo un identificador.

```
<Definición_de_Variables> ::= <Tipo> ID
```

El tipo de una variable a su vez puede ser cualquiera de los tipos predefinidos que soporta el lenguaje.

```

<Tipo> ::= SHORT
          | INT
          | LONG
          | FLOAT
          | DOUBLE

```

Asignación

Se asigna el valor de una expresión a una variable.

```
<Asignación> ::= ID = <expresión>
```

Escritura por pantalla

Se escriben por pantalla una lista de expresiones.

```

<Escritura_pantalla> ::= WRITE <lista_expresiones>
<lista_expresiones> ::= <expresión> ','
                       | <expresión>

```

Lectura de datos de teclado

Se leen por pantalla una serie de datos que son asignados a una lista de variables.

```
<Lectura_teclado> ::= READ <lista_variables>  
<Lista_variables> ::= IDENT ' , ' <Lista_variables>  
| IDENT
```

Control de flujo

Para el control de flujo tenemos dos tipos de sentencias condicionales.

La sentencia *if-then-else* y el bucle condicional *while-do*.

Sentencia if

Si la condición es cierta se ejecuta la sentencia que sigue a la parte *then* si es falsa y existe una parte *else* se ejecuta la sentencia que sigue al *else*. En caso de haber secuencias de sentencias *if else* emparejadas, el *else* se asocia al *if* más cercano.

```
<Sentencia_if> ::= IF <expresión> THEN <sentencia> ELSE <sentencia>  
| IF <expresión> THEN <sentencia>
```

Sentencia While

Mientras la guarda del bucle sea cierta se ejecuta la sentencia que sigue a la parte *do*.

```
<Sentencia_while> ::= WHILE <expresión > DO <sentencia>
```

Sentencia multiple. Bloques de sentencias

Es una lista de sentencias que puede ser vacía delimitadas por llaves.

```
<bloque_sentencias> ::= '{ ' '  
| '{ <lista_sentencias> }'  
<lista_sentencias> ::= <sentencia> <lista_sentencias>  
| <sentencia>
```

6.1.2.3 Estructura del programa

Ahora que se ha explicado en que consisten las sentencias, un programa no es más que una bloque de sentencias.

```
<Programa> ::= <bloque_sentencias>
```

Es decir se admite como programa válido el programa vacío. Y se pueden declarar variables en cualquier punto del mismo.

6.1.2.4 Expresiones

Antes de diseñar completamente la gramática queda ver la sintaxis de las expresiones válidas del lenguaje.

Esta esquema que se muestra a continuación se debe estudiar teniendo en cuenta la tabla de precedencias y asociatividades de los operadores

```
<expresión> ::= expresión <operador_binario> expresión  
| <operador_unario> expresión  
| '( <Type> ' ) expression  
| '( <expresión> ' )  
| CONST_INT  
| CONST_REAL  
| IDENT
```

```

<operador_binario> ::= '+'
                    | '-'
                    | '*'
                    | '/'
                    | '%'
                    | '<'
                    | '>'
                    | '>='
                    | '<='
                    | '=='
                    | '!='
                    | '&'
                    | '|'

```

```

<operador_unario> ::= '+'
                  | '-'
                  | '!'

```

6.1.2.5 Gramática

Ahora ya se está en condiciones de definir una gramática que represente la sintaxis completa del lenguaje en notación EBNF [EBNF96]. El diseño del módulo sintáctico se realizará bien manualmente o con una herramienta de análisis descendente, por lo que es necesario definir una gramática LL(K) cuando más pequeño sea este K más fácil será de analizar.

```

<program> ::= <block>
           ;

```

```

<block> ::= '{' ( <stament> )* '}'
         ;

```

```

<stament> : <varDefinition> ';'
          | <asignation> ';'
          | "write" <expression> (',' <expression> )* ';'
          | "read" IDENT (',' IDENT)* ';'
          | "if" <expression> "then" <stament> ("else" <stament>)?
          | "while" <expression> "do" <stament>
          | <block>
          ;

```

```

<varDefinition> ::= <type> IDENT
                ;

```

```

<type> ::= "short"
         | "int"
         | "long"
         | "real"
         | "double"

```

```

;

<asignation> ::= IDENT '=' <expression>
;

<expression> ::= <equalityExpression> ( ('&' | '|') <equalityExpression> )*
;

<equalityExpression> ::= <relationalExpression> ( ("!=" | "==" ) <relationalExpression>
)*
;

<relationalExpression> ::= <additiveExpression> ( ("<" | ">" | "<=" | ">=" )
<additiveExpression> )*
;

<additiveExpression> ::= <multiplicativeExpression> ( ("+" | "-")
<multiplicativeExpression> )*
;

<multiplicativeExpression> ::= <unaryExpression> ( ("*" | "/" | "%") <unaryExpression> )*
;

<unaryExpression> ::= '-' | '+' <unaryExpression>
| <unaryExpressionNotPlusMinus>
;

<unaryExpressionNotPlusMinus> ::= '!' <unaryExpressionNotPlusMinus>
| <castExpression>
;

castExpression ::= '(' <type> ')' <castExpression>
| <primaryExpression>
;

<primaryExpression> ::= IDENT
| CONST_INT
| CONST_REAL
| '(' <expression> ')'
;

```

La gramática resultante es una gramática LL(1) , lo que quiere decir que se puede realizar un análisis descendente con un solo *token* de *lookahead*..

6.1.2.6 Diseño del módulo sintáctico

A la hora de construir un analizador sintáctico de forma manual lo recomendable es transformar la gramática del lenguaje a una gramática LL(K) a ser posible LL(1), que son las más posibles de analizar.

Después se sigue un algoritmo muy sencillo, se define una función por cada regla, cada no terminal se transforma en una llamada a función y cada terminal en una llamada a la función *yylex()* del módulo léxico.

En el caso de hacer un analizador sintáctico orientado a objetos se define una clase *Parser* con un método por cada regla. De hecho usando esta misma gramática ANTLR generaría un *parser* similar.

A la hora de ir analizando un programa sintácticamente, la fase semántica se puede realizar al mismo tiempo, o bien construir un árbol sintáctico e donde cada uno de sus nodos no hoja represente un no terminal de la gramática y sus hojas los símbolos terminales.

Por ejemplo sea el programa válido:

```

{
  if x >0 then write 1;
  else write 2;
}

```

Su árbol sintáctico asociado se puede ver en la Figura 9.

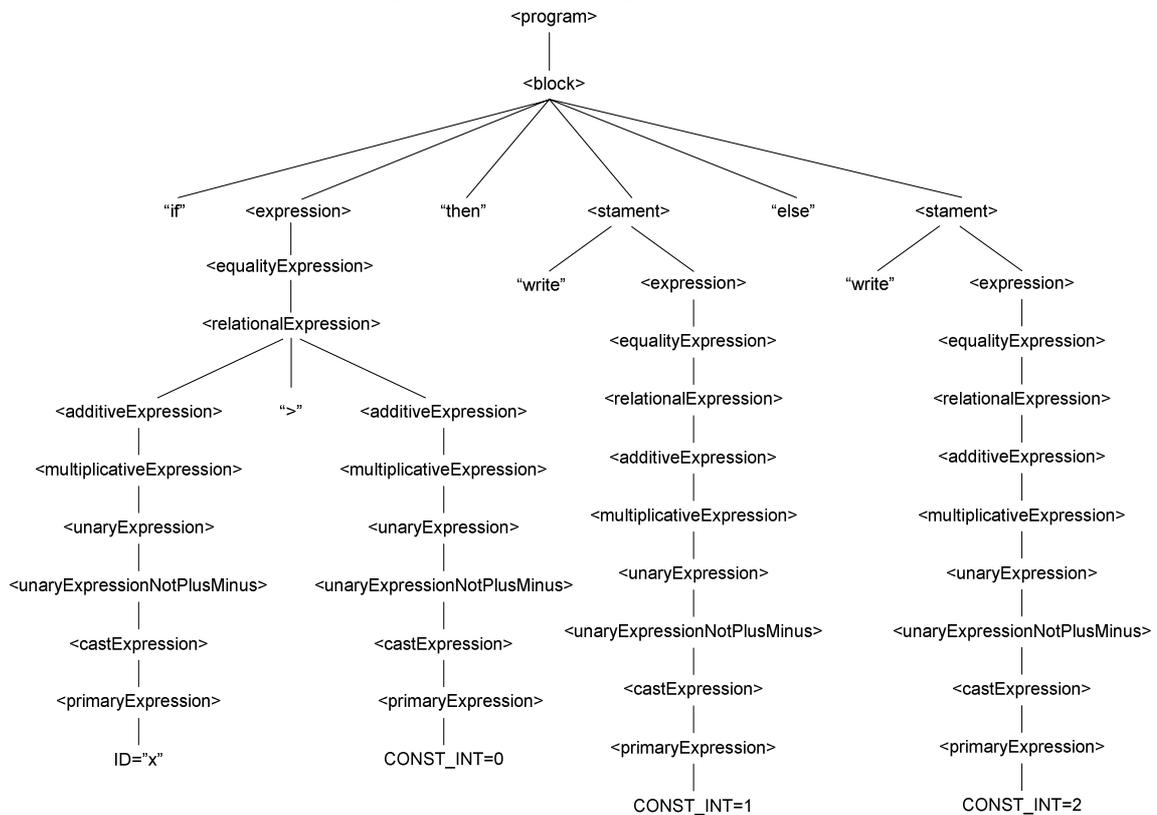


Figura 9: Arbol sintáctico asociado al programa ejemplo.

Como se va a realizar un intérprete y para ilustrar como con TyS se puede separar de manera efectiva la fase semántica del resto, se va a generar un árbol sintáctico abstracto, para después visitar cada uno de sus nodos y realizar por separado la fase semántica de la interpretación.

A la hora de realizar un AST tampoco es necesario representar en el árbol todas sus producciones de hecho como se verá en el siguiente punto, se puede representar un árbol más sencillo atendiendo a las precedencias y asociatividades de cada operador.

Lo que importa es que el árbol reconozca el mismo lenguaje. La simplificación se verá con más detalle en el siguiente punto, pero por ejemplo un árbol igualmente válido se puede ver en Figura 10.

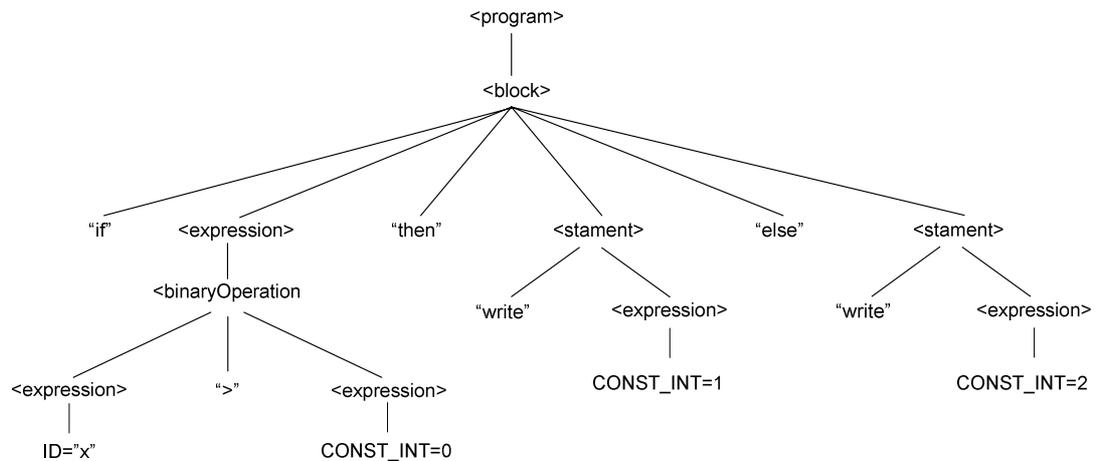


Figura 10: Arbol sintáctico simplificado asociado al programa ejemplo.

Que teniendo en cuenta las precedencias y asociatividades reconoce exactamente el mismo programa.

6.1.2.7 Construcción del árbol sintáctico Orientado a objetos

Para construir un árbol sintáctico sobre el que se puedan realizar operaciones semánticas, de interpretación, o cualquier otra acción, como generación de código, nuevamente las tecnologías orientadas a objetos nos resuelven de manera efectiva el problema. En concreto existe un patrón de diseño para comportamientos como éste: El patrón *Visitor* [GOF94 página 333].

Este patrón se utiliza cuando se desea realizar distintas operaciones a través de una estructura de objetos. Permitiendo definir operaciones nuevas sin tener que cambiar dicha estructura de objetos.

Se quieren definir dos operaciones, el chequeo de tipos, y la interpretación del programa.

Yendo paso a paso, lo primero es diseñar una jerarquía de clases que sirva para representar un programa reconocido por el lenguaje (sintácticamente) y cuyos roles de clases se adapten al patrón *Visitor*. Esta forma de proceder es la preferida por la metodología OORAM [Reenskaug95] observar la funcionalidad de los objetos e intentar sintetizar su comportamiento con algún patrón de diseño. Básicamente no reinventar la rueda.

Los participantes del patrón *Visitor* son los siguientes:

- **Visitor:** es una clase abstracta que representa una operación a realizar sobre un nodo de nuestro árbol sintáctico. Por cada operación que se desee realizar se debe definir una clase derivada de ésta. Se llamará a esta clase abstracta pura *Visitor*. Se deben definir métodos para implementar acciones sobre cada nodo (uno de los inconvenientes del *Visitor*), sus métodos dependen inextricablemente de la estructura de objetos a visitar. Por lo que se aplaza su especificación.
- **Concrete Visitor** implementa cada una de las operaciones declaradas en *Visitor*. Como se quieren implementar dos operaciones, se tendrán que definir dos subclases, por ejemplo *Interpreter*, y *Semantic*.

- **Element** es una clase abstracta que representa el nodo de la estructura de objetos. Define una operación para aceptar un objeto *Visitor* como argumento. Por convenio esta operación se suele llamar *accept()*. Será la forma que tiene la estructura de objetos de permitir que la operación se propague por todos sus objetos. Se adelanta que este nodo abstracto será una sentencia del programa: *Stament*
- **ConcreteElement** se denomina así a cada subclase de *Element* Su cometido será implementar una operación *accept()* que tome un objeto *Visitor* como uno de sus argumentos. El comportamiento de este método, suele ser el de un despachador, es decir se toma el argumento, y se llama a la operación adecuada en *Visitor* para el nodo que se está visitando. Como el argumento de tipo *Visitor* es en realidad una subclase que realiza una operación específica, lo que se hace es llamar al método de esa operación con argumento el nodo visitado: Genéricamente.

```
class ConcreteVisitorA {
    ...
    accept(Visitor v) {
        v.visitConcreteVisitor(A); //Dispatcher
    }
}
```

En el caso de estudio serán elementos concretos todas las sentencias posibles del programa junto con las expresiones.

- **ObjectStructure** simplemente provee un interfaz de alto nivel para permitir al objeto *Visitor* visitar los elementos. En el caso desarrollado, la clase *Program* que simplemente contenga una serie de sentencias.

Se tienen todos los participantes excepto los nodos concretos del árbol sintáctico. Se definirá un *ConcreteElement* por cada una de las sentencias posibles del programa. Esto es una clase abstracta base *Stament* que jugará el rol de *Element* y los siguientes *ConcreteElements*.

Tipo de sentencia	Clase que la representa
Sentencia (abstracta)	Stament
Definición de Variable	VarDer
Asignación	Assignation
Sentencia IF	If
Sentencia While	While
Bloque de sentencias	Block
Sentencia de lectura	Read
Sentencia de escritura	Write

Como se puede observar las expresiones no aparecen por ninguna parte, esto es porque en este lenguaje no se tratan como sentencias, hay que ocuparse de ella, pero antes se terminará de mapear el modelo al patrón *Visitor*.

Una vez que se tienen las *ConcreteElement* la clase *Visitor* tiene que implementar métodos *visit* para cada tipo de nodo que puede visitar. Igualmente sus subclases, Es el gran inconveniente de este patrón que si se cambia la estructura del lenguaje es necesario cambiar el interfaz *Visitor* y por tanto todas su subclases.

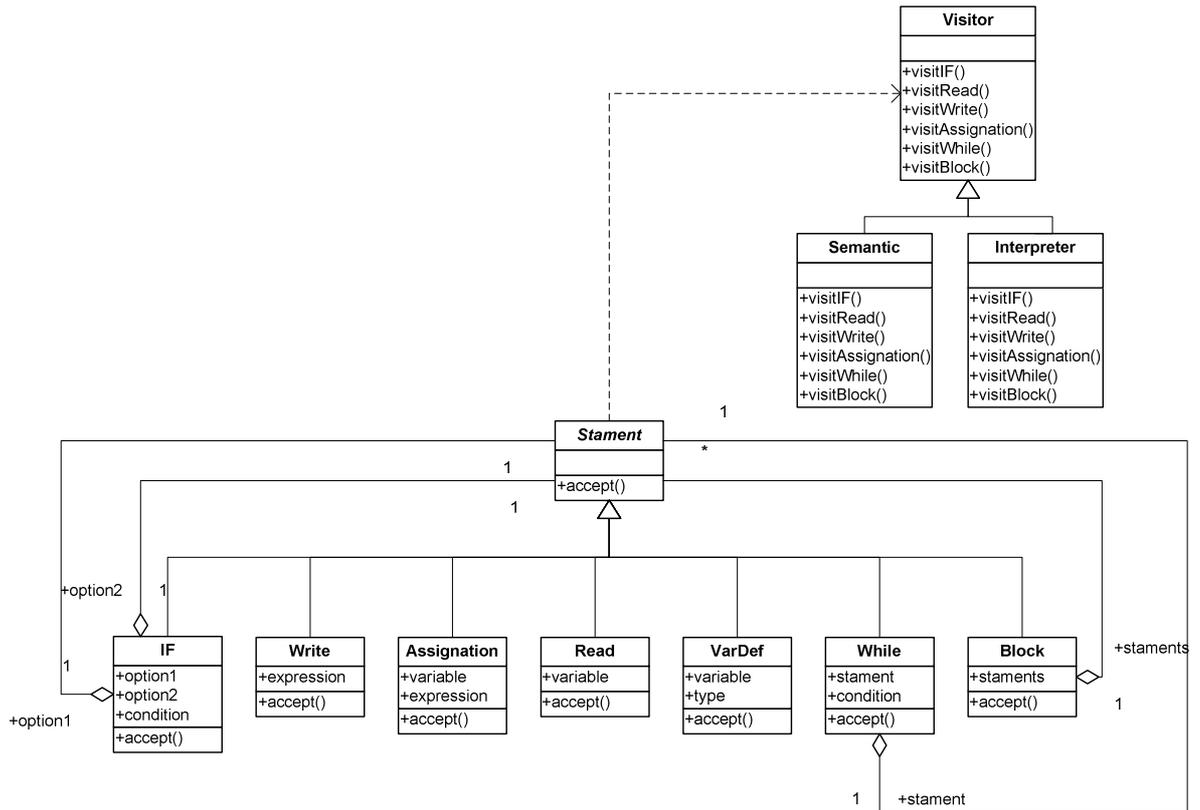


Figura 11: Diagrama de clases de las sentencias del lenguaje Frog.

Ahora hay que tener en cuenta las expresiones.

Con las expresiones tenemos que realizar dos operaciones fundamentales, la comprobación de tipos y la evaluación. De manera que se declara una clase abstracta *Expression* con dos métodos para realizar estas operaciones *typeChecking()* y *evaluation()*.

Al añadir estas operaciones el diseñador se puede dar cuenta de que esta clase también sería candidata a formar parte del *Visitor* ya que se puede hacer que acepten una operación y luego visitarlas para evaluarlas y realizar la comprobación de tipos. Así que es mejor olvidarse, de las operaciones *typeChecking()* y *evaluation()* dejando que las realicen los *ConcreteVisitor Semantic* e *Interpreter*.

Por último como las expresiones tendrán asociadas un tipo será necesario un método consultor *getType()* para saber el tipo de dicha expresión.

La clase *Value* sirve para propósitos de interpretación, encapsula el valor de la expresión.

La clase *Type* representa el tipo asociado a la expresión. No se detallará de momento, pues será en el módulo semántico donde se definirá aprovechando el *framework TyS*.

Ahora para cada una de las operaciones definidas en *Frog* se deriva una subclase de expresión. Para evitar una explosión de subclases, se agrupan por semejanza de comportamiento en operaciones, aritméticas, lógicas. Ya que se puede aplicar la precedencia y asociatividad de los operadores para un correcto recorrido.

<i>Tipo de expresión (u operación)</i>	<i>Clase que la representa</i>
Expresión (abstracta)	Expression
Variable	Variable

Constante entera, real, etc	Constant
Resto, operación módulo	ModOp
Operación lógica: and, or	LogicalOp
Operación de igualdad: != ==	EqualityOp
Operación de comparación: > < >= <=	RelationalOp
Operaciones aritméticas: + - * /	AritmeticalOp
Operaciones unarias: - + !	UnaryOp
Casting	CastOp

Cada operación así representada tomará una o dos expresiones como argumentos, las que indique su aridad.

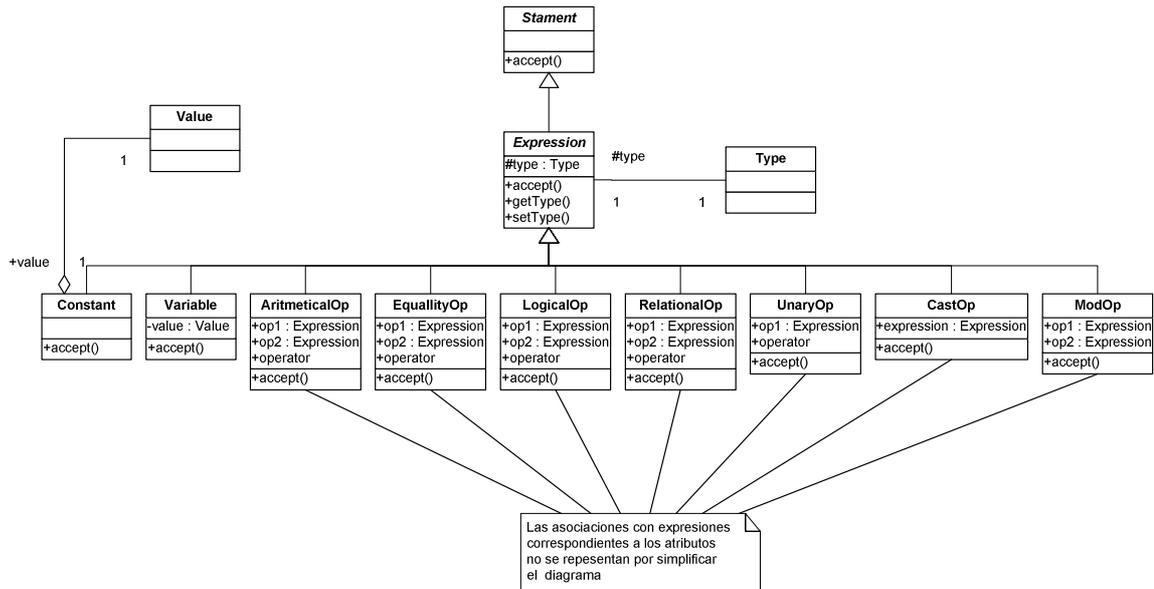


Figura 12: Diagrama de clases de las expresiones del lenguaje *Frog*.

La clase *Constant* tiene un agregado de tipo *Value* esta clase la usará el intérprete para envolver todos los valores posibles. La colaboración (Figura 13) muestra como se mapean las clases en el patrón.

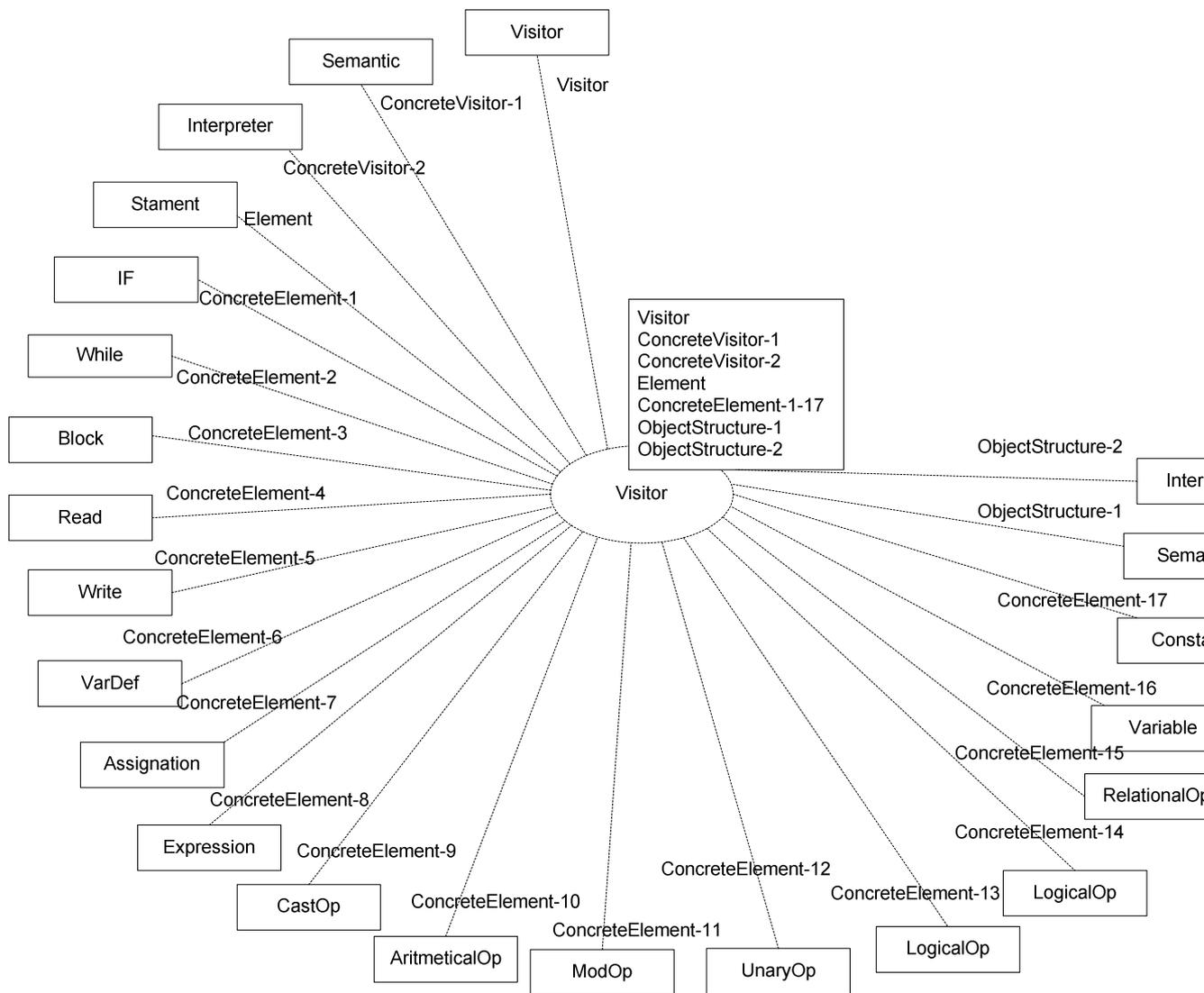


Figura 13: Colaboración *Visitor* que estructura el AST para el lenguaje *Frog*.

Y el diagrama de clases completo, en el que se han omitido algunas agregaciones sería el que muestra la Figura 14.

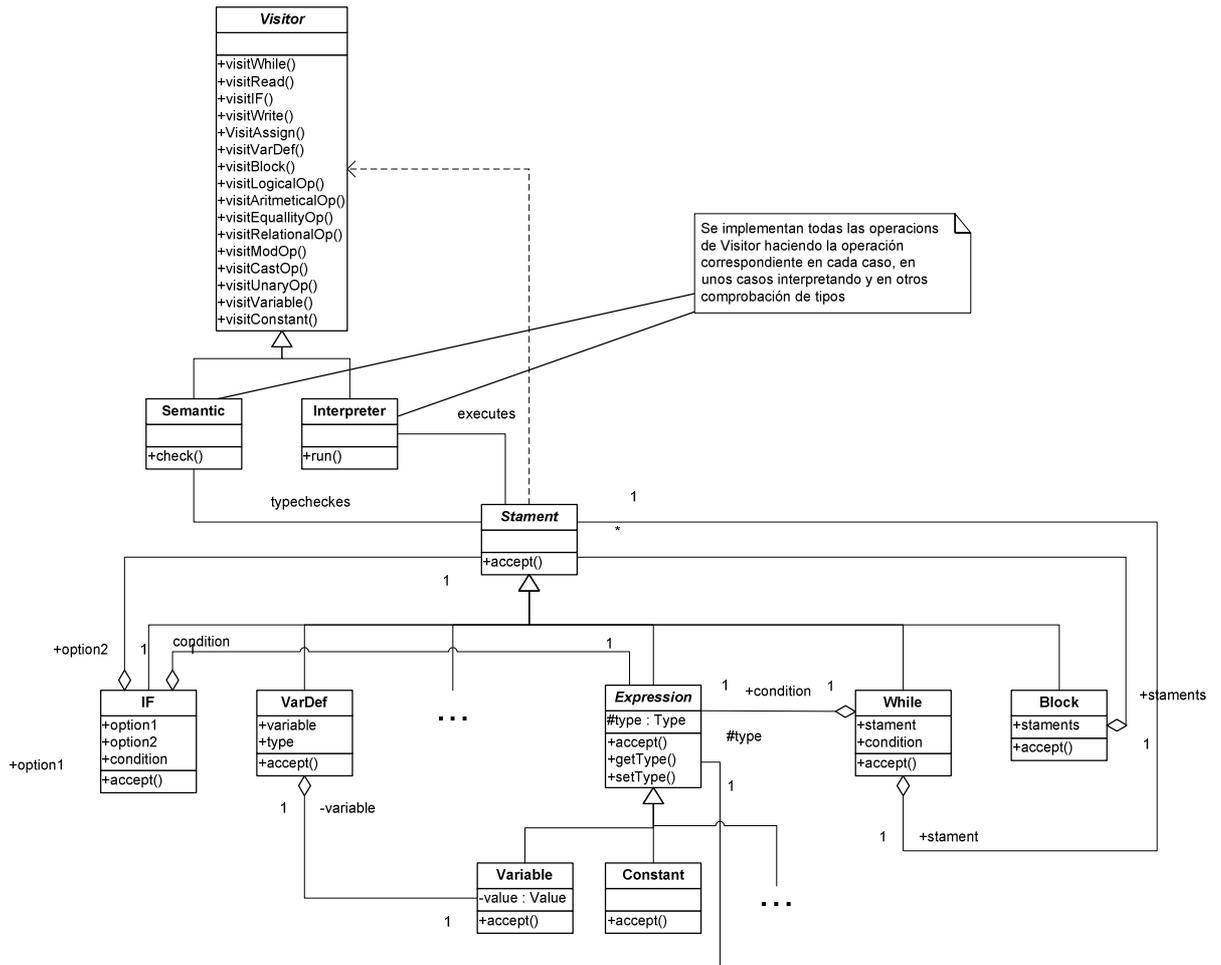


Figura 14: Diagrama de clases que modela el patrón *Visitor* para *Frog*.

Se ha implementado una versión de este intérprete usando como *parser* la herramienta ANTLR. Para ver el código completo se puede consultar CDROM en el directorio *Frog*, el fichero *Interpreter.java*. A medida que se reconocen expresiones y sentencias se va construyendo la estructura de objetos y se va conectando formando el árbol sintáctico adecuado al programa. Se genera un árbol sintáctico manualmente en lugar de usar el AST que puede generar ANTLR (ver <http://www.antlr.org>) porque resulta más ilustrativo mostrar el uso del *Visitor* y la construcción de nodos en lugar de usar los *TreeWalker* de ANTLR.

Para terminar con el módulo sintáctico, se muestra como se mapearía en objetos el árbol sintáctico anterior.

```

{
  if x >0 then write 1;
  else write 2;
}
  
```

Un análisis sintáctico de este programa correcto mapeado en la estructura de objetos definida anteriormente daría como resultado el conjunto de objetos mostrado en la Figura 15.

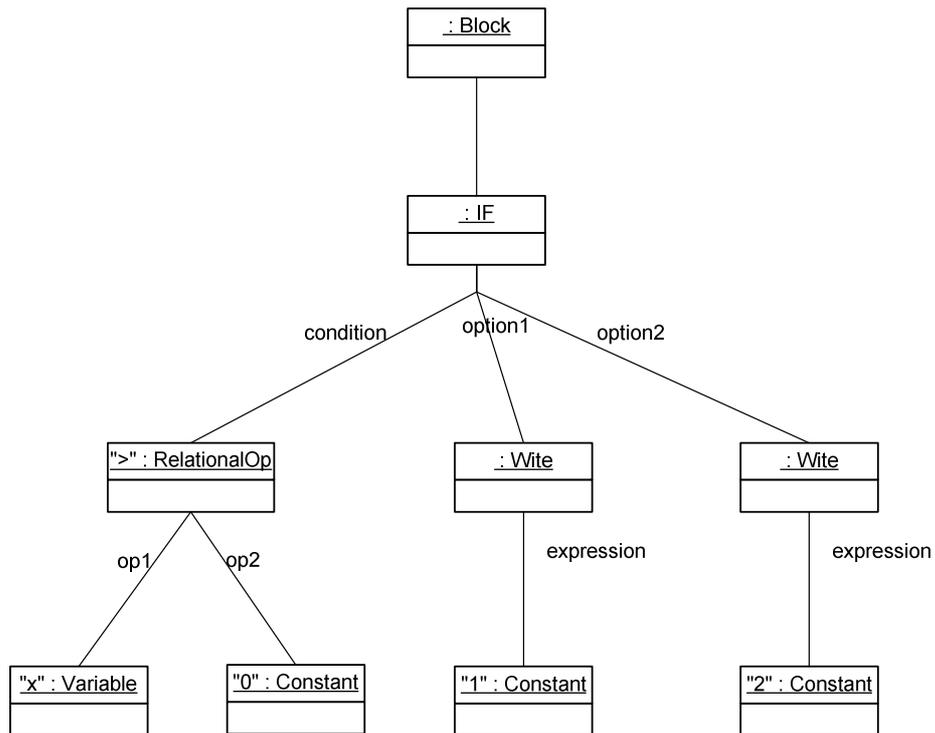


Figura 15: Diagrama de objetos, que representa el árbol sintáctico del ejemplo.

Cada una de los objetos que representan sentencias y expresiones tienen un método *accept()* que recibe un objeto *Visitor* por lo que se podrán definir operaciones sobre ellos.

En concreto El objeto *Block* resultante servirá como argumento a las clases que juegan el rol de *ObjectStructure*: el intérprete y el analizador semántico. El recorrido de cada uno de los *Visitor* comenzará así:

```

public class Semantic extends Visitor{
    . . .
    void check(Stament s) throws TypeException {
        s.accept(this);
    }
    . . .
}

public class Interpreter extends Visitor {
    . . .
    public void run(Stament s) throws TypeException{
        s.accept(this);
    }
    . . .
}
  
```

6.1.3 Módulo semántico

Ahora viene la parte nueva e interesante: la construcción de un semántico utilizando el *framework* TyS. Hasta ahora la construcción de analizadores semántico era más o menos un “arte”. Ahora se demostrará como utilizando el *framework* TyS se automatiza casi completamente este proceso.

Se seguirá un diseño *top-down* pues de esta manera será más fácil definir las operaciones sobre tipos necesarias. En principio se supone que ya se cuenta con la clase *Type*, que será la que genere la aplicación TyCC. Posteriormente se irá realizando el módulo semántico e irán surgiendo las operaciones necesarias para la comprobación de tipos.

El uso de los tipos se dividirá en tres aspectos fundamentales. A gran nivel en el *ConcreteVisitor* semántico, en la creación de objetos tipo y su uso en las expresiones, donde quedarán definidas casi todas las operaciones.

6.1.3.1 Construcción de objetos tipo

Se usará una clase para construir adecuadamente los objetos tipo. Esta clase es una especie de *wrapper* sobre el comprobador de tipos. Se encarga de construir objetos más específicos, como por ejemplo constantes, en los que el tipo es una sola de sus componentes. Esta clase recibirá el nombre de *TypeBuilder*.

Tiene 3 métodos fundamentales:

getType() devuelve un objeto *Type*, alzando una excepción si no se trata de uno de los tipos definidos en el lenguaje. El *framework* TyS permite tratar tipos no definidos en el lenguaje con vistas a que el usuario los decida definir más tarde.

```
public Type getType(String s) throws TypeException {
    Type t = typesTable.getType(s);
    if (t.isReference()) throw new TypeException("Type " + s +
        ": This Language has not named types");
    return t;
}
```

constReal() devuelve un objeto de tipo *Constant*, con el tipo real asociado más pequeño en el que quepa el valor que recibe como argumento.

```
public Constant constReal(String s) throws TypeException {
    double d = java.lang.Double.parseDouble(s);
    Type t = null;

    if (d==0 || (d<=Float.MAX_VALUE && d>=Float.MIN_VALUE) )
        t = typesTable.getType("Real");
    else if (d<=java.lang.Double.MAX_VALUE &&
        d>=java.lang.Double.MIN_VALUE)
        t = typesTable.getType("Double");
    else throw new TypeException("Strange Real Constant : " +
        s);
    return new Constant( new Value (d) , t );
}
```

constInt() devuelve un objeto de tipo *Constant*, con el tipo entero asociado más pequeño en el que quepa el valor que recibe como argumento.

```
public Constant constInt(String s) throws TypeException {
    double d = java.lang.Double.parseDouble(s);
    Type t = null;
    if ((short)d<=java.lang.Short.MAX_VALUE &&
        (short)d>=java.lang.Short.MIN_VALUE)
        t = typesTable.getType("Short");
    else if ((int)d<=Integer.MAX_VALUE &&
        (int)d>=Integer.MIN_VALUE)
        t = typesTable.getType("Int");
    else if ((long)d<=java.lang.Long.MAX_VALUE &&
        (long)d>=java.lang.Long.MIN_VALUE)
        t = typesTable.getType("Long");
    else throw new TypeException("Strange Integer Constant : " + s);
    return new Constant( new Value (d), t );
}
```

Hasta ahora no se ha sacado ninguna operación útil para implementar en el comprobador de tipos, pues la operación *getType()* es generada automáticamente por la aplicación TyCC.

6.1.3.2 Comprobación de tipos específica de las sentencias

En este punto surgirá alguna de las operaciones que se necesitarán definir en el comprobador de tipos.

Hay que recordar que la clase semántico es una implementación de la clase *Visitor* cuyo cometido es ir realizando un recorrido en un orden determinado por los nodos del árbol sintáctico y realizar operaciones sobre cada uno de ellos.

Cuando un elemento es visitado, llama a la operación de la clase *Visitor* que corresponda con su clase usándose así mismo como argumento, para que el *Visitor* pueda acceder a su estado.

Se iterará sentencia por sentencia scando los casos de uso [Cockburn01] del módulo semántico, en función de lo que necesite cada sentencia para ejecutarse correctamente. De esta forma vamos se sacarán algunas operaciones a implementar por el comprobador de tipos. Este método es algo así como lo que proponía Rumbaugh en [Rumbaugh95]de ir buscando nombres en un enunciado de un problema para sacar clases, pero con utilidad práctica.

El listado del este fichero donde se definen todas las sentencias se puede encontrar en el CD-ROM en el fichero *frog\staments.java*

6.1.3.2.1 Block

Esa clase cubre el bloque de sentencias, en este caso no hay nada útil que sacar de aquí para la comprobación de tipos, pues su análisis semántico se reduce a iterar sobre cada una de sus sentencias componentes analizándola a su vez semánticamente.

```
void visitBlock(Block b) throws TypeException{
    Iterator it = b.staments.iterator();
    Object ret = null;
    while (it.hasNext()) ((Stament)it.next()).accept(this);
}
```

Lo que se ve aquí en principio puede parecer una barbaridad para los puristas de la OO. Se está accediendo directamente a los atributos de una clase, en lugar de tratarlos como propiedades. Ese es el gran inconveniente del *Visitor* rompe la encapsulación. Se asume que cada *ConcreteElement* tiene un interfaz lo suficientemente amplio como para que cada *ConcreteVisitor* pueda realizar su trabajo.

Lo que se debería hacer cuando se manejan los atributos de una clase es definir operaciones públicas consultoras y modificadoras (*getters* y *setters*) que permitan el acceso y modificación de propiedades. Pero se prefiere no adoptar un criterio tan rígido, teniendo en cuenta que en realidad los nodos del árbol sintáctico no representan ninguna entidad. Hay que considerar que, aunque hemos definido los *ConcreteVisitor* como clases, su cometido es el de meros registros de datos con alguna operación, y que su encapsulación carece de sentido.

6.1.3.2.2 While

Esta clase cubre la sentencia while-do. Su análisis semántico se reduce a comprobar que la guarda del bucle es un elemento condicional válido y a propagar el análisis semánticamente a la sentencia del cuerpo del bucle.

```
Object visitWhile(While w) throws TypeException {
    Type cond = (Type)w.condition.accept(this);
```

```

        if (!cond.isBoolean())
            throw new TypeException("Cannot use a " + cond.getName() +
                " as a guard of a While-stament must be a boolean
expression");
        w.stament.accept(this);
        return null;
    }

```

Para ello se visita el nodo de la expresión condicional. Se supone que este nodo devolverá un objeto de tipo *Type* por lo que el *cast* es seguro. En caso de haber alguna inconsistencia en la comprobación de tipos se alzará una excepción de tipo *TypeException*. En las visitas a las expresiones estará el grueso de la comprobación de tipos.

Si la condición es un tipo válido queda por comprobar que sea una guarda válida para un bucle. Aquí aparece la primera operación de comprobación de tipos. *isBoolean()*, Este método debe indicar si un objeto *Type* puede ser usado como guarda de bucle.

Si todo es correcto se sigue visitando la sentencia a repetir. Nótese nuevamente la violación total de la encapsulación.

6.1.3.2.3 IF

Esta clase cubre la sentencia *if-then-else*. Su análisis semántico se reduce a comprobar que la condición es una expresión condicional válida y a extender el análisis semántico de la sentencia asociada a la parte *then* y, en caso de existir, a la parte *else*.

```

Object visitIF(IF i) throws TypeException {
    Type cond = (Type)i.condition.accept(this);
    if (!cond.isBoolean())
        throw new TypeException("Cannot use a " + cond.getName() +
            " as a condition of a If-stament. It must be a boolean
            expression");
    i.option1.accept(this);
    if (i.option2 != null) i.option2.accept(this);
    return null;
}

```

No hay ninguna operación nueva que añadir al comprobador de tipos. De nuevo se hace uso de *isBoolean()*.

6.1.3.2.4 Read

Esta clase cubre la sentencia de lectura de datos por teclado.

```

Object visitRead(Read r) throws TypeException {
    return r.variable.accept(this);
}

```

Dado que todos los tipos pertenecientes al lenguaje son susceptibles de ser leídos por teclados no es necesario realizar ninguna comprobación adicional. Basta visitar la variable para que compruebe que no hay inconsistencias en la expresión a leer.

6.1.3.2.5 Write

Esta clase cubre la sentencia de escritura de datos por pantalla.

```

    Object visitWrite(Write w) throws TypeException {
        return w.expression.accept(this);
    }

```

Dado que todos los tipos pertenecientes al lenguaje son susceptibles de ser impresos en pantalla no es necesario realizar ninguna comprobación adicional. Basta visitar la expresión a escribir para que se compruebe la falta de inconsistencias.

6.1.3.2.6 Assign

Esta clase cubre la sentencia de asignación de valor de una expresión a una variable.

```

    Object visitAssign(Assign a) throws TypeException {
        Type tv = (Type)a.variable.accept(this);
        Type te = (Type)a.expression.accept(this);
        return tv.assignment(te);
    }

```

Se debe como siempre visitar la expresión para comprobar que todo es correcto. En el caso de la variable este método comprobará además que la variable esté correctamente definida. Con ello si todo es correcto se obtiene un objeto *Type* asociado a la variable y otro asociado a la expresión que se va a asignar.

Aquí hay que definir otra operación para comprobar si una asignación entre dos tipos es válida: *assignment(Type t)* será el método de la clase *Type* que validará si el tipo que recibe como argumento se puede asignar al objeto que llama al método. En caso contrario alzará una excepción.

6.1.3.2.7 VarDef

Esta clase cubre la definición de variables.

Aquí no hay que hacer ninguna comprobación de tipos. Únicamente inserta la variable en la tabla de símbolos (el contexto).

```

    void visitVarDef(VarDef a) throws TypeException{
        context.insert(a.variable, new STSlot(new Value(), a.type));
    }

```

Se debe introducir la variable en un elemento de tipo *STSlot* (*Symbol Table Slot*) que son los tipos de dato que maneja el contexto.

El método *insert()* alza una excepción en el caso de que la variable este definida en ese contexto, es decir, en el ámbito en el que se encuentre. En este ejemplo todas las variables tienen ámbito global, es decir que no hay una pila de tablas de símbolos.

6.1.3.3 **Comprobación de tipos en las expresiones**

A continuación se examina cómo implementar los métodos *visitxxx* de todas las expresiones posibles en el lenguaje *Frog*.

Se han implementado las expresiones con un atributo protegido *type* que almacena el tipo asociado a la expresión; pero este tipo no es inicializado correctamente hasta que le visite un analizador semántico gracias a la operación *accept()*

6.1.3.3.1 Constant

Esta clase cubre las expresiones formadas únicamente por una constante.

```
Object visitConstant(Constant c) throws TypeException {
    return c.getType();
}
```

Aquí no hay que hacer comprobación de tipos. El método *visitConstant()* únicamente debe devolver el tipo asociado a la constante.

6.1.3.3.2 LogicalOp

Esta clase cubre las operaciones lógicas *and* y *or*. Primero se realiza la comprobación de tipos de cada expresión asociada. En este caso se debe comprobar que cada uno de los operandos se le pueda aplicar estas operaciones. No hay interferencias entre el tipo de uno y de otro, por lo que se puede definir una operación de comprobación de aridad uno, que se refiere únicamente al objeto que la invoca.

```
Object visitLogicalOp(LogicalOp l) throws TypeException {
    ((Type)l.op1.accept(this)).logical();
    return l.setType(((Type)l.op2.accept(this)).logical());
}
```

Surge una nueva operación para la comprobación de tipos *logical()*, que alza una excepción si el tipo que la invoca no es válido para realizar operaciones *and-or*, y devuelve un objeto *Type* que representa el tipo devuelto por una expresión *and-or*

6.1.3.3.3 AritmeticalOp

Esta clase cubre expresiones relacionadas por las operaciones aritméticas, tales como suma, resta, producto, y división.

```
Object visitAritmeticalOp(AritmeticalOp b) throws TypeException {
    Type t1 = (Type)b.op1.accept(this);
    Type t2 = (Type)b.op2.accept(this);
    return b.setType(t1.aritmetical(t2, b.operator));
}
```

Como en los casos anteriores se visita cada expresión asociada y se define una nueva operación de comprobación de tipos, en este caso de aridad 3, puesto que se pasa también como parámetro el operador. De manera que otra nueva operación de comprobación de tipos. *aritmetical()* que comprobará si entre dos expresiones se puede realizar la operación aritmética que se pasa como parámetro.

6.1.3.3.4 ModOp

Esta clase cubre expresiones relacionadas por la operación aritmética módulo (resto). Como siempre primero se realiza la visita de cada expresión asociada.

```
Object visitModOp(ModOp m) throws TypeException {
    Type t1 = (Type)m.op1.accept(this);
    Type t2 = (Type)m.op2.accept(this);
    return m.setType(t1.mod(t2));
}
```

Se define otra operación de comprobación de tipos de aridad 2 que comprobará si se puede realizar la operación módulo del objeto llamador, con el tipo que recibe como argumento: *mod(Type)*.

6.1.3.3.5 RelationalOp

Esta clase cubre expresiones relacionadas por las operaciones relacionales, tales como mayor, menor, mayor o igual y menor o igual.

```
Object visitRelationalOp(RelationalOp r) throws TypeException {
    Type t1 = (Type)r.op1.accept(this);
    Type t2 = (Type)r.op2.accept(this);
    return r.setType(t1.relationalOp(t2, r.operator));
}
```

Como en los casos anteriores se visita cada expresión asociada y se define una nueva operación de comprobación de tipos, en este caso de aridad 3, puesto que se pasa también como parámetro el operador. De manera que surge otra nueva operación de comprobación de tipos: *relationalOp()*, que comprobará si entre dos expresiones se puede realizar la operación relacional que se pasa como parámetro.

6.1.3.3.6 EquallityOp

Esta clase cubre expresiones de comprobación de igualdad tales como igual que, o distinto de.

```
Object visitEquallityOp (EquallityOp e) throws TypeException {
    Type t1 = (Type)e.op1.accept(this);
    Type t2 = (Type)e.op2.accept(this);
    return e.setType(t1.equallity(t2, e.operator));
}
```

Como en los casos anteriores se visita cada expresión asociada y se define una nueva operación de comprobación de tipos, en este caso de aridad 3, puesto que se pasa también como parámetro el operador. Así tenemos otra nueva operación de comprobación de tipos: *equallity()*, que comprobará si entre dos expresiones se puede realizar la operación de comprobación de igualdad que se pasa como parámetro.

6.1.3.3.7 UnaryOp

Esta clase se cubre las expresiones a las que se les aplica un operador unario del tipo más, menos, o negación lógica (*not*).

```
Object visitUnaryOp(UnaryOp u) throws TypeException {
    Type t1 = (Type)u.op1.accept(this);
    return u.setType(u.operator == BNOT ? t1.unaryNot()
                    : t1.unary(u.operator));
}
```

Como siempre se visita cada expresión asociada y se define una nueva operación de comprobación de tipos de aridad 2, puesto que se pasa también como parámetro el operador. La operación *unary()*: que comprobará si el tipo de una expresión puede realizar la operación unaria que recibe como parámetro.

6.1.3.3.8 CastOp

Esta clase cubre las expresiones a las que se les aplica un operador de ahormado o *casting*.

```
Object visitCastOp(CastOp c) throws TypeException {
    c.expression.accept(this);
    return c.getType();
}
```

Se visita la expresión asociada y se define una nueva operación de comprobación de tipos de aridad 2, puesto que se pasa también como parámetro el tipo al que se quiere ahormar. La operación *cast()* comprobará si el tipo de una expresión se puede promocionar explícitamente al tipo que recibe como parámetro.

6.1.3.3.9 Variable

En esta clase se cubren las variables como expresiones

```
Object visitVariable(Variable v) throws TypeException {
    STSlot s = context.look(v.getName());
    if (s == null) throw new InterpreterException("Variable " +
        v.getName() + " not declared");
    return v.setType(s.type);
}
```

Lo que se hace es comprobar que la variable esté definida. En el caso de que lo esté se devuelve su tipo asociado, sino se alza una excepción informando del error.

Con el análisis de los casos de uso de las sentencias tenemos ya por definir las siguientes operaciones de comprobación de tipos: *checkBoolean()*, *cast()*, *unary()*, *arimetical()*, *relational()*, *equality()*, *mod()*, *unary()* y *logical()*.

6.2 **Especificación de tipos (Type Specification)**

Se debería realizar un esquema de los tipos que van a formar parte del lenguaje que se va a definir y las operaciones que soportarán dichos tipos.

<i>Tipos</i>	<i>Naturaleza</i>
Short	Entera
Int	Entera
Long	Entera
Float	Real
Double	Real

<i>Nombre de la operación</i>	<i>Operación</i>	<i>Semántica</i>	<i>Método del Comprobador de tipos</i>
Asignación	$v = e$	<i>e</i> debe ser una expresión promocionable implícitamente al tipo de la variable <i>v</i> . Se devuelve el tipo de la variable <i>v</i> . En caso de no ser posible la asignación se lanza una excepción.	<i>assignment(Type t)</i>

And lógico	$e1 \ \& \ e2$	$e1$ y $e2$ deben ser operandos enteros. Se devuelve el tipo <i>Short</i> . En caso de no ser alguno de ellos enteros se alza una excepción	<i>logical():Type, sobre cada operando</i>
Or lógico	$e1 \ \ e2$	$e1$ y $e2$ deben ser operandos enteros. Se devuelve el tipo <i>Short</i> . En caso de no ser alguno de ellos enteros se alza una excepción	<i>logical():Type, sobre cada operando</i>
Not lógico	$! \ e$	e debe ser un operando entero. Se devuelve el tipo <i>Short</i> . En caso de no ser operando entero se alza una excepción.	<i>unaryNot():Type</i>
Mayor	$e1 \ > \ e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación mayor. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Short</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>relational(Type t, int operator):Type</i>
Menor	$e1 \ < \ e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación menor. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Short</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>relational(Type t, int operator):Type</i>
Menor o igual	$e1 \ <= \ e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación menor o igual. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Short</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>relational(Type t, int operator) :Type</i>
Mayor o igual	$e1 \ >= \ e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación mayor o igual. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Short</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>relational(Type t, int operator) :Type</i>

Igual	$e1 == e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación de igualdad. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Short</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>equality(Type t, int operator) :Type</i>
Distinto	$e1 != e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación distinto. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Short</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>equality(Type t, int operator) :Type</i>
Suma	$e1 + e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido la suma. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo con más rango de las dos expresiones. Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>aritmetical(Type t, int operator) :Type</i>
Resta	$e1 - e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido la suma. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo con más rango de las dos expresiones. Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>aritmetical(Type t, int operator) :Type</i>
Mas	$+ e$	e debe ser una expresión de tipo numérico sobre la que tenga sentido aplicar el más unario. Se devuelve el tipo de e . Si no es una expresión válida se devuelve un error.	<i>unary(int operator) :Type</i>
Menos	$- e$	e debe ser una expresión de tipo numérico sobre la que tenga sentido la operación el menos unario. Se devuelve el tipo de e . Si no es una expresión válida se devuelve un error.	<i>unary(int operator) :Type</i>

División	$e1 / e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido la división. Además debe existir una promoción implícita de uno de los operandos al otro Se devuelve el tipo con más rango de las dos expresiones. Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>aritmetical(Type t, int operator):Type</i>
Producto	$e1 * e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido el producto. Además debe existir una promoción implícita de uno de los operandos al otro Se devuelve el tipo con más rango de las dos expresiones. Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>aritmetical(Type t, int operator):Type</i>
Modulo	$e1 \% e2$	$e1$ y $e2$ deben ser operandos enteros. Se devuelve el tipo con más rango de las dos expresiones. Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>mod(Type t):Type</i>
Casting	$(Type) e$	e es una expresión de cualquier tipo, y $type$ una expresión de tipo. Se devuelve el tipo $type$. En principio se permiten todas las conversiones no se alzan errores.	<i>cast(Type t):Type</i>

A la vista de la tabla se ve que será necesaria otra operación más para uso de interno del comprobador de tipo. *majorType(Type t)*, que devuelva el tipo que está más a la derecha en la cadena de promociones, de los dos objetos tipo parámetro (el llamador, y el parámetro). En caso de no existir una cadena de promociones se devolverá null.

6.2.1 Preámbulo

De momento no es necesario introducir ninguna cláusula *import*, ni código adicional, con lo que queda vacío.

6.2.2 Sección de declaración de tipos

Se pretende que el lenguaje cuente con 5 tipos predefinidos todos ellos numéricos: *Short, Int, Long, Float* y *Double*.

Existirá una promoción implícita entre ellos desde el tipo con menor rango al tipo con mayor rango. De esta manera se crea un fichero de texto ASCII, que se llame por ejemplo *Types.txt* y se escribe lo siguiente:

```
Types = { Short < Int < Long < Real < Double }
```

Esta declaración constituye la sección de declaración de tipos del fichero de especificación de tipos de la aplicación TyCC. Se declara que existen 5 tipos, y la promoción implícita que existe entre ellos. En caso de que entre algún tipo no existiese promoción implícita estaría separado del resto por caracteres blancos (espacios, tabuladores, saltos de línea). Pero todos los tipos deben aparecer en esta sección.

Aunque TyCC define 3 tipos más automáticamente: *BaseType*, *NamedType* y *TypeReference*. Estos no deben aparecer nunca en la sección de declaración de tipos.

El fichero completo de especificaciones de tipos puede consultarse en el CD-ROM en el fichero *FrogTypes.txt*.

6.2.3 Definición de tipos

El fichero de especificación de tipos consta de tres partes: preámbulo, sección de declaración de tipos –que acaba de definir–, y sección de definición de tipos.

Realizando el diseño del módulo semántico se había constatado la necesidad de una serie de operaciones a definir para la comprobación de tipos: *isBoolean()*, *assignation()*, *logical()*, *aritmetical()*, *mod()*, *relational()*, *equallity()*, *unary()*, *cast()* y *majorType()*.

Se empezará el estudio por las operaciones que menos características vayan a usar de la especificación de tipos para TyCC.

6.2.3.1 *logical()*

Esta operación debe comprobar que se pueden realizar las operaciones lógicas *and-or* sobre el objeto invocador del método. La forma más sencilla, es definir esta operación en todos los tipos que pueden realizar esta operación y en el resto dejarla sin definir. Si se deja una operación sin definir en algún tipo y no está definido en la clase *BaseType* (se hablará de ella en más adelante), se genera una operación con cabecera idéntica por defecto, que alza una excepción indicando que esa operación no es apropiada para ese tipo.

En computación las operaciones *and-or* suelen tener sentido sólo si hablamos de números enteros, así que se definirán para todos aquellos tipos enteros, es decir: *Short*, *Int*, y *Long*.

Según la especificación de la tabla semántica de operadores debe devolverse un tipo *Short*

Para definir un tipo se pone la palabra reservada `class` seguida del nombre del tipo y entre llaves se definen las operaciones.

Las operaciones se especifican poniendo el nombre del tipo de retorno, que es cualquier identificador de tipo Java válido, el nombre del método seguido de su signatura al estilo Java. No se ponen declaradores de visibilidad ni se especifican si se puede alzar excepciones, todos los métodos pueden alzar por defecto *TypeException*.

```
class Int {
    Type logical() {
        return Type.typesTable.getType("Short");
    }
    . . .
}
```

Se usará como tabla de tipos la variable de clase *typesTable*, de la clase *Type*. De otra forma se tendría que andar pasando como argumento la tabla de tipos en aquellas operaciones

que requiriesen un objeto tipo a partir de su nombre. Es decir, como tabla de tipos se usará un objeto estático.

Igualmente para la el tipo *Long*

```
class Long {
    Type logical() {
        return Type.typesTable.getType("Short");
    }

    . . .
}
```

Y para el tipo *Short*

```
class Short {
    Type logical() {
        return Type.typesTable.getType("Short");
    }

    . . .
}
```

Aquí se aprecia algo que puede parecer extraño. Parecería más sencillo en lugar de hacer una llamada a la tabla de tipos para el tipo *Short*, devolver simplemente *this*, pues estamos definiendo una operación sobre el tipo *Short*.

La única limitación que tiene la definición de operaciones es que no se puede usar el parámetro *this* para acceder a los objetos *Type*, pues en realidad el código que se está escribiendo en las operaciones no lo ejecuta un objeto *Type*, sino a un objeto delegado. La forma correcta de referirse a la instancia implícita de una operación es mediante la tabla de tipos.

Es decir, mirando en el código que va a generar TyCC lo que se define en la clase *Type* para *logical()* sería un código con el siguiente aspecto:

```
class Type {
    private BaseType type;
    . . .
    Type logical () throws TypeException {
        return type.logical()
    }
    . . .
}
```

Es decir que la clase *Type* hace como un envoltorio de la clase *BaseType*, define todas las operaciones que se hayan definido en la sección de declaraciones de tipos como métodos que delegan esa operación en un método con el mismo nombre de la clase *BaseType*.

La operación *logical()* ya está completamente definida. Adelantándose al código que va a generar TyCC se observa que para la clase *BaseType* se ha implementado una operación *logical()* tal que así.

```
class BaseType {
    . . .
    Type logical () throws TypeException {
        throw new TypeException ("operation Type logical () not available
for class "+ getType());
    }
}
```

```
...
}
```

La clase *BaseType* es la clase base de todos los tipos, de manera que si un objeto no entero llama a la operación *logical()* dado que no la tiene redefinida se llamará a la de su clase base que la tiene implementada de esta manera, alzándose un error de operación no apropiada.

6.2.3.2 *unaryNot(int operator)*

Esta operación debe comprobar que se puede realizar la operación de negación lógica sobre el objeto invocador del método. La forma más sencilla, es definir esta operación en todos los tipos que pueden realizar esta operación y en el resto dejarlo sin definir.

En computación la negación lógica suele tener sentido sólo en números enteros, así pues se define para todos los tipos enteros: *Short*, *Int*, y *Long*.

```
class Int {
    ...
    Type unaryNot() {
        return Type.typesTable.getType("Short");
    }
    ...
}
```

Igualmente se procederá para los tipos *Long*, y *Short*.

6.2.3.3 *mod(Type t)*

Esta operación debe comprobar que se puede realizar la operación resto entre dos tipos. En concreto que se puede realizar el módulo del objeto llamador con el tipo que recibe como argumento.

La forma más sencilla, es definir esta operación en todos los tipos que pueden realizar la operación módulo y en el resto dejarlo sin definir. En computación el resto suele tener sentido sólo entre números enteros, de manera que para todos aquellos tipos enteros: *Short*, *Int*, y *Long*, se define una operación así:

```
Type mod(Type t) {
    Type ret = majorType (t);
    if ( ret == null) throw new TypeException("Cannot make mod
operations between an " + t.getName() + " and a " + getName());
    return ret;
}
```

Esta operación hace uso de la operación *majorType()* que se definirá a continuación.

6.2.3.4 *majorType(Type t)*

Esta operación comprueba si dos tipos están enlazados por alguna cadena de promociones, en caso de ser así, se devuelve el tipo que está más a la derecha de la cadena de promociones, en caso contrario se devuelve *null*.

```
class BaseType{
    ...
}
```

```

    Type majorType(Type t) {
        Type ret = implicitCast(t);
        if (ret == null) ret =
t.implicitCast(Type.typesTable.getType(getName()));
        return ret;
    }
    ...
}

```

Se define esta operación en la clase *BaseType*, al estar definida en la clase base, todos los tipos la pueden usar. Hace uso de la operación *implicitCast()*, que será generada por TyCC en base al contenido de la sección de declaración de tipos.

6.2.3.5 *cast(Type t)*

Esta operación promociona explícitamente el objeto llamador al tipo que recibe como parámetro.

```

class BaseType{
    ...

    Type cast (Type t) {
        return t;
    }
    ...
}

```

En principio es posible cualquier conversión. Así que se define en la clase *BaseType*, para que tenga un comportamiento similar en todos los tipos. Si se añadieran más tipos, y se quisiera que estos tuvieran un comportamiento distinto no habría más que redefinir esta operación dentro de su propia especificación de operaciones.

6.2.3.6 *assignment(Type t)*

Esta operación se encargará de verificar si se puede asignar el tipo que se recibe como parámetro, al objeto invocador del método.

Dado que se supone que el tipo invocador del método corresponde a una expresión izquierda (*l-value*), la única comprobación semántica que hay que realizar es que la expresión de la derecha sea promovible implícitamente a la de la derecha.

```

class BaseType{
    ...

    Type assignment(Type t) {
        Type ret = t.implicitCast(Type.typesTable.getType(getName()));
        if ( ret == null) throw new TypeException("Cannot assign a " +
t.getName() + " to a " + getName());
        return ret;
    }
    ...
}

```

Como esta operación será igual para todos los tipos, se define en la clase *BaseType*.

6.2.3.7 unary(int operator)

Esta operación se encarga de comprobar la corrección de la operación unaria '+' o '-' sobre el tipo invocador de la operación.

```
class BaseType{
...
    Type unary(int op) {
        return Type.typesTable.getType(getName());
    }
...
}
```

Dado que el lenguaje solo tiene tipos numéricos, se define una única operación en *BaseType*, que simplemente devuelve el tipo invocador de la operación.

Nótese que de nuevo hay que devolver el propio tipo llamando a la tabla de tipos. En lugar de usar la referencia *this*.

6.2.3.8 aritmetical(Type t, int operator)

Esta operación se encargará de verificar si se pueden realizar operaciones aritméticas suma, producto, resta y división, entre el invocador, y el parámetro *t*. La operación vendrá determinada por el parámetro *operator*.

Para que se puedan realizar dichas operaciones aritméticas es necesario que uno de los dos tipos sea promovible implícitamente al otro, y se debe devolver el mayor de los dos tipos.

```
class BaseType{
...
    Type aritmetical(Type t, int operator) {
        Type ret = majorType(t);
        if (ret == null) throw new TypeException("Cannot make " +
            stringOperation(operator) + " operations between an " + t.getName() + "
            and a " + getName());
        return ret;
    }
...
}
```

De nuevo, dado que todos los tipos presentarán un comportamiento idéntico ante esta operación se puede definir en la clase base.

Se ha introducido un elemento nuevo. Como se puede ver en la cadena de caracteres que informa sobre el error producido en caso de no poder realizarse la conversión, hay una llamada a un método nuevo *stringOperation()*. Este método no es un método específico de comprobación de tipos, sino una utilidad que se define para describir el *token* de un operador en forma de cadena de caracteres de forma que sea legible para el usuario. De manera que se puede implementar como un método propio de la clase *BaseType* pero a la que el comprobador tipos no da acceso externo. Es decir no podrá ser llamada por el usuario, desde la clase *Type*, sólo desde el ámbito de las clases derivadas de *BaseType*.

La forma de definir miembros puramente de la clase *BaseType*, o de otra cualquiera, es enclaustrándolos entre estos dos separadores `%{ %}`
De manera que el método *stringOperation(int tokenOperator)* se definiría como sigue:

```
class BaseType{
```

```

...
%{
String stringOperation(int operator) throws TypeException{
    switch (operator) {
        case FrogTokenTypes.PLUS : return "additive";
        case FrogTokenTypes.MINUS : return "subtractive";
        case FrogTokenTypes.DIV : return "division";
        case FrogTokenTypes.TIMES : return "multiplicative";
        case FrogTokenTypes.MOD : return "module";
        case FrogTokenTypes.AND : return "and";
        case FrogTokenTypes.OR : return "or";
        case FrogTokenTypes.LT : return "less-comparation";
        case FrogTokenTypes.GT : return "great-comparation";
        case FrogTokenTypes.EQUALS : return "equality";
        case FrogTokenTypes.GTE : return "great-equal-comparation";
        case FrogTokenTypes.LTE : return "less-equal-comparation";
        case FrogTokenTypes.NOT_EQUALS : return "not-equal-comparation";
        default: throw new TypeException ("operator " + operator + "
unknown");
    }
}
%}
...
}

```

Los *tokens* son los generados para el *scanner* que proporciona ANTLR. Se definen en forma de constantes enteras de clase dentro de un interfaz que recibe el nombre de *FrogTokenTypes*.

6.2.3.9 relational(Type t, int operator)

Esta operación se encargará de verificar si se pueden realizar operaciones relacionales, en concreto: la operación mayor que, menor que, mayor o igual que, y menor o igual que, entre el invocador, y el parámetro *t*. La operación a realizar vendrá determinada por el parámetro *operator*.

Par que se puedan realizar dichas operaciones relacionales es necesario que uno de los dos tipos sea promovible implícitamente al otro, y se debe devolver el tipo definido para las operaciones booleanas: el tipo *Short*.

```

class BaseType{
...
    Type relational(Type t, int operator) {
        if (majorType(t)== null) throw new TypeException ("Cannot make " +
stringOperation(operator) + " operations between an " + t.getName() + "
and a " + getName());
        return Type.typesTable.getType("Short");
    }
...
}

```

6.2.3.10 equality(Type t, int operator)

Esta operación se encargará de verificar si se pueden realizar operaciones de comprobación de igualdad, igual que, y distinto de, entre el invocador y el parámetro *t*. La operación a realizar vendrá determinada por el parámetro *operator*.

Para que se puedan realizar dichas operaciones de igualdad es necesario que uno de los dos tipos sea promovible implícitamente al otro, y se debe devolver el tipo definido para las operaciones booleanas: el tipo *Short*.

```
class BaseType{
    ...
    Type equality(Type t, int operator) {
        return relational(t, operator);
    }
    ...
}
```

Esto puede parecer un poco extraño, pero es que las operaciones que se realizan para comprobar los tipos en este caso son exactamente las mismas que para las operaciones relacionales. De manera que reutilizamos el código llamando al método. Una forma quizás más correcta sea definir un método común para ser llamado a las dos operaciones.

Nuevamente se define en la clase *BaseType*, pues este algoritmo es válido para todos los tipos del lenguaje *Frog*, al ser todos numéricos.

6.2.3.11 *isBoolean()*

Esta operación devolverá cierto si el tipo invocador puede ser empleado como expresión de tipo booleana, y falso en caso contrario.

Aquí se verá otro ejemplo de redefinición de métodos.

Por defecto consideraremos que los tipos no se pueden emplear como expresiones de tipo booleanas. Así que se define el método en la clase *BaseType* haciendo que devuelva *false*.

```
class BaseType{
    ...
    boolean isBoolean() {
        return false;
    }
    ...
}
```

Los tipos que se considera que pueden emplearse como expresiones de tipo booleanas serán los de naturaleza entera, es decir: *Short*, *Long* y *Int*, por lo que se redefinirá la operación para cada uno de ellos. Haciendo que devuelva *true*.

```
class Short {
    ...
    boolean isBoolean() {
        return true;
    }
    ...
}
```

Para el *Int*:

```
class Int {
    ...
    boolean isBoolean() {
        return true;
    }
    ...
}
```

```

Y finalmente para el Long:
class Long{
...
boolean isBoolean() {
    return true;
}
...
}

```

6.2.4 Generación del comprobador de tipos

Si se salva lo escrito hasta ahora en un fichero de texto *FrogTypes.txt*, y se ejecuta la siguiente línea de comandos.

```
TyCC Frogypes.txt.
```

Se genera un fichero *Type.java* conteniendo la siguiente estructura jeararquica mostrada en la Figura 16.

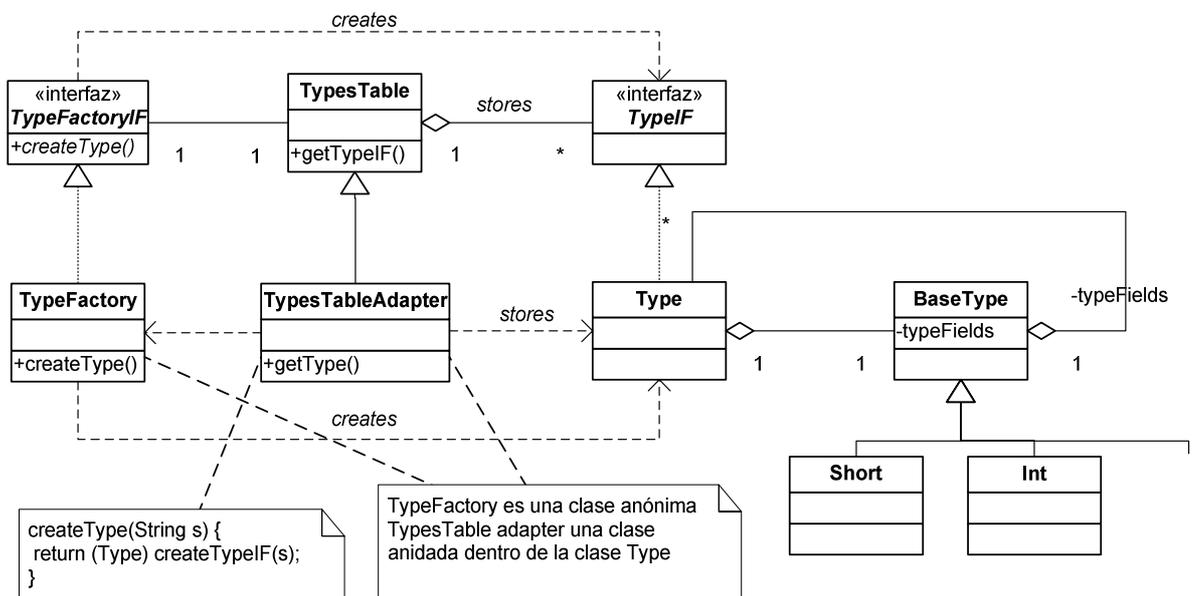


Figura 16: Diagrama de clases. Jeararquía de clases generada por TyCC para *Frog*.

Como se puede observar la clase *Type* que se va a generar implementa un interfaz *TypeIF*, esto es necesario para poder usar el API. Como la tabla de tipos reside en el API y maneja objetos tipo de clase *TypeIF*, se define una clase adaptadora de la tabla de tipos por comodidad para el usuario para poder usar directamente las clases *Type*: *TypesTableAdapter* que está anidada dentro de la clase *Type*. Además existe una variable de clase de tipo *TypesTableAdapter*, o mejor dicho *Type.TypesTableAdapter*, en la clase *Type* que se puede usar como tabla de tipos.

Para poder crear tipos es necesario instanciar un *abstractFactory* que realice la creación. Esto también lo genera TyCC definiendo una clase que implementa el interfaz *TypeFactoryIF*, que servirá para poder crear los objetos *Type*. Esta clase el usuario no la tiene que utilizar directamente, sino que será el *framework* el que haga uso de ella. Es una clase anónima, de manera que ni siquiera se tiene que tener acceso a ella. Simplemen se hace un comentario acerca de ella de carácter ilustrativo.

Ahora con más detalle, las clases que representan los tipos especificados por el usuario siguen la siguiente jerarquía mostrada en la Figura 17.

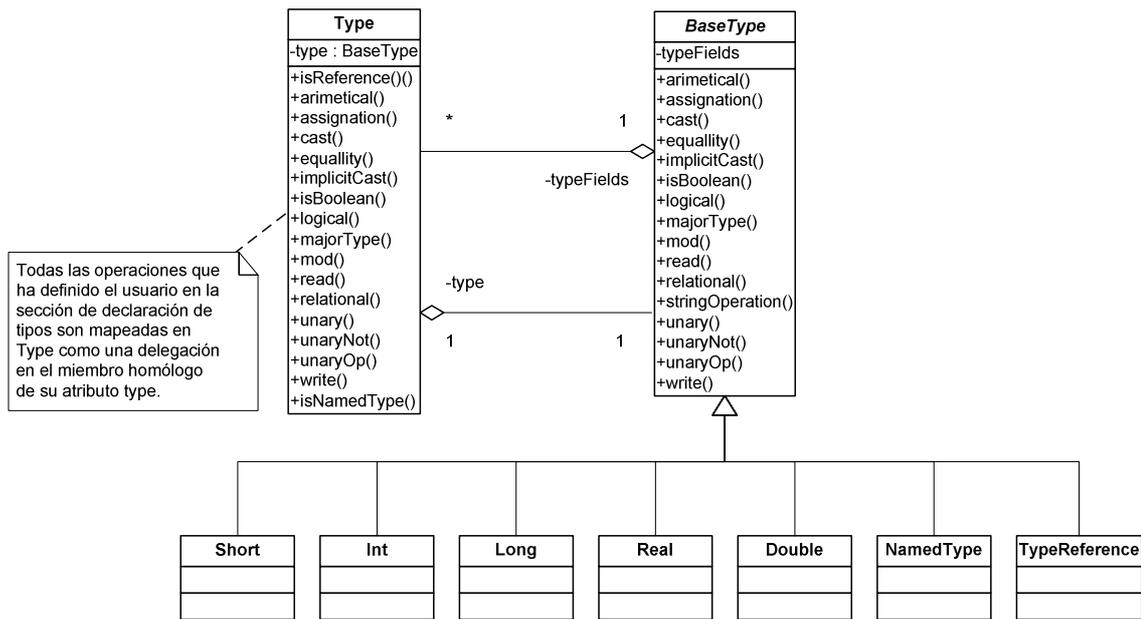


Figura 17: Diagrama de clases. Jerarquía de tipos generada por TyCC para Frog.

Como se puede observar en la figura se han creado tantas clases representantes de objetos tipo como tipos se han especificado en la sección de declaración de tipos. Además se han creado dos casos *NamedType* y *TypeReference*, que en nuestro ejemplo no tienen aplicación. Todas estas clases derivan de la clase *BaseType*. El interfaz al usuario será la clase *Type* que tienen todas las operaciones definidas en la sección de definición de tipos implementadas como una delegación en el objeto *BaseType* que envuelve.

6.2.5 API

En el caso de usar la aplicación TyCC, el funcionamiento del API es transparente al programador. Sólo se usan los métodos *getType()* y *assignName()* de la tabla de tipos. Este último si se van a utilizar expresiones de tipo con nombre, por lo que en el ejemplo se ha estado estudiando no tiene aplicabilidad.

La clase que implementa el interfaz *TEParserIF*, es por defecto *TEParser*, que es un compilador de expresiones de tipo para el lenguaje TEL: el usado en nuestro ejemplo. La clase que implementa la factoría para la creación de clases compilador, es generada por TyCC como una clase anónima, por lo que el usuario tampoco tiene acceso. La clase que define el interfaz *TypeFactoryIF*, es implementada como anónima como ya se comentó.

6.3 Comprobación de adaptabilidad

La especificación de tipos estaría completada con la definición de estas operaciones.

Usando la aplicación TyCC, se generaría un comprobador de tipos listo para utilizar con el resto del compilador ya diseñado. Para comprobar la completa adaptabilidad a los cambios se introducen dos nuevos tipos en el lenguaje ahora que el programa ya está completo.

6.3.1 Análisis léxico

Estos dos tipos serán las cadenas de caracteres y los caracteres simples, representados respectivamente por los tipos *string* y *char*.

Habrà por tanto que añadir dos palabras reservadas nuevas: *char* y *string*.

Tambièn habrà dos nuevas constantes de caracteres *STRING_LITERAL* y *CHARLIT*.

- *CHARLIT* identificarà cualquier carácter imprimible y las secuencias de escape $\backslash n$ y $\backslash t$ todo ello encerrado en comillas simples
- *STRING_LITERAL* cualquier secuencia carácter imprimible y las secuencias de escape $\backslash n$ y $\backslash t$ todo ello encerrado entre comillas dobles.

6.3.2 Análisis sintáctico

Hay que modificar levemente la gramática para que acepte dos nuevas palabras reservadas *string* y *char* y dos constantes nuevas: *CHARLIT* y *STRING_LITERAL*.

En la regla de tipo habrà que añadir los dos nuevos tipos “*string*” y “*char*”.

```
<type> ::= "short"  
        | "int"  
        | "long"  
        | "real"  
        | "double"  
        | "char"  
        | "string"  
        ;
```

Y en la de constante dos nuevas constantes *CHARLIT* y *STRING_LITERAL*.

```
<constant> ::= CONST_INT  
            | CONST_REAL  
            | CHARLIT  
            | STRING_LITERAL  
            ;
```

6.3.3 Análisis semántico

Como nombre del tipo de dato que represente los caracteres simples se usará *Char*.

Para las cadenas de caracteres se usará el identificador *TString*. Se podría usar *String*, pero entonces al usar el *String* estándar de Java para evitar colisiones de nombres habría que usar su nombre completo es decir *java.lang.String*. De esta manera el uso resulta más sencillo.

<i>Nuevos Tipos</i>	<i>Naturaleza</i>
Char	Carácter
TString	Carácter

A continuación se muestra la tabla de la semántica de operadores, para que no resulte farragoso y dado que muchas operaciones no se pueden realizar sobre los tipos, se hablarà únicamente de la semántica de las operaciones que se pueden realizar sobre alguno de los dos nuevos tipos de datos.

Nombre de la operación	Operación	Semántica	Método del Comprobador de tipos
Asignación	$v = e$	e debe ser una expresión promocionable implícitamente al tipo de la variable v . Se devuelve el tipo de la variable de v . En caso de no ser posible la asignación se alza una excepción.	<i>assignment(Type t)</i>
And lógico	$e1 \ \& \ e2$	No aplicable a caracteres.	<i>logical():Type</i>
Or lógico	$e1 \ \ e2$	No aplicable a caracteres.	<i>logical():Type</i>
Not lógico	$! \ e$	No aplicable a caracteres.	<i>unaryNot():Type</i>
Mayor	$e1 \ > \ e2$	No aplicable a caracteres.	<i>relational(...):Type</i>
Menor	$e1 \ < \ e2$	Sólo tendrá sentido si los dos tipos son caracteres. Se devuelve el tipo <i>Short</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>relational(...):Type</i>
Menor o igual	$e1 \ <= \ e2$	Sólo tendrá sentido si los dos tipos son caracteres. Se devuelve el tipo <i>Short</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>relational(...):Type</i>
Mayor o igual	$e1 \ >= \ e2$	Sólo tendrá sentido si los dos tipos son caracteres. Se devuelve el tipo <i>Short</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>relational(...):Type</i>
Igual	$e1 \ == \ e2$	Se usa la misma especificación semántica que para los números.	<i>equality(...) :Type</i>
Distinto	$e1 \ != \ e2$	Se usa la misma especificación semántica que para los números.	<i>equality(...) :Type</i>
Suma	$e1 \ + \ e2$	Sólo tiene sentido si los dos operandos son cadenas de caracteres. Se produce la concatenación de las dos cadenas de caracteres. Se devuelve el tipo <i>TString</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.	<i>aritmetical(...) :Type</i>
Resta	$e1 \ - \ e2$	No aplicable a caracteres.	<i>aritmetical(...) :Type</i>
Mas	$+ \ e$	No aplicable a caracteres.	<i>unary(...) :Type</i>
Menos	$- \ e$	No aplicable a caracteres.	<i>unary(...) :Type</i>
División	$e1 \ / \ e2$	No aplicable a caracteres.	<i>aritmetical(...):Type</i>
Producto	$e1 \ * \ e2$	No aplicable a caracteres.	<i>aritmetical(...):Type</i>
Modulo	$e1 \ \% \ e2$	No aplicable a caracteres.	<i>mod(Type t):Type</i>

Casting	(Type) e	Se permite la conversión de <i>TString</i> a <i>Char</i> y viceversa,	<i>cast(Type t):Type</i>
---------	----------	---	--------------------------

6.3.3.1 Construcción de objetos tipo

Se usaba una clase para construir adecuadamente los objetos tipo, la clase *TypeBuilder*. Esta clase es una especie de *wrapper* sobre el comprobador de tipos. Se encarga de construir objetos más específicos en los que el tipo es una sola de sus componentes.

Se deben añadir dos métodos para poder crear los dos nuevos tipos de constantes. El método *getType()* no es necesario modificarlo, pues se basa en el comprobador de tipos.

constChar() devuelve un objeto de tipo *Constant*, con el tipo *Char* asociado

```
public Constant constChar(String s) throws TypeException {
    return new Constant(new Value(s), getType("Char"));
}
```

constString() devuelve un objeto de tipo *Constant*, con el tipo *TString* asociado

```
public Constant constString(String s) throws TypeException {
    return new Constant(new Value(s), getType("TString"));
}
```

Como puede verse, hasta ahora no se ha modificado nada del código anterior, simplemente se ha añadido los métodos imprescindibles. La única clase que es necesario modificar es la clase *Value*, para que admita valores de cadenas de caracteres, pero esto sirve a fines de interpretación, no de semántica.

6.3.3.2 Implementación del módulo semántico. Comprobación de tipos específica de las sentencias

Ahora se va comprobar si es necesario añadir alguna funcionalidad extra para tratar los dos nuevos tipos, analizando primeramente la comprobación de tipos en las sentencias.

Haciendo memoria, la clase *Semantic* es una implementación de la clase *Visitor* cuyo cometido es ir realizando un recorrido en un orden determinado por los nodos del árbol sintáctico y realizar operaciones sobre cada uno de ellos.

Cuando un elemento es visitado, llama a la operación *visit* que corresponda con su clase en la clase *Visitor*, usándose así mismo como argumento, para que el *Visitor* pueda acceder a su estado.

Nuevamente se examinan todas las clases para comprobar si es necesaria alguna modificación.

6.3.3.2.1 Block

```
void visitBlock(Block b) throws TypeException{
    Iterator it = b.staments.iterator();
    Object ret = null;
    while (it.hasNext()) ((Stament)it.next()).accept(this);
}
```

En esta sentencia no se hacen referencia a ninguna expresión por lo que no es necesario añadir ni modificar nada.

6.3.3.2.2 While

```
Object visitWhile(While w) throws TypeException {
    Type cond = (Type)w.condition.accept(this);
    if (!cond.isBoolean())
        throw new TypeException("Cannot use a " + cond.getName() +
            " as a guard of a While-stament must be a boolean
expression");
    w.stament.accept(this);
    return null;
}
```

Tampoco es necesario modificar nada. Es necesario tener en cuenta que ahora si la condición es una cadena de caracteres o un carácter ya no es booleana, pero eso es cuestión de especificación de tipos. No hay nada que modificar.

6.3.3.2.3 IF

```
Object visitIF(IF i) throws TypeException {
    Type cond = (Type)i.condition.accept(this);
    if (!cond.isBoolean())
        throw new TypeException("Cannot use a " + cond.getName() +
            " as a condition of a If-stament. It must be a boolean
expression");
    i.option1.accept(this);
    if (i.option2 != null) i.option2.accept(this);
    return null;
}
```

Nada que modificar.

6.3.3.2.4 Read

```
Object visitRead(Read r) throws TypeException {
    return r.variable.accept(this);
}
```

Nada que modificar. Como los caracteres y las cadenas de caracteres son susceptibles de ser leídos por teclado, basta con comprobar que la expresión es coherente.

Quizás sea éste uno de los pocos métodos en que si se definiese un nuevo tipo, por ejemplo *void*, habría que modificar; pero también es porque *read* y *write* no deberían ser unas construcciones propias de un lenguaje.

6.3.3.2.5 Write

```
Object visitWrite(Write w) throws TypeException {
    return w.expression.accept(this);
}
```

Nada que modificar. Como los caracteres y las cadenas de caracteres son susceptibles de ser impresos por pantalla, basta con comprobar que la expresión es coherente.

Este sería otro de los métodos en que si se definiese un nuevo tipo, por ejemplo *void*, habría que modificar, habría que modificar; pero podemos aplicar el razonamiento anterior.

6.3.3.2.6 Assign

```
Object visitAssign(Assign a) throws TypeException {
    Type tv = (Type)a.variable.accept(this);
    Type te = (Type)a.expression.accept(this);
    return tv.assignment(te);
}
```

Tampoco hay que modificar nada, en todo caso la operación *assignment()*, pero nuevamente es función de la especificación de tipos.

6.3.3.2.7 VarDef

```
void visitVarDef(VarDef a) throws TypeException{
    context.insert(a.variable, new STSlot(new Value(), a.type));
}
```

Todo correcto. Nada que modificar.

Se ha comprobado que ninguna sentencia hay que modificar nada. Se han añadido dos nuevos tipos y hasta ahora sólo se han anotado dos posibles modificaciones en el fichero de especificaciones de tipos para la aplicación TyCC.

6.3.3.3 **Comprobación de tipos en las expresiones**

6.3.3.3.1 Constant

```
public Type typeChecking(Context c) throws TypeException {
    return type;
}
```

Evidentemente esto se queda como está.

6.3.3.3.2 LogicalOp

```
Object visitLogicalOp(LogicalOp l) throws TypeException {
    ((Type)l.op1.accept(this)).logical();
    return l.setType(((Type)l.op2.accept(this)).logical());
}
```

Este código no hay que tocarlo, hay que tratar la operación *logical()*, pero de nuevo se trata del fichero de entrada en la comprobación de tipos

6.3.3.3.3 AritmeticalOp

```
Object visitAritmeticalOp(AritmeticalOp b) throws TypeException {
    Type t1 = (Type)b.op1.accept(this);
    Type t2 = (Type)b.op2.accept(this);
    return b.setType(t1.aritmetical(t2, b.operator));
}
```

Este código no hay que tocarlo, hay que comprobar la operación *aritmetical()*, pero de nuevo se trata del fichero de entrada en la comprobación de tipos

6.3.3.3.4 ModOp

```
Object visitModOp(ModOp m) throws TypeException {
    Type t1 = (Type)m.op1.accept(this);
    Type t2 = (Type)m.op2.accept(this);
    return m.setType(t1.mod(t2));
}
```

Este código no hay que tocarlo, hay que comprobar la operación *mod()*, pero de nuevo se trata del fichero de entrada en la comprobación de tipos

6.3.3.3.5 RelationalOp

```
Object visitRelationalOp(RelationalOp r) throws TypeException {
    Type t1 = (Type)r.op1.accept(this);
    Type t2 = (Type)r.op2.accept(this);
    return r.setType(t1.relationalOp(t2, r.operator));
}
```

De nuevo todo se queda en revisar la operación *relational()*.

6.3.3.3.6 Equality

```
Object visitEqualityOp (EqualityOp e) throws TypeException {
    Type t1 = (Type)e.op1.accept(this);
    Type t2 = (Type)e.op2.accept(this);
    return e.setType(t1.equality(t2, e.operator));
}
```

No hay que modificar nada se anota consultar la operación *equality()*.

6.3.3.3.7 UnaryOp

```
Object visitUnaryOp(UnaryOp u) throws TypeException {
    Type t1 = (Type)u.op1.accept(this);
    return u.setType(u.operator == BNOT ? t1.unaryNot() :
t1.unary(u.operator));
}
```

Nada que modificar. Comprobar la operación *unary()* en la comprobación de tipos.

6.3.3.3.8 CastOp

```
Object visitCastOp(CastOp c) throws TypeException {
    c.expression.accept(this);
    return c.getType();
}
```

La operación *cast()* debe ser modificada, Anteriormente se permitía cualquier *cast*, pero ahora no debe ser así. No se permiten *castings* de tipos carácter a números ni viceversa.

6.3.3.3.9 Variable

Esta clase se cubren las variables como expresiones

```
Object visitVariable(Variable v) throws TypeException {
    STSlot s = context.look(v.getName());
}
```

```

        if (s == null) throw new InterpreterException("Variable " +
v.getName() + " not declared");
        return v.setType(s.type);
    }

```

Nuevamente no hay que modificar nada. Lo que se hace es comprobar que la variable esté definida.

Bien, hasta ahora no se ha tenido que modificar absolutamente nada, y se han añadido dos tipos. Sólo queda revisar una serie de operaciones en la comprobación de tipos.

Por tanto haya que modificar lo que haya que modificar se ha demostrado que utilizando nuestra herramienta se realiza una separación efectiva de incumbencias. La comprobación de tipos está aislada completamente de la fase semántica, sintáctica y léxica; excepto en lo obviamente imprescindible (adición de palabras reservadas y producciones para tratar las nuevas constantes).

6.3.4 Especificación de tipos

Hay que introducir dos nuevos tipos. Y decidir si se quiere realizar alguna operación sobre ellos.

Se ha visto que toda la modificación de código se reduce a modificar el fichero de entrada a la aplicación TyCC. Ahora se comprobará además, que la modificación es mínima y que se reduce a añadir código sin tener que desechar nada de lo implementado hasta ahora.

6.3.4.1 *Preámbulo*

Nuevamente no parece que haya que añadir ninguna cláusula *import* ni código adicional.

6.3.4.2 *Declaración de tipos.*

Habrà una promoción implícita de *Char* *TString*, así que en la sección de declaración de tipos hay que añadir *Char <TString*, es decir la sección de declaración de tipos quedaría de esta manera.

```
Types = { Short < Int < Long < Real < Double Char < TString }
```

Como podemos comprobar el nuevo tipo *Char* esta separado de la cadena de promociones de los tipos de naturaleza numérica. Esto quiere decir que no existe promoción implícita de ningún tipo numérico a los tipos para representar caracteres.

```
Types = { Short < Int < Long < Real < Double Char < TString }
```

6.3.4.3 *Definición de tipos*

No se añaden nuevas operaciones pero es necesario revisar todas las operaciones definidas hasta ahora y ver lo que hay que modificar.

6.3.4.3.1 logical()

No se pueden realizar operaciones lógicas sobre los dos nuevos dos tipos.

Como `logical()` sólo estaba definida para los tipos de naturaleza entero. Para los tipos no enteros, alza por defecto una excepción con lo que no hay que añadir nada.

6.3.4.3.2 `unaryNot()`

No se pueden realizar operaciones de negación lógica sobre los nuevos dos tipos.

Como sólo estaba definida para los tipos de naturaleza entera, por el mismo motivo que en el caso anterior no hay que añadir nada.

6.3.4.3.3 `mod(Type)`

Tampoco hay que añadir ni modificar nada, pues sólo esta definida para los tipos enteros. De manera que cualquier intento de uso hara que se alce una excepción por intento de usar operación no definida para el tipo en cuestión.

6.3.4.3.4 `majorType(Type)`

Esta operación comprueba si dos tipos están enlazados por alguna cadena de promociones, en caso de ser así se devuelve el tipo que está más a la derecha de la cadena de promociones, en caso contrario se devuelve *null*.

```
class BaseType{
...

    Type majorType(Type t) {
        Type ret = implicitCast(t);
        if (ret == null) ret =
t.implicitCast(Type.typesTable.getType(getName()));
        return ret;
    }
...
}
```

Esta operación tal como está implementada se comporta correctamente para los dos nuevos tipos y para cualquier tipo que se añada. Se queda como está..

6.3.4.3.5 `cast(Type)`

Esta operación promociona explícitamente el objeto llamador al tipo que recibe como parámetro.

```
class BaseType{
...

    Type cast (Type t) {
        return t;
    }
...
}
```

Como `cast()` estaba definida en la clase base, debemos redefinir cada uno de los métodos, o más rápido modificar este método de la siguiente manera.

```
    Type cast (Type t) {
        if (majorType(t) == null) throw new
            TypeException("Incompatible cast form " + getName() + " to " +
t.getName());
        return t;
    }
```

```
}
```

De esta manera, evitamos promociones explícitas entre tipos que no se encuentren en la misma cadena de promociones.

6.3.4.3.6 assignation(Type)

```
class BaseType{
...

    Type assignation(Type t) {
        Type ret = t.implicitCast(Type.typesTable.getType(getName()));
        if ( ret == null) throw new TypeException("Cannot assign a " +
t.getName() + " to a " + getName());
        return ret;
    }
...
}
```

Esta operación hace uso de *implicitCast()*, que se comportará correctamente con los dos nuevos tipos al ser generada de nuevo por TyCC, por lo que tampoco hay que añadir ni modificar nada.

6.3.4.3.7 unary(int operator)

Esta operación se encarga de comprobar la corrección de la operación unaria, '+' o '-', sobre el tipo invocador de la operación.

Se definía únicamente en la clase *BaseType*.

```
class BaseType{
...

    Type unary(int op) {
        return Type.typesTable.getType(getName());
    }
...
}
```

Ahora es necesario añadir código. Pues no esta operación no tiene sentido ni en caracteres ni en cadenas de caracteres.

```
class TString {
...

    Type unary(int op) {
        throw new TypeException("Operator " + stringOperation(op) + " not
available for type TString");
    }

...
}

class Char{
...

    Type unary(int op) {
        throw new TypeException("Unary operator " + stringOperation(op) + "
not available for type Char");
    }
}
```

```

    }
    ...
}

```

Se puede aducir que esta modificación es producto de una falta de previsión del programador, pues esta operación se colocó —adrede— en la clase *BaseType* suponiendo que todos los tipos iban a ser numéricos. Si se hubiera definido para cada tipo, ahora no sería necesario generar código indicar que la operación no es válida.

6.3.4.3.8 aritmetical(Type t, int operator)

Esta operación se encargará de verificar si se pueden realizar operaciones aritméticas, suma, producto, resta y división, entre los dos tipos argumento, el invocador y el parámetro. La operación vendrá determinada por el argumento *operator*.

Con los dos nuevos tipos sólo tiene sentido la suma. En los *TString* por la concatenación de cadenas y en los *Char* por que son promovibles implícitamente a *TStrings*. El resto habrá que desecharlos, así que se sobrescribe los métodos, para cada clase.

```

class TString {
    ...

    Type aritmetical(Type t, int operator) {
        if (operator != FrogTokenTypes.PLUS) throw new TypeException("With
chars and string you can only do additions");
        return t.icast(Type.typesTable.getType("TString"));
    }
    ...
}

class Char{
    ...
    Type aritmetical(Type t, int operator) {
        if (operator != FrogTokenTypes.PLUS) throw new TypeException("With
chars and string you can only do additions");
        return t.icast(Type.typesTable.getType("TString"));
    }
    ...
}

```

Ha surgido una nueva operación *icast()* que es como *implicitCast()*, pero que si no se puede realizar el ahormado, lanza una excepción informando del error.

6.3.4.3.9 icast(Type t)

Esta operación se comporta de la misma manera que *implicitCast()*, pero si no se puede realizar el ahormado, lanza una excepción informando del error.

```

class BaseType{
    ...
    Type icast(Type t){
        //Like implicit cast but it raises a type exception if no conversion is
posible
        Type ret = implicitCast(t);
        if ( ret == null) throw new TypeException("Cannot cast from
"+getName()+ " to a " + t.getName());
    }
}

```

```

    return ret;
}
...
}

```

6.3.4.3.10 relational(Type t, int operator)

Esta operación se encargará de verificar si se pueden realizar operaciones relacionales, mayor que, menor que, mayor o igual que y menor o igual que, entre los dos tipos argumento: el invocador, y el parámetro. La operación a realizar vendrá determinada por el argumento *operator*.

Tienen sentido en los tipos *Char* pues podemos comparar en base a su número ASCII.

```

class BaseType{
...
    Type relational(Type t, int operator) {
        if (majorType(t)== null) throw new TypeException ("Cannot make " +
stringOperation(operator) + " operations between an " + t.getName() + " and
a " + getName());
        return Type.typesTable.getType("Short");
    }
...
}

```

En este caso no sirve el comportamiento que proporciona por defecto *BaseType* pues en caso de querer hacer una comprobación relacional `char >= string`, al haber una promoción implícita de `char` a `string` se permitiría la operación. Se debe especificar que no está permitida para cadenas de caracteres y sí lo está para caracteres simples.

```

class Char{
...
    Type relational(Type t, int opearator) {
        t.icast(Type.typesTable.getType("Char"));
        return Type.typesTable.getType("Short");
    }
...
}

```

Para el *Char* debe comprobarse que el otro operando es de tipo *Char*.

```

class TString {
...
    Type relational(Type t, int operator) {
        throw new TypeException("With TString yo can not perform " +
stringOperation(operator) + " operations");
    }
...
}

```

Y Para el *TString* no debe permitirse realizar la operación.

6.3.4.3.11 equality(Type t, int operator)

Esta operación se encargará de verificar si se pueden realizar operaciones de comprobación de igualdad, en concreto igual que y distinto de, entre los dos tipos argumento, el invocador, y el parámetro. La operación a realizar vendrá determinada por el argumento *operator*.

Se puede realizar la operación de igualdad entre los dos nuevos tipos. Por la manera en que se había definido la operación *equality()* en *BaseType* haciendo uso de *relational()* es necesario redefinirla ahora para los *TString* y los *Char*. En los dos casos debe comprobarse que los dos operandos son caracteres.

Una forma rápida de hacer esto, es usar el método *icast()* sobre el mayor de los dos tipos.

```
class Char{
...

    Type equality(Type t, int operator) {
        t.icast(Type.typesTable.getType("TString"));
        return Type.typesTable.getType("Short");
    }
...
}
class TString{
...

    Type equality(Type t, int operator) {
        t.icast(Type.typesTable.getType("TString"));
        return Type.typesTable.getType("Short");
    }
...
}
```

6.3.4.3.12 isBoolean()

Esta operación devolverá cierto si el tipo invocador puede ser empleado como expresión de tipo booleana, y falso en caso contrario. Así que se define el método en la clase *BaseType* haciendo que devuelva *false*.

```
class BaseType{
...
    boolean isBoolean() {
        return false;
    }
...
}
```

Como se considera que los dos nuevos tipos no son susceptibles de ser aceptados como expresiones booleanas, tampoco es necesario añadir nada. Pues al no redefinir la función se llamará por defecto a la de la clase base devolviéndose *false*.

6.3.5 Conclusiones de la prueba de refactoring

Hemos añadido dos tipos y únicamente hemos tenido que redefinir, que modificar, unas cuantas operaciones en el fichero de especificación de tipos. Y los cambios mínimos y por otra parte inevitablemente necesarios en el analizador léxico, sintáctico y semántico

Se ha comprobado que haciendo un buen diseño de un compilador, usando conjuntamente TyCC y el API TyS, la adaptabilidad a los cambios es total. Se logra una separación de incumbencias total, delegando en cada etapa de construcción funcionalidades sin acoplamiento entre etapas.

6.4 Unificación de la comprobación de tipos estática y dinámica

Otra enorme ventaja del *framework* TyS, es que unifica la comprobación de tipos estática y dinámica.

Lo normal es realizar la comprobación de tipos durante la fase de análisis semántico; y si es necesario realizarla en tiempo de ejecución tener que usar otro comprobador dinámico. Para probar que TyS se puede utilizar como comprobador de tipos estático y dinámico sin realizar ningún cambio se ha construido un intérprete que utiliza las características de TyS.

Recordando el ejemplo construíamos un árbol sintáctico para poder visitarlo con dos *ConcreteElement* el analizador semántico y el intérprete. De manera que para implementar el intérprete habría que proceder igual que para el analizador semántico implementando los métodos *visit* propios de cada nodo. Dicho intérprete se ha implementado por completo y es funcional, sin embargo no se detallará aquí la forma de realizarlo, pues no es el objetivo de este proyecto. Si se desea ver el código fuente completo se puede consultar en el apéndice D.1.6.

A modo de ejemplo se muestra como se realizaría la instrucción *if*

```
Object visitIF(IF i) throws TypeException {
    if (((Value)i.condition.accept(this)).dValue() != 0.0)
i.option1.accept(this);
    else if (i.option2 != null) i.option2.accept(this);
    return null;
}
```

El esquema es bien sencillo se debe visitar cada nodo hijo, que en este caso devolverá objetos *Value*, conteniendo el resultado la evaluación. Si la condición es correcta se visita la sentencia correspondiente al *then*, caso de ser falsa y de existir componente *else* se visita esta componente.

Pero en este punto sólo se mostrarán aquellos métodos donde se haga uso del comprobador de tipos dinámico.

6.4.1 Sentencia Write

La sentencia *Write* dependerá del tipo la expresión que se va a imprimir. Se podría delegar esta responsabilidad en la clase *Value*, pero en vez de eso se puede aprovechar que se tiene un comprobador de tipos que puede llamar operaciones en función del tipo asociado.

Por ello el código de esta sentencia puede simplificarse de esta manera.

```
Object visitWrite(Write w) throws TypeException {
    w.expression.getType().write((Value)w.expression.accept(this));
    return null;
}
```

Lo único que hace falta es añadir una operación al fichero de especificación de tipos la operación *write(Value)* que imprime por pantalla el valor de un *Value* en función del tipo asociado a este.

Se define para todos los tipos, pero se hará una simplificación: los tipos de naturaleza entera. se escriben todos con el mismo formato por pantalla, y son los más numerosos, así que se define el comportamiento por defecto de *write* como escritura por pantalla de un número entero.

```
class BaseType{
...
    void write(Value v) {
        System.out.print((long)v.dValue());
    }
...
}
```

Después hace falta redefinirla en cada uno de los tipos no enteros.

```
class Double {
...
    void write (Value v) {
        System.out.print(v.dValue());
    }
...
}

class Real {
...
    void write (Value v) {
        System.out.print((float)v.dValue());
    }
...
}

class TString {
...
    void write(Value v) {
        System.out.print(v.sValue());
    }
...
}

class Char{
...
    void write(Value v) {
        System.out.print(v.sValue());
    }
...
}
```

6.4.2 Sentencia Read

La sentencia *read* dependerá del tipo la expresión a leer. De nuevo podemos aprovechar que tenemos un comprobador de tipos que puede llamar operaciones en función del tipo asociado.

Hay que añadir una operación al fichero de especificación de tipos la operación *read(Value)* lee por pantalla el tipo asociado al *Value* que recibe como argumento. Se definirá para todos los tipos.

```
class Short {
...
    void read (Value v) {
        v.sValue(ConsoleReader.readShort());
    }
...
}
class Int {
...
    void read (Value v) {
        v.sValue(ConsoleReader.readInt());
    }
...
}
class Long {
...
    void read (Value v) {
        v.sValue(ConsoleReader.readLong());
    }
...
}
class Double {
...
    void read (Value v) {
        v.sValue(ConsoleReader.readDouble());
    }
...
}

class Real {
...
    void read (Value v) {
        v.sValue(ConsoleReader.readReal());
    }
...
}

class TString {
...
    void read (Value v) {
        v.sValue(ConsoleReader.readStringSafe());
    }
...
}

class Char{
...
    void read (Value v) {
        v.sValue(ConsoleReader.readChar());
    }
...
}
```

Todo se reduce en delegar la lectura en un método estático de clase apropiado para cada tipo.

La clase *ConsoleReader* sirve para leer estos cuatro tipos de datos de teclado. En Java la lectura por teclado es un poco tediosa, y esta clase lo simplifica. Si se desea ver la implementación se puede consultar el apéndice D.1.9.

6.4.3 Operaciones aritméticas

Cada vez que se evalúan las operaciones aritméticas se debe devolver un objeto *Value* con el valor correspondiente a la evaluación.

Se supone que el semántico ya ha comprobado la validez de cada expresión. El intérprete evalúa cada expresión aplicando el operador apropiado y devuelve el valor calculado en un *double* sin embargo ahora se debe ahormar en función del tipo a devolver calculado por el semántico.

```

Object visitAritmeticalOp(AritmeticalOp b) throws TypeException {
    Value v1 = (Value)b.op1.accept(this);
    Value v2 = (Value)b.op2.accept(this);
    if (b.getType().getName().equals("TString"))
        return new Value(v1.sValue() + v2.sValue());
    double d1 = v1.dValue();
    double d2 = v2.dValue();
    double ret = d1;
    switch (b.operator) {
        case PLUS:
            ret += d2;
            break;
        case MINUS:
            ret -= d2;
            break;
        case TIMES:
            ret *= d2;
            break;
        case DIV:
            ret /= d2;
            break;
        default:
            throw new TypeException("Error operator " + b.operator);
    }
    return b.getType().castValue(new Value(ret)); //TypeChecking
}
interpreter interaction
}

```

Para ello se define una nueva operación *castValue(Value)* que ahorme el valor asociado al *Value* en función del tipo devuelto por la operación. Esta operación debe definirse para cada tipo.

```

class Short {
    ...
    Value castValue(Value v) {
        return new Value((short)v.dValue());
    }
    ...
}
class Int {
    ...
    Value castValue(Value v) {
        return new Value((int)v.dValue());
    }
    ...
}

```

```

}
class Long {
...
    Value castValue(Value v) {
        return new Value((long)v.dValue());
    }
...
}
class Double {
...
    Value castValue(Value v) {
        return new Value((double)v.dValue());
    }
...
}

class Real {
...
    Value castValue(Value v) {
        return new Value((float)v.dValue());
    }
...
}

```

6.4.4 Operaciones módulo (resto)

Al igual que con las operaciones aritméticas es necesario realizar un ahormado del valor correspondiente a la evaluación al tipo que ha especificado el semántico.

```

Object visitModOp(ModOp m) throws TypeException {
    //TypeChecking interpreter interaction
    long op1 = (long)((Value)m.op1.accept(this)).dValue();
    long op2 = (long)((Value)m.op2.accept(this)).dValue();
    return m.getType().castValue(new Value(op1 % op2));
}

```

Se hace uso de nuevo de la nueva operación *castValue(Value)*

6.4.5 Operaciones lógicas

Una vez evaluados los operandos de una operación lógica se debe devolver un objeto *Value* con el valor correspondiente a la evaluación de la operación. Como siempre debe devolver un valor de tipo *Short* no es necesario hacer uso del comprobador de tipos para hacer el ahormado anterior

6.4.6 Operaciones relacionales

Una vez evaluados los operandos de una operación relacional se debe devolver un objeto *Value* con el valor correspondiente a la evaluación de la operación. Para poder evaluar es necesario distinguir si una operación es un de tipo carácter o un número.

¡Ojo!, esto se podría resolver preguntando el nombre de cada tipo, pero se pretende demostrar la facilidad de uso que da el comprobador de tipos dinámico. De manera que se definen dos nuevas operación *isChar()*, e *isNumber()* que informan si un tipo es de tipo carácter, o numérico. Por omisión el valor devuelto será false.

```
class BaseType{
...
    boolean isChar() {
        return false;
    }
    boolean isNumber() {
        return false;
    }
...
}
```

Y luego según cada tipo.

```
class Short{
...
    boolean isChar() {
        return false;
    }
    boolean isNumber() {
        return true;
    }
...
}
class Int{
...
    boolean isChar() {
        return false;
    }
    boolean isNumber() {
        return true;
    }
...
}
class Long{
...
    boolean isChar() {
        return false;
    }
    boolean isNumber() {
        return true;
    }
...
}
class Float{
...
    boolean isChar() {
        return false;
    }
    boolean isNumber() {
        return true;
    }
...
}
class Double{
...
    boolean isChar() {
        return false;
    }
    boolean isNumber() {
        return true;
    }
}
```

```

...
}
class Char{
...
    boolean isChar() {
        return true;
    }
    boolean isNumber() {
        return false;
    }
...
}
class TString{
...
    boolean isChar() {
        return true;
    }
    boolean isNumber() {
        return false;
    }
...
}

```

De manera que ahora se puede usar de la siguiente manera en la interpretación.

```

Object visitRelationalOp(RelationalOp r) throws TypeException {
    Value v1 = (Value)r.op1.accept(this);
    Value v2 = (Value)r.op2.accept(this);
    int ret = 0;
    double d1 = v1.dValue();
    double d2 = v2.dValue();

    if (r.getType().isChar()) {
        d1 = v1.sValue().charAt(0);
        d2 = v2.sValue().charAt(0);
    }

    switch (r.operator) {
        case LT:
            ret = d1 < d2 ? 1 : 0;
            break;
        case GT:
            ret = d1 > d2 ? 1 : 0;
            break;
        case EQUALS:
            ret = d1 == d2 ? 1 : 0;
            break;
        case NOT_EQUALS:
            ret = d1 != d2 ? 1 : 0;
            break;
        case GTE:
            ret = d1 >= d2 ? 1 : 0;
            break;
        case LTE:
            ret = d1 <= d2 ? 1 : 0;
            break;
        default:
            throw new TypeException("Error operator " +
r.operator);
    }
    return new Value(ret);
}

```

```
}
```

Si se trata de un número se opera de una manera y si se trata de un tipo carácter de otra. Sólo se interpretan los casos posibles, dado que se supone que el semántico se ha encargado de validar todas las expresiones.

En cuanto al valor de retorno, como siempre debe devolver un valor de tipo *Short*, no es necesario hacer uso del comprobador de tipos para ahorrar a un valor determinado.

6.4.7 Operaciones de igualdad

Una vez evaluados los operandos de una operación relacional se debe devolver un objeto *Value* con el valor correspondiente a la evaluación de la operación. Para poder evaluar es necesario distinguir si una operación es un de tipo carácter o un número. De nuevo se usa *isNumber()* que informan si un tipo es numérico.

```
Object visitEquallityOp (EquallityOp e) throws TypeException {

    Value v1 = (Value)e.op1.accept(this);
    Value v2 = (Value)e.op2.accept(this);
    int ret = 0;

    if (e.getType().isNumber()) {
        double d1 = v1.dValue();
        double d2 = v2.dValue();
        switch (e.operator) {
            case EQUALS:
                ret = d1 == d2 ? 1 : 0;
                break;
            case NOT_EQUALS:
                ret = d1 != d2 ? 1 : 0;
                break;
            default:
                throw new TypeException("Error operator " +
                    e.operator);
        }
    }
    else {
        String s1 = v1.sValue();
        String s2 = v2.sValue();
        switch (e.operator) {
            case EQUALS:
                ret = s1.equals(s2) ? 1 : 0;
                break;
            case NOT_EQUALS:
                ret = s1.equals(s2) ? 0 : 1;
                break;
            default:
                throw new TypeException("Error operator " +
                    e.operator);
        }
    }
    return new Value(ret); //TypeChecking interpreter interaction
}
```

En cuanto al valor de retorno, como siempre debe devolver un valor de tipo *Short*, no es necesario hacer uso del comprobador de tipos para ahorrar a un valor determinado.

6.4.8 Operaciones unarias

Las operaciones unarias, +, -, ! son aplicables únicamente a números y de nuevo después de la evaluación hay que ahormar del valor evaluado al tipo que espera el semántico.

```
Object visitUnaryOp(UnaryOp u) throws TypeException {
    double d1 = ((Value)u.op1.accept(this)).dValue();
    double ret = 0.0;
    switch (u.operator) {
        case BNOT:
            ret = d1 == 0 ? 1 : 0;
            break;
        case MINUS:
            ret = -d1;
            break;
        case PLUS:
            ret = d1;
            break;
        default:
            throw new TypeException("Error operator " +
                u.operator);
    }
    return u.getType().castValue(new Value(ret));
}
```

De nuevo se hará uso de la operación *castValue (Value)*

6.4.9 Operaciones de cast explícito

En el caso de las operaciones de *cast* explícito se debe evaluar la expresión y después aplicarle el operador de *cast*. Para ello se hace nuevamente uso de la operación *castValue()*.

```
Object visitCastOp(CastOp c) throws TypeException {
    Value v = (Value)c.expression.accept(this);
    //Example of direc accoplation between Interpreter & Typechecking
    return c.getType().castValue(v);
}
```

Hasta ahora no se había implementado para el tipo *Char* ni el *TString*.

Par el tipo *Char* se debe truncar la cadena al primer carácter.

```
class Char{
    ...
    Value castValue(Value v) {
        return new Value(" " +v.sValue().charAt(0));
    }
    ...
}
```

Y en las cadenas de caracteres dejarlo como está.

```
class TString{
    ...
    Value castValue(Value v) {
        return new Value (v.sValue());
    }
}
```

```

...
}

```

6.5 Resultado de la refactorización

El diagrama simplificado de la jerarquía de clases que el *framework* forma para *Frog* es el mismo que se mostraba en la Figura 16.

La nueva jerarquía de tipos se ha visto alterada por la inclusión de dos nuevos tipos. *TString* y *Char*, una nueva operación *castValue()*, y la redefinición de otras varias. Todo ello automáticamente añadido por TyCC.

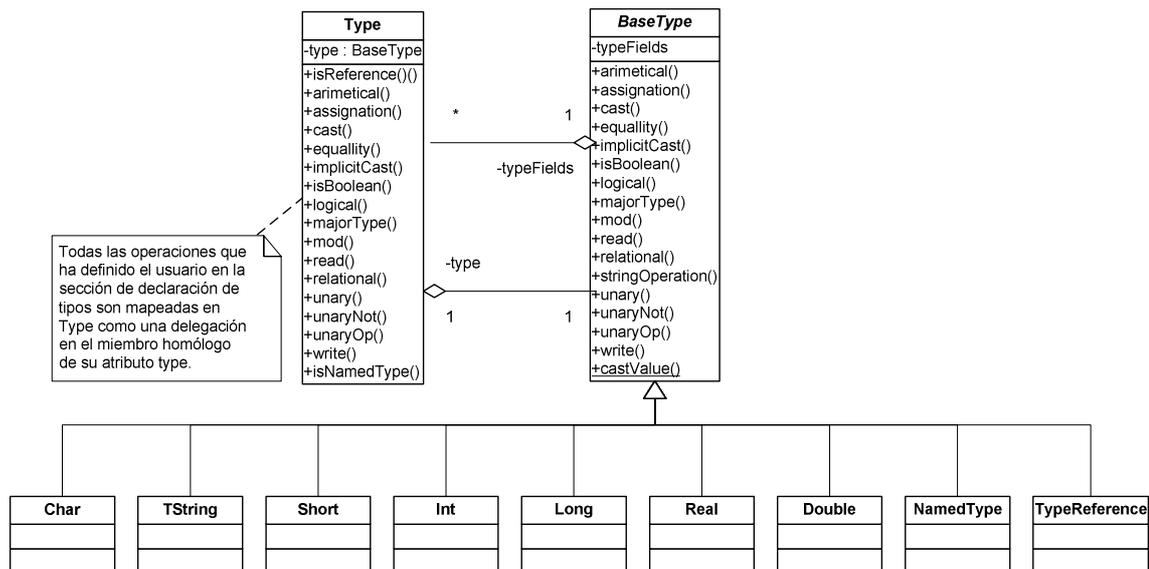


Figura 18: Diagrama de clases generado por TyCC para Frog tras añadir dos nuevos tipos.

6.6 Compilador resultante (Compiler .OBJ)

Una vez se ha llegado a este punto. Se compilan todos los módulos y se enlazan para obtener el código objeto de nuestro compilador.

Normalmente no es necesario recompilar un API cuando se usa. En el caso de la implementación en Java de TyS esto es necesario debido al manejador de errores.

Por defecto *TypeException* no tiene clase base, pero se da la oportunidad al usuario de definirle una clase base, por lo que es necesario recompilar la parte del manejador de errores.

6.7 Uso avanzado. Aplicaciones reflectivas

Dado que se unifica la comprobación de tipos estática y dinámica, se puede usar el *framework* TyS para la construcción de sistemas reflectivos.

Reflectividad o reflexión (reflection) es la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo, así como ajustar su comportamiento en función de ciertas condiciones [Maes87].

El dominio computacional de un sistema reflectivo –el conjunto de información que computa– añade al dominio de un sistema convencional la estructura y comportamiento de sí

mismo. Se trata pues de un sistema capaz de acceder, analizar y modificarse a sí mismo, como si de una serie de estructuras de datos propias de un programa de usuario se tratase.

Un ejemplo de un sistema completamente reflectivo a todos los niveles, estructural, lingüística y computacional es *nitrO* como se puede ver en [Ortín2001].

Se demuestra que esto se puede realizar, utilizando un compilador similar al anterior, pero al que se incorpora el análisis semántico en tiempo de ejecución. Como se da acceso a la tabla de símbolos y tipos en tiempo de ejecución y el *framework* TyS crea los tipos siempre dinámicamente proporciona mecanismos muy útiles para la reflectividad estructural.

7 INTERACCIÓN CON YACC

TyS se integra perfectamente con cualquier herramienta existente de ayuda al análisis. Con las más avanzadas como ANTLR, se acopla sin tener que hacer nada más que hacer que *TypeException* derive de *antlr.ParserException*.

Sin embargo con herramientas menos avanzadas, como BYACCJ, se conecta sin problemas, pero su uso puede ser un poco farragoso.

Con objeto de mostrar la usabilidad de las herramientas diseñadas se diseña una herramienta que mejora la interacción de la herramienta estándar en el análisis sintáctico: YACC, con nuestro conjunto de utilidades.

El *framework* se puede usar con YACC sin necesidad de esta herramienta. El objetivo de *YACCPT* (*YACC Preprocesor for TyS*), es mostrar más claramente cómo gracias a TyS se puede diferenciar claramente el chequeo de tipos del resto de acciones semánticas.

Lo que hace *YACCPT*, Es permitir especificar directamente acciones que conllevan chequeo de tipos, sin alterar la sintaxis de *YACC*. De esta forma se logra la separación de incumbencias en las acciones mencionada en el párrafo anterior.

7.1 Invocación desde la línea de comandos

YACCPT es un programa de consola. Hay dos formas de invocarlo:

```
YACCpt <fichero de entrada>
YACCpt <fichero de entrada> <fichero de salida>
```

En el primero de los casos si el fichero de entrada tiene como extensión “.yt” el fichero de salida se llamará igual que el primero pero cambiando su extensión por “.y”. De no ser así se llamará igual que el de entrada añadiendo un “.y” al final. En el segundo, el fichero de salida se llamará como indique el segundo parámetro, siempre que no coincida con el primero.

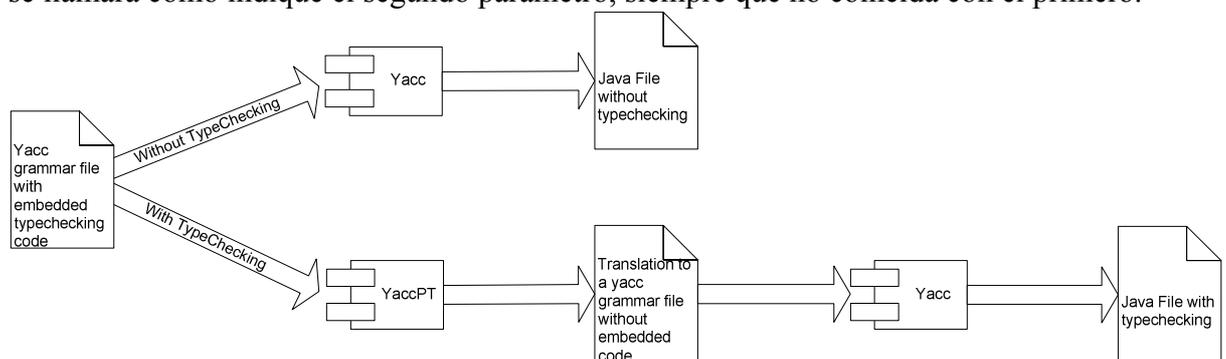


Figura 19: Uso de YACCPT

7.2 Escritura del fichero de entrada

El formato del fichero de entrada es igual que el de un fichero *YACC*.

Se añade al valor semántico que se arrastra en el análisis ascendente un campo *type* de clase *Type* generada por TyCC.

Para poder independizar la comprobación de tipos del resto análisis semántico se pueden encerrar el código correspondiente a comprobaciones entre comentarios con una marca especial.

En medio de una acción si YACCPT encuentra un comentario estilo C seguido de “<” eliminará el comentario y generará código para poder ser usado en BYACC. Es decir :

```
/*<Código Java de comprobación de tipos >*/ Se transformará
```

después de su preproceso en código que YACC podrá preprocesar normalmente.

Lo único que el usuario tiene que añadir para que todo sea correcto es un atributo de clase *Type.TypesTableAdapter* a la clase *Parser* que generará BYACC. También código para su construcción. Por ejemplo:

```
Type.TypesTableAdapter typesTable = Type.typesTableAdapter;  
(el constructor de TypesTable no lleva argumentos).
```

No obstante esto no es necesario, puesto que la clase *Type* contiene una variable de clase de tipo *Type.TypesTableAdapter* por lo que se puede usar ese atributo como tabla de tipos.

En el ejemplo de uso, que mostraremos a continuación, para ahorrar se ha asignado una referencia a esta tabla de tipos.

```
Type.TypesTableAdapter typesTable = Type.typesTable();
```

7.3 Interacción con BYACCJ

Aparte de lo dicho anteriormente. Si se desea que un mismo fichero de entrada pueda usarse con BYACCJ sin pasar por YACCPT. Deben usarse como valor semántico el que nosotros hemos ampliado para añadirle el atributo *type*.

La opción debe ser la siguiente

```
-Jsemantic = semanticValue
```

Por ejemplo para un fichero *Parser.yt* sería

```
YACC -Jsemantic = semanticValue Parser.yt
```

7.4 Ejemplo de uso

Se ha diseñado un intérprete de una calculadora simple con comprobación de tipos. Si la calculadora se preprocesa con YACCPT, existe comprobación de tipos, sino, se comporta como un intérprete sin comprobación de tipos.

En el apéndice D.3 se encuentra el listado completo de la especificación de tipos y de la entrada a YACCPT.

8 INTERACCIÓN CON ANTLR. COMPROBACIÓN DE TIPOS DE UN LENGUAJE OO

TyS se integra perfectamente con cualquier herramienta existente de ayuda al análisis. Con las más avanzadas como ANTLR, se acopla sin tener que hacer nada más que hacer que *TypeException* derive de *antlr.ParserException*.

Usando TyCC esto se reduce a incluir la siguiente opción de configuración:

```
TyCC ... -baseError=antlr.ParserException fichero_tipos
```

Como ejemplo final de la potencia de TyS y de su interacción con ANTLR se diseña un analizador semántico para un lenguaje OO: el lenguaje *Drill*. A continuación se muestra el diseño del lenguaje.

8.1 Análisis léxico

Los lexemas admitidos por el lenguaje *Drill* son los siguientes:

```
IDENT ::= (a..z)(A..Z| a..z) *
TYPE ::= (A..Z)(A..Z| a..z) *
CONST_INT ::= (0..9)+
CONST_REAL ::= CONST_INT '.' CONST_INT+
ESCAPE ::= ' ( 'n' | 'r' | 't' | '"' | '\'|'\\' )
CHARLIT ::= '\\ ' (~('\\'|ESCAPE) '\\ '
STRING_LITERAL ::= '"' ( '"' '"' | ~('\\'|'n'|'r'|'\\')|ESCAPE) * '"'
```

Operadores

- *Identificadores*: cualquier secuencia de letras del alfabeto inglés empezando por una letra minúscula (no sirve la ñ).
- *Tipos de datos*: cualquier secuencia de letras del alfabeto inglés empezando por una letra mayúscula.
- *Constantes enteras*: cualquier secuencia de dígitos. En el léxico suele ser buena idea no considerar el signo de los números
- *Constantes reales*: números reales sin signo. Secuencias de dígitos con un punto por el medio.
- *Constantes de carácter*: cualquier letra entrecomillada por comillas simples, o un carácter de escape.
- *Constante cadena de caracteres*: cualquier secuencia de caracteres imprimibles entrecomillados por comillas dobles.
- *Operadores*: los siguientes caracteres o grupos de caracteres: ! + - / * % > < = ; , . == () && || >= <= != == []
- *Palabras reservadas*: Char Int Real Void MethodClass ClassClass AttributeClass Bool Array CharString Base typedef return class public protected private this super true false null

A continuación se muestra una tabla con el mapeo de tokens en componentes léxicas.

<i>Lexema</i>	<i>Token</i>
Fin de fichero	EOF
Identificador	IDENT
Constante entera	CONST_INT

Constante real	CONST_REAL
!	BNOT
.	DOT
=	ASSIGN
:	COLON
;	SEMI
,	COMMA
==	EQUALS
(LPAREN
)	RPAREN
!=	NOT_EQUALS
<	LT
<=	LTE
>	GT
>=	GTE
+	PLUS
-	MINUS
*	TIMES
/	DIV
{	LBRACE
}	RBRACE
	OR
&	AND
%	MOD

8.2 Sintaxis de Drill

En estos puntos se detallan aspectos sintácticos del lenguaje *Drill*.

8.2.1 Uso y ubicación de operadores

A continuación se detalla el uso sintáctico de cada operador

Operador	Nombre	Descripción
.	Selección	Accede al método o atributo de una variable de tipo clase.
[]	Indexación	Accede a una componente de un variable de tipo Array
,	Secuencia	Secuencia de expresiones,
=	Asignación	Asignación, su resultado no es una expresión por lo que no se puede usar en otras expresiones ni en conjunción con otros operadores
&	And lógico	Realiza la operación <i>and</i> lógica sobre dos expresiones. Su resultado es otra expresión.
	Or lógico	Realiza la operación <i>or</i> lógica sobre dos expresiones. Su resultado es otra expresión
!	Not lógico	Realiza la operación <i>not</i> o negación lógica sobre una expresión

<i>Operador</i>	<i>Nombre</i>	<i>Descripción</i>
		Su resultado es otra expresión
<	Mayor	Realiza la operación de comparación “menor que” entre dos expresiones. Su resultado será una expresión indicando este resultado.
>	Menor	Realiza la operación de comparación “mayor que” entre dos expresiones. Su resultado será una expresión indicando este resultado.
<=	Menor o igual	Realiza la operación de comparación “menor o igual que” entre dos expresiones. Su resultado será una expresión indicando este resultado.
>=	Mayor o igual	Realiza la operación de comparación “mayor o igual que” entre dos expresiones. Su resultado será una expresión indicando este resultado.
==	Igual	Realiza la operación de comparación de igualdad entre dos expresiones. Su resultado será una expresión indicando este resultado.
!=	Distinto	Realiza la operación de comparación que indica si dos expresiones son distintas. Su resultado será una expresión indicando este resultado.
+	Suma	Realiza la suma entre dos expresiones devolviendo la expresión resultante.
-	Resta	Realiza la resta entre dos expresiones devolviendo la expresión resultante.
+	Mas	No tiene efecto. Devuelve la misma expresión.
-	Menos	Cambia el signo a una expresión. Y devuelve la expresión resultante.
/	División	Realiza la división entre dos expresiones devolviendo la expresión resultante.
*	Producto	Realiza el producto entre dos expresiones devolviendo la expresión resultante.
%	Modulo	Realiza el módulo (resto) entre dos expresiones devolviendo la expresión resultante.
()	Agrupado	Agrupar expresiones, para interferir en el orden de evaluación. El resultado de la agrupación es la expresión que se evalúa entre los dos paréntesis
(Type)	Casting	Cambia el tipo de una expresión, su resultado es una expresión con el tipo resultante.
(, . . . ,)	Función	Llama al método correspondiente con los argumentos que encierra.

A continuación se muestra la tabla de precedencias y asociatividades. También se indica la ubicación del operador, indicando si es infija, prefija, postfija, o si se coloca en algún lugar especial.

Los lugares más altos de la tabla muestran las precedencias más pequeñas, es decir a según vamos bajando en la tabla va creciendo la precedencia de los operadores.

Operador	Aridad	Asociatividad	Ubicación
,	2	Izquierda	Infija
=	2	-	Infija
&	2	Izquierda	Infija
= ==	2	Izquierda	Infija
< > <= >=	2	Izquierda	Infija
+ -	2	Izquierda	Infija
/ * %	2	Izquierda	Infija
+ -	1	Izquierda	Prefija
!	1	Izquierda	Prefija
() casting	2	Izquierda	Encerrando la expresión de tipo y delante de la expresión a ahorrar
() función	indefinida	-	Postfija, u encerrando los argumentos.
[]	2	Izquierda	Postfija, y encerrando el índice del arreglo
.	2	Izquierda	Infija
() agrupado	1	-	Encerrando las expresiones

8.2.2 Gramática de Drill

A continuación se presenta la gramática completa del lenguaje OO que se ha inventado.

$program \rightarrow (typeDefinition)^+ (methodDefinition)^* EOF$

$typeDefinition \rightarrow classDefinition$
 $|\text{"typedef" TYPE SEMI!}$

$classDefinition \rightarrow \text{"class" TYPE}$
 $(scope TYPE)? LBRA CE (methodOrAttribute)^* RBRACE$

$scope \rightarrow \text{"public"}$
 $|\text{"private"}$
 $|\text{"protected"}$

$methodOrAttribute \rightarrow type IDENT$
 $($
 $(LPAREN (type (IDENT)? (COMMA type (IDENT)?)^*)? RPAREN)$
 $|$
 $)$
 $SEMI$

$type \rightarrow TYPE (LBRACKET RBRACKET)^*$

statement → "return" *expression SEMI*
 | *expression SEMI*
 | *block*
 | *type IDENT (COMMA IDENT) * SEMI*

expressionList → *expression (COMMA expression) **

argList → *expressionList*
 | λ

expression → *assignmentExpression*

assignmentExpression → *conditionalExpression(ASSIGN assignmentExpression) ?*

conditionalExpression → *equalityExpression ((OR | AND) equalityExpression) **

equalityExpression → *relationalExpression*
 ((NOT_EQUALS | EQUALS) *relationalExpression*) *

relationalExpression → *additiveExpression*
 ((LT | LTE | GT | GTE) *additiveExpression*) *

additiveExpression → *multiplicativeExpression*
 ((PLUS | MINUS) *multiplicativeExpression*) *

multiplicativeExpression → *unaryExpression*
 ((TIMES | DIV | MOD) *unaryExpression*) *

unaryExpression → (MINUS | PLUS) *unaryExpression* }
 | *unaryExpressionNotPlusMinus*

unaryExpressionNotPlusMinus → BNOT *unaryExpression*
 | LPAREN *type RPAREN unaryExpressionNotPlusMinus*
 | *postfixExpression*

postfixExpression → *primaryExpression (DOT (IDENT*
 | "this"
 | "super"
))
 | LPAREN *argList RPAREN*
 | LBRACKET *expression RBRACKET*
) *

primaryExpression → *IDENT*
 | *constant*
 | "true"
 | "false"
 | "this"
 | "null"
 | LPAREN *assignmentExpression RPAREN*

| "super"

constant → *CONST_INT*
| *CONST_REAL*
| *CHARLIT*
| *STRING_LITERAL*

methodDefinition → *type TYPE DOT IDENT*
LPAREN
(type IDENT (COMMA type IDENT))?*
RPAREN
block

block → *LBRACE (stament)* RBRACE*

8.3 Análisis semántico

En los puntos siguientes se detalla la semántica propia de las construcciones del lenguaje *Drill*.

8.3.1 Definición de tipos de datos (Constructores de tipos)

En *Drill* únicamente se definen tipos de datos. Estos tipos pueden ser o nombres de tipos, al estilo de los *typedefs* de C, clases o arreglos de tipos de datos.

Los nombres de tipo únicamente se comportan como un alias del tipo al que dan un nuevo nombre.

Las clases están formadas por cero o más miembros de clase. Cada miembro de clase tiene una visibilidad que puede ser pública protegida y privada, con la misma semántica que en C++ [Stroustrup94b]. Los miembros de clase pueden ser de dos tipos: *atributos* o *métodos*.

Los *atributos* son variables que pueden ser de cualquier tipo; es decir de tipo definido o los predefinidos. El acceso a una variable miembro dentro del ámbito de la propia clase que la define se hace mediante el nombre de la variable. Si se trata de una variable miembro perteneciente a otra clase se usa el operador de selección de miembro. El ‘.’ Al igual que en Java y C++.

El acceso a un **método** es igual que el acceso a los variables miembro. Se escribe el nombre del método seguido por la signatura. Una llamada a un método será correcta si el número parámetros empleados en la llamada coincide con el indicado en la definición del método. Además el tipo de cada parámetro debe ser el mismo o promovible implícitamente al indicado en la definición del método.

Todas las clases siempre tienen la variable miembro *this*, que representa la instancia de la clase que define el método donde se está empleando *this*.

Si es una clase hija. Es decir que hereda de otra clase existe otra variable miembro al estilo de *this*, pero que se refiere a la clase base o padre. Se trata de la variable miembro *super*.

8.3.2 Tipos predefinidos

El lenguaje define una serie de tipos predefinidos y unas operaciones básicas para ellos.

- *Void*: se usa para la construcción de métodos procedimiento. Es decir funciones que no devuelven nada.
- *Int*: para representar el conjunto de números enteros. Sobre este tipo se pueden realizar todas las operaciones lógicas y aritméticas (incluida la operación resto), y de comparación.
- *Real*: para representar el conjunto de números reales. Sobre este tipo se pueden realizar todas las operaciones aritméticas menos la operación resto, todas las de comparación, y algunas lógicas.
- *Char*: para representar caracteres ASCII. Como este tipo de dato tiene promoción implícita a tipo *Int*, se pueden realizar sobre él todas las operaciones que se pueden realizar sobre los enteros.
- *Bool*: para representar los resultados de las operaciones lógicas. Se definen dos valores de verdad: *true* o *false*.
- *Base*: representa una clase a la que todas las clases se pueden promover automáticamente. Puede tomar el valor *null*.

8.3.3 Promociones implícitas

En los tipos predefinidos hay promoción implícita de tipo *Char* a tipo *Int* y de tipo *Int* a tipo *Real*.

En las clases hay promoción automática de una clase a su clase base. Y de cualquier clase al tipo *Base*.

8.3.4 Promociones explícitas (cast forzado)

Se permite cualquier tipo de coerción explícita. Es decir usando el operador de *cast* `(' type ')` *expresion* se puede promover cualquier expresión al tipo *type*.

8.3.5 Herencia

Los tipos clase pueden heredar las características de otro tipo clase. La herencia se puede realizar con tres niveles de visibilidad al igual que en C++ [Stroustrup94b].

Se puede acceder directamente a la clase base desde la clase derivada usando la variable miembro *super*. Existe una promoción automática de la clase derivada a la clase padre (*upcasting*).

Mediante un cast forzado se puede hacer un *downcasting*, pero al igual que en C++ esto no es seguro.

8.3.6 Equivalencia de expresiones de tipo

La equivalencia es puramente nominal. Es decir dos expresiones son iguales únicamente si representan exactamente el mismo tipo.

8.3.7 Semántica de operadores

<i>Tipos</i>	<i>Naturaleza</i>
Char	Carácter
Int	Entera
Real	Real
String	Carácter
Void	Procedimiento
Bool	Booleana
Clases	Objeto

<i>Nombre de la operación</i>	<i>Operación</i>	<i>Semántica</i>
Asignación	$v = e$	e debe ser una expresión promocionable implícitamente al tipo de la variable v . Se devuelve el tipo de la variable v . En caso de no ser posible la asignación se alza una excepción.
And lógico	$e1 \ \& \ e2$	$e1$ y $e2$ deben ser operandos enteros. Se devuelve el tipo <i>Bool</i> . En caso de no ser alguno de ellos enteros se alza una excepción
Or lógico	$e1 \ \ e2$	$e1$ y $e2$ deben ser operandos enteros. Se devuelve el tipo <i>Bool</i> . En caso de no ser alguno de ellos enteros se alza una excepción
Not lógico	$! \ e$	e debe ser un operando entero. Se devuelve el tipo <i>Bool</i> En caso de no ser operando entero se alza una excepción.
Mayor	$e1 \ > \ e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación mayor. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Bool</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.
Menor	$e1 \ < \ e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación menor. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Bool</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.
Menor o igual	$e1 \ <= \ e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación menor o igual. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Bool</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.

Nombre de la operación	Operación	Semántica
Mayor o igual	$e1 \geq e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación mayor o igual. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Bool</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.
Igual	$e1 == e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación de igualdad. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Bool</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.
Distinto	$e1 != e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido una comprobación distinto. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo <i>Bool</i> . Si no se cumplen estas condiciones se alza una excepción informando del error.
Suma	$e1 + e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido la suma. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo con más rango de las dos expresiones. Si no se cumplen estas condiciones se alza una excepción informando del error.
Resta	$e1 - e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido la suma. Además debe existir una promoción implícita de uno de los operandos al otro. Se devuelve el tipo con más rango de las dos expresiones. Si no se cumplen estas condiciones se alza una excepción informando del error.
Más	$+ e$	e debe ser una expresión de tipo numérico sobre la que tenga sentido aplicar el más unario. Se devuelve el tipo de e . Si no es una expresión válida se devuelve un error.
Menos	$- e$	e debe ser una expresión de tipo numérico sobre la que tenga sentido la operación el menos unario. Se devuelve el tipo de e . Si no es una expresión válida se devuelve un error.
División	$e1 / e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido la división. Además debe existir una promoción implícita de uno de los operandos al otro Se devuelve el tipo con más rango de las dos expresiones.

<i>Nombre de la operación</i>	<i>Operación</i>	<i>Semántica</i>
		Si no se cumplen estas condiciones se alza una excepción informando del error.
Producto	$e1 * e2$	$e1$ y $e2$ deben ser operandos sobre los que tenga sentido el producto. Además debe existir una promoción implícita de uno de los operandos al otro Se devuelve el tipo con más rango de las dos expresiones. Si no se cumplen estas condiciones se alza una excepción informando del error.
Modulo	$e1 \% e2$	$e1$ y $e2$ deben ser operandos enteros. Se devuelve el tipo con más rango de las dos expresiones. Si no se cumplen estas condiciones se alza una excepción informando del error.
Casting	$(Type) e$	e es una expresión de cualquier tipo, y $type$ una expresión de tipo. Se devuelve el tipo $type$. En principio se permiten todas las conversiones no se alzan errores.
Selección	$e . ID$	e debe ser una expresión de tipo clase e ID un campo o un método válido de esa clase.
Indexación	$e1 [e2]$	e debe ser una expresión de tipo array, y $e2$ un tipo de dato entero o promovible a entero.
Función	$e(e1, \dots, en)$	e debe ser una expresión de tipo método de clase, y $e1..en$ deben coincidir en número con el número de argumentos que toma la función y cada una de las expresiones ser promovibles al tipo de cada uno de los parámetros.

8.4 Especificación de tipos

El fichero completo de entrada a TyCC, se puede ver completo en el apéndice D.2.1.

Como se puede observar será necesario introducir una cláusula *import* en el preámbulo, pues vamos a hacer uso de las funciones de entrada salida.

No se harán más comentarios sobre como se ha implementado la comprobación de tipos, pues ya se ha especificado un método sistemático para la construcción de un compilador usando TyCC. Aunque en este caso el estudio de la funcionalidad se hace en base al análisis sintáctico, ya que la comprobación de tipos se hace a la par que el análisis sintáctico, es decir, insertando reglas de comprobación en las producciones de la gramática.

Se supone que con lo expuesto en este documento el lector ya tiene capacidad para comprender el compilador desarrollado y sobre todo el fichero de especificación de tipos, a pesar de tratarse del fichero de especificación más complejo de los tratados hasta ahora.

El código fuente de este compilador así como diversas pruebas A y B, se hayan tanto en el CD-ROM como en los apartados del apéndice D.2.

8.5 Jerarquía de objetos generada

Como siempre se genera una clase *Type* y un árbol de herencia coronado por la clase *BaseType* y cuyos descendientes son todos los tipos que el usuario quiere que formen parte de su lenguaje.

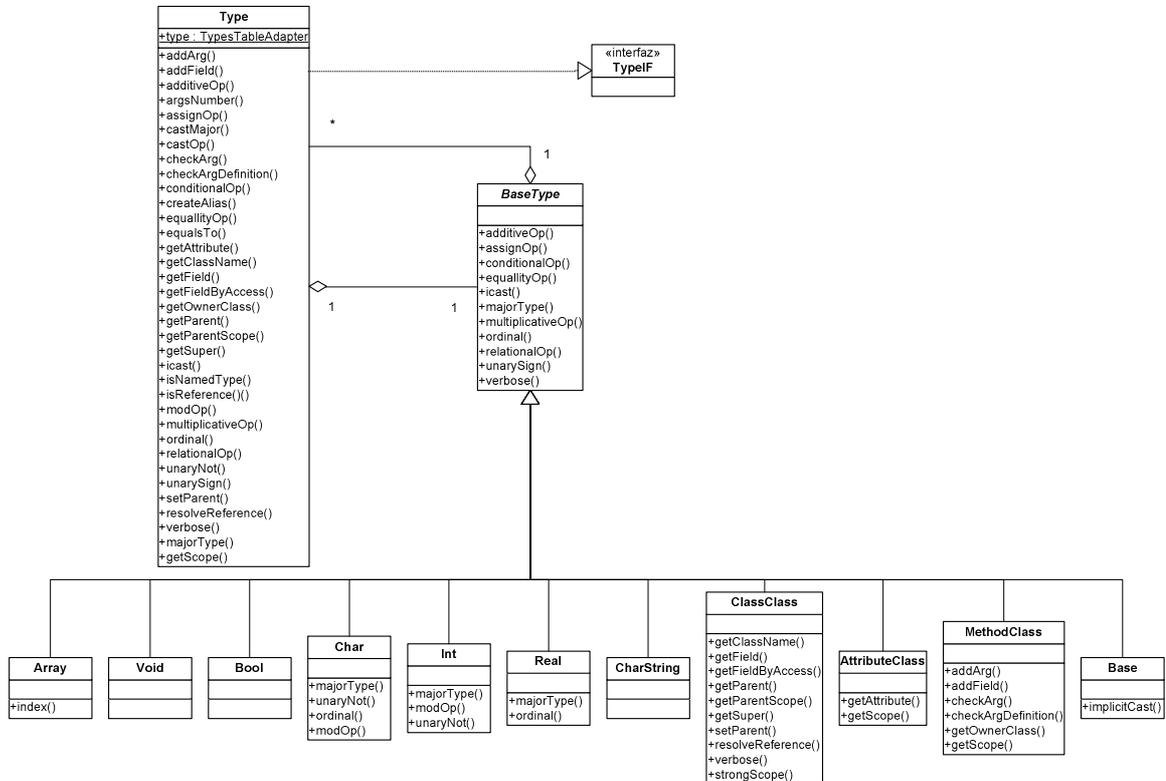


Figura 20: Diagrama de clases. Jerarquía de tipos generada por TyCC para el lenguaje *Drill*.

En la figura se pueden ver el diagrama de clases que representa la jerarquía de tipos generada por TyCC para el lenguaje *Drill*, a partir de nuestra especificación de tipos. Esta jerarquía se puede observar mejor en el CDROM en el fichero *Drill/Type.java*. Se omite, por simplicidad, en el diagrama, las clases *TypesTableAdapter* y *TypeFactory* que son generadas automáticamente por TyCC.

La clase *Type*, implementa el interfaz *TypeIF*, y la clase *TypeFactoryIF* es un *ConcreteFactory* para la clase *Type*, es implementado como una clase anónima en el código generado por TyCC, de manera que no puede ser usada por el usuario. Se utiliza para poder inicializar correctamente la tabla de tipos de clase *TypesTableAdapter*. En concreto la variable de clase *typesTable*, en la clase *Type*, que servirá como tabla de tipos de ámbito global.

8.6 Prueba del analizador semántico

En el directorio *Frog/A* y *Frog/B* se podrán encontrar pruebas A y B para nuestro analizador semántico. Para ilustrar la potencia del lenguaje creado usando TyCC y el *framework* TyS, es interesante observar el fichero *A00.txt*

9 BIBLIOGRAFÍA