CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

TRABAJO DE FIN DE GRADO

Desarrollo de un Compilador para el Lenguaje de Especificación de Eventos en Series Temporales TESL

Autor: Diego José Barberá Torralvo

Directora: Aurora Pérez Pérez

MADRID, JUNIO 2014

Resumen

Este Trabajo de Fin de Grado recoge el diseño e implementación de un compilador y una librería de entorno de ejecución para el lenguaje específico del dominio TESL, un lenguaje de alto nivel para el análisis de series temporales diseñado por un grupo de investigación de la Universidad Politécnica de Madrid. Este compilador es el primer compilador completo disponible para TESL y sirve como base para la continuación del desarrollo del lenguaje, estando ideado para permitir su adaptación a cambios en el mismo. El compilador ha sido implementado en Java siguiendo la arquitectura clásica para este tipo de aplicaciones, incluyendo un Analizador Léxico, Sintáctico y Semántico, así como un Generador de Código. Se ha documentado su arquitectura y las decisiones de diseño que han conducido a la misma. Además, se ha demostrado su funcionamiento con un caso práctico de análisis de eventos en métricas de servidores. Por último, se ha documentado el lenguaje TESL, en cuyo desarrollo se ha colaborado.

Abstract

This Bachelor's Thesis describes the design and implementation of a compiler and a runtime library for the domain-specific language TESL, a high-level language for analyzing time series events developed by a research group from the Technical University of Madrid. This is the first fully implemented TESL compiler, and serves as basis for the continuation of the development of the language. The compiler has been implemented in Java following the classical architecture for this kind of systems, having a four phase compilation with a Lexer, a Parser, a Semantic Analyzer and a Code Generator. Its architecture and the design decisions that lead to it have been documented. Its use has been demonstrated in an use-case in the domain of server metrics. Finally, the TESL language itself has been extended and documented.

Índice

1 Introducción

El análisis de eventos en series temporales es un área de investigación actual con aplicaciones en diferentes campos como la medicina, la economía, la sismología o la meteorología [1]. Fruto de las investigaciones realizadas en este área por miembros de la Universidad Politécnica de Madrid surge el lenguaje para la especificación de eventos en series temporales denominado ESL (de sus siglas en inglés, Event Specification Language) y desarrollado por Lara et al. [1] en 2009. Se trata de un lenguaje de programación declarativo, de alto nivel, que permite a un experto en el dominio definir, de una manera sencilla, cómo son los eventos en el dominio, es decir, qué comportamientos de la serie corresponden a eventos. Como parte de la misma investigación se contemplaba el desarrollo del compilador del lenguaje. Este compilador ha de analizar la especificación de eventos realizada y, si no contiene errores léxicos, sintácticos ni semánticos, traducirla a un programa capaz de recorrer la serie y extraer todos los eventos que contenga. El primer prototipo de un compilador para el lenguaje ESL fue desarrollado en C# por Martínez [2] en 2011.

Pese al gran avance logrado, el lenguaje ESL adolecía de una importante limitación al no permitir establecer relaciones temporales entre los eventos. Ello llevó al mencionado grupo de investigación a desarrollar un nuevo lenguaje, mucho más potente que el anterior, al cual se llamó TESL (por sus siglas en inglés, Temporal Event Specification Language). Este lenguaje ha sido diseñado por Álvarez [3] como una extensión y mejora de los conceptos planteados en el lenguaje original ESL, e incluye un gran número de nuevas funcionalidades entre las que se encuentran: la posibilidad de definir eventos compuestos por otros eventos, la de definir conjuntos de eventos, la de establecer relaciones temporales (de anterioridad, posterioridad o simultaneidad) entre los elementos del lenguaje, la de contar el número de eventos que ocurren en un intervalo, etc.

Para obtener el compilador de TESL se pensó en ampliar el anterior compilador de ESL. De la adaptación de los módulos de análisis se encargó Ojeda [4] en 2014. Aunque este prototipo supuso un gran avance, la implementación arrastraba decisiones de diseño las cuales hacían muy difícil la extensión y mejora del código del traductor. Debido a ello, surge la necesidad del rediseño y

reimplementación de un nuevo compilador; un compilador con una arquitectura sólida que permita continuar con el desarrollo del lenguaje en el futuro y utilizando herramientas modernas y multiplataforma. Es en ese contexto en el que se enmarca el presente trabajo de Fin de Grado.

El presente trabajo trata sobre el diseño e implementación de un nuevo compilador para el lenguaje TESL. El compilador ha sido programado en Java y tiene como código objeto el mismo lenguaje. Sigue una arquitectura clásica con cuatro fases, en la cual el programa fuente de entrada es procesado progresivamente por los analizadores léxico, sintáctico y semántico y el generador de código, utilizándose como representación intermedia un Árbol Sintáctico Concreto. Los analizadores léxico y sintáctico han sido generados automáticamente con la herramienta ANTLR, cuya librería es también usada en otros componentes del compilador.

La memoria del trabajo está organizada como sigue. Un capítulo de Conocimientos Específicos, en el cual se realiza una revisión de los conocimientos relevantes en el área de los lenguajes de programación y compiladores. Un capítulo de Estado del Arte, en el que se proporciona una visión de conjunto sobre las herramientas disponibles para análisis de series temporales y sus implementaciones. El siguiente capítulo, El Compilador, es el capítulo principal del trabajo y en él se muestra el diseño e implementación del compilador realizado. A continuación, en el capítulo de Caso de Uso se presenta la compilación y ejecución de una especificación de eventos en TESL. Finalmente se presentan unas conclusiones y unas líneas futuras de trabajo. Además, la memoria incluye un anexo en el cual puede encontrase la especificación del lenguaje TESL.

2 Conocimientos específicos

2.1 Gramáticas libres de contexto

Las gramáticas libres de contexto son gramáticas formales compuestas por un conjunto de símbolos terminales, los tokens, un conjunto de símbolos no terminales, de los cuales uno es el axioma, y un conjunto de producciones. Las producciones están compuestas por un símbolo no terminal en la parte izquierda de la misma y una secuencia de símbolos terminales y no terminales (o la cadena vacía) en la parte derecha. Debido a que solo hay un único símbolo en la parte izquierda, las producciones pueden ir aplicándose sin tener en cuenta el contexto [5]. Un ejemplo de gramática libre de contexto para expresiones aritméticas es el siguiente:

Terminales = {int, float, +, -, *, /}

No terminales = {EXP, CONST, OP}

Axioma = EXP

Reglas de producción:

EXP -> CONST OP CONST

OP -> + | - | * | /

CONST -> int | float

Listado 2.1: Ejemplo de gramática de contexto libre

Para derivar las reglas, se selecciona un símbolo no terminal (inicialmente, el axioma) y se sustituye por la secuencia de símbolos asociada al mismo en una de las producciones. Después se selecciona otro de los símbolos no terminales de la secuencia resultante y se repite el proceso hasta que la secuencia esté compuesta únicamente por símbolos terminales.

El conjunto de todas las secuencias de símbolos que se pueden obtener mediante la derivación de reglas es el lenguaje que la gramática representa. Los lenguajes representados por gramáticas de contexto libre se denominan lenguajes de contexto libre [5].

Las gramáticas de contexto libre pueden ser usadas para describir la amplia mayoría de los lenguajes de programación [5]. Por ello, son usadas profusamente en especificaciones de lenguajes, manuales, y en programas como generadores automáticos de analizadores sintácticos, los cuales suelen utilizar alguna variación de la notación BNF.

2.1.1 Notación de Bakus-Naur (BNF)

La Notación de Bakus-Naur (BNF) es una notación para representar gramáticas de contexto libre creada por John Bakus y Peter Naur en la década de los 50 y usada en generadores de analizadores sintácticos clásicos como Yacc. La sintaxis de la misma, descrita en la propia notación, es la siguiente:

<production> ::= “<” <production-name> “>” “::=” <expression>

<expression> ::= <list> | <list> “|” <expression>

<list> ::= <term> | <term> <list>

<term> ::= <literal> | <production-name>

<token> ::= ‘“’ <text> ‘“’ | “‘” <text> “’”

Listado 2.2: Notación BNF expresada en BNF

2.1.2 Notación de Bakus-Naur Extendida (EBNF)

La Notación de Bakus-Naur Extendida (EBFN) es una extensión de la notación BFN desarrollada originalmente por Niklaus Wirth. Es más expresiva y permite describir las gramáticas de manera más sencilla. La sintaxis de la misma, descrita en la propia notación, es la siguiente:

```
production = production_name “=” expression “;” ;

expression = alternative { “|” alternative } ;

alternative = term { term } ;

term = production_name | token [ "..." token ] | group | option | repetition ;
group = "(" expression ")" ;
option = "[" expression "]" ;
repetition = "{" expression "}" ;
```

Listado 2.3: Notación EBNF expresada en EBNF

La especificación del lenguaje TESL incluida en el anexo de esta memoria está escrita con esta notación.

2.2 Compiladores

Tal y como se define en una de las publicaciones más relevantes del campo, “Compilers. Principles, Techniques & Tools” [6], más comúnmente conocida como “el Libro del Dragón”, un compilador es un programa que lee un programa escrito en un lenguaje, denominado lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, denominado lenguaje objeto.

El término compilador está normalmente reservado para programas que traducen un lenguaje fuente de alto nivel a un lenguaje objeto de bajo nivel. Aunque esta es la tarea más habitual para un programa de este tipo, existen compiladores que realizan otro tipo de traducciones. Algunos ejemplos de ello son: el compilador de CoffeeScript, que traduce este lenguaje a JavaScript; HiveQL, un lenguaje similar a SQL el cual es traducido a un grafo dirigido acíclico de trabajos de Map-Reduce; o Pandoc, un compilador que realiza conversiones entre lenguaje de marcado como HTML, Latex o Markdown.

2.2.1 Historia

Se piensa que el primer compilador, así como el término en sí mismo, fue desarrollado por Grace Murray Hopper en 1952 para el lenguaje A-0 [7]. Este programa, denominado A-0 system, era capaz de traducir código matemático simbólico en código máquina. Hopper recibió cierta oposición a su programa debido a que el pensamiento imperante de la época era que los ordenadores no eran capaces de escribir sus propios programas [7]. En aquella época los programas eran escritos directamente en código ensamblador o en código máquina.

Años más tarde, en 1957, un equipo liderado por John Backus en IBM desarrolló el lenguaje FORTRAN y un compilador para el mismo. Este compilador se considera el primer compilador completo, siendo capaz de traducir programas escritos en FORTRAN a código máquina para el *mainframe* de IBM 704 [8].

2.2.2 Arquitectura clásica

La arquitectura clásica de un compilador está compuesta por cuatro fases las cuales transforman gradualmente un fichero fuente de entrada en un fichero objeto de salida. Estas fases son realizadas por cuatro módulos independientes denominados Analizador Léxico, Analizador Sintáctico, Analizador Semántico y Generador de Código. Los módulos se comunican entre sí mediante algún tipo de representación intermedia del programa fuente [6].

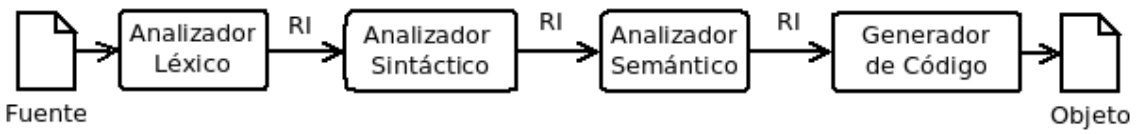

Figura 2.1: Fases clásicas de un compilador

Representación intermedia

La representación intermedia debe ser algún tipo de estructura de datos que permita representar la información contenida en el lenguaje fuente y sea fácil de recorrer. Esta estructura de datos debe representar no solo los elementos contenidos en el lenguaje fuente, si no también las relaciones de orden y anidamiento entre ellos. Debido a ello, un árbol es la estructura de datos perfecta para la tarea [9]. Existen dos tipos de árboles usados mayoritariamente en este tipo de aplicaciones, denominados Árbol Sintáctico Concreto y Árbol Sintáctico Abstracto.

Un Árbol Sintáctico Concreto es un árbol el cual representa con sus nodos el lenguaje tal y como ha sido reconocido por el Analizador Sintáctico. Sus nodos interiores son reglas de la gramática y sus hojas son tokens. Este tipo de árboles están directamente asociados a la sintaxis del lenguaje [9].

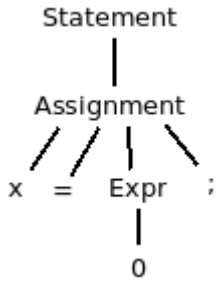

Figura 2.2: Árbol Sintáctico concreto

Un Árbol Sintáctico Abstracto es un árbol el cual representa el lenguaje fuente sin incluir ciertas construcciones sintácticas no esenciales como puntos, puntos y coma o paréntesis. Por ello, recorrer este tipo de árboles es más eficiente que recorrer los anteriormente descritos, ya que su número de nodos es mucho menor. Idealmente, este tipo de árboles no deben verse afectados por cambios sintácticos en el lenguaje [9].

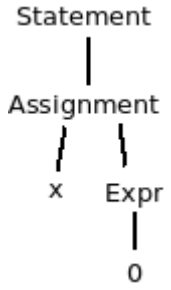

Figura 2.3: Árbol Sintáctico Abstracto

La clave del porqué del uso de Árboles Sintácticos Concretos es que pueden ser generados muy fácilmente. Por ello existen herramientas automáticas capaces de generar Analizadores Sintácticos que los construyen a partir de una gramática del lenguaje.

Análisis Léxico

La fase de Análisis Léxico es la primera fase del compilador y el módulo principal encargado de la misma es el Analizador Léxico, también denominado Lexer. Su objetivo principal es transformar el programa fuente en una secuencia de tokens (los componentes léxicos mínimos) a partir de la lectura de dicho programa fuente carácter a carácter. También se encarga de la detección de errores léxicos. Este tipo de errores se detectan cuando el Analizador Léxico encuentra una secuencia de caracteres que no encaja con ninguno de los tokens del lenguaje [6].

Cada uno de los tokens tiene un código que lo identifica y un atributo que proporciona información concreta sobre el token siempre que sea necesario. Así por ejemplo, la secuencia de caracteres "1 + 5.3;" podría representarse por la secuencia de tokens <INT, '1'>, <OP_ARIT, '+'>, <FLOAT, '5.3'>, <SEMICOLON, -> [6] [9].

Es habitual que los analizadores léxicos ignoren los espacios en blanco y saltos de línea, puesto que no suelen tener valor sintáctico ni semántico. Esto no siempre es así, ya que existen ciertos lenguajes para los cuales estos espacios son esenciales y el analizador léxico debe generar token para ellos [6][10].

Análisis Sintáctico

La fase de Análisis Sintáctico tiene como objetivo comprobar que la sintaxis de la secuencia de tokens generada por el analizador léxico es correcta, es decir, que el orden de los tokens que la conforman encaja con las reglas del lenguaje, así como transformar dicha secuencia en algún tipo de estructura de datos que represente el programa fuente. El módulo encargado de ello es el Analizador Sintáctico, también denominado Parser y típicamente el resultado de este análisis es un Árbol Sintáctico Concreto [6].

Existen múltiples algoritmos para realizar esta tarea. Estos pueden clasificarse generalmente en descendentes (top-down) y ascendentes (bottom-up). Tal y como su nombre indica, la clasificación se basa en el orden en el que se realiza la construcción del árbol, bien sea empezando por el nodo raíz o por sus hojas. Algunos ejemplos de ello son el método LL, un algoritmo descendente el cual trabaja expandiendo siempre el símbolo no terminal más a la izquierda en cada instante, o el método LR, el cual construye un árbol igual que el que se obtendría expandiendo siempre el símbolo no terminal más a la derecha [6].

El tipo de errores que se pueden detectar en esa fase se denominan errores sintácticos. Ejemplos de este tipo de errores podría ser el olvido de terminar una sentencia con punto y coma o el no cerrar un paréntesis en una expresión.

Los analizadores sintácticos pueden programarse manualmente siguiendo algún algoritmo como el LL(1) o el SLR(1) o bien ser generados automáticamente por algún tipo de herramienta a partir de una gramática que represente los tokens del lenguaje [6][9]. Es habitual el uso de herramientas automáticas para la generación del analizador, denominadas Generadores de Analizadores Sintácticos. El ejemplo más clásico de estas herramientas es el generador Yacc, el cual es capaz de generar un analizador LARL(1) en C a partir de una gramática expresada en una notación similar a BNF [6][11].

Análisis Semántico

El Análisis Semántico tiene como objetivo comprobar que el programa fuente respeta las reglas semánticas del lenguaje [6]. Pongamos por ejemplo la

expresión “n / 2”. Una regla semántica para este tipo de expresiones podría ser que “n” debe ser de tipo numérico. Esta regla puede comprobarse alternativamente en dos momentos, durante la compilación del programa, en cuyo caso se dice que el lenguaje tiene Tipado Estático, o durante la ejecución del mismo, denominado Tipado Dinámico. La fase de Análisis Semántico, por lo tanto, será parte del compilador únicamente cuando nos encontremos ante la implementación de un lenguaje con tipado estático [9].

El módulo encargado de esta fase se denomina Analizador Semántico. Para llevarla a cabo, el analizador semántico hace uso de la estructura intermedia generada en la fase anterior, realizando sobre la misma las tareas de cálculo y promoción de tipos [9]. El cálculo de tipos consiste en calcular los tipos de los diferentes elementos usados a lo largo del programa y cómo estos interactúan entre sí con las construcciones del lenguaje, comprobando que se respetan las diferentes reglas semánticas del mismo. Estos tipos calculados se van anotando sobre el árbol.

En ciertos lenguajes, se da el caso de que tipos diferentes pueden ser transformados para operar entre sí. La tarea de realizar estas transformaciones se denomina Promoción de tipos [9]. Por ejemplo, en el caso de la expresión “1+1.5”, el compilador debe o bien promocionar el tipo entero del primer operando al tipo real del segundo operando o detectar un error en caso de que el lenguaje no permita las operaciones entre tipos enteros e reales.

Los errores detectados en esta fase se denominan errores semánticos y son los que se dan, por ejemplo, cuando se realizan operaciones entre operadores y operandos que no son compatibles.

Generación de código

Una vez se ha comprobado que el programa respeta todas las reglas del lenguaje, se pasa a la fase de generación de código. El objetivo de esta fase es transformar la representación intermedia del programa, generada y anotada en las fases anteriores, a un programa equivalente en el lenguaje objeto de salida.

Tradicionalmente, el lenguaje objeto es un lenguaje de más bajo nivel que el lenguaje fuente, normalmente código máquina o código ensamblador. En tal caso, la generación de código se convierte en una tarea muy compleja, ya que el compilador debe generar instrucciones específicas para una arquitectura concreta, así como manejar la memoria de la máquina y los registros de la misma. Por ello, los generadores de código suelen ser reutilizados para la implementación de múltiples lenguajes. Ese es el caso por ejemplo de GCC, cuyo generador y optimizador de código es usado por compiladores de C, C++, Fortran, Objective-C o Go [12]. Otro ejemplo es el generador de código y optimizador de LLVM, usado por compiladores de C, C++, Ada, Objective-C, Rust o Haskell [13].

Ciertos compiladores generan código al mismo nivel que su lenguaje fuente de entrada, en cuyo caso suelen ser llamados traductores [9]. Existen múltiples motivos para ello, desde no tener otra alternativa, como es el caso de los compiladores que generan JavaScript para poder funcionar en los navegadores, hasta facilitar el rápido desarrollo del lenguaje generando C en vez de código ensamblador. Algunos ejemplos son los compiladores a JavaScript de CoffeeScript o Dart, el compilador a C del lenguaje Nimrod o CFront, el primer compilador de C++ el cual generaba C.

2.3 Generadores automáticos de analizadores sintácticos

Un generador de analizadores sintácticos es una herramienta de generación automática de código la cual es capaz de generar analizadores sintácticos para un lenguaje a partir de una gramática del mismo. Mientras que las primeras herramientas con intenciones similares datan de los años 60, probablemente la más conocida sea Yacc, desarrollada en 1970 por Stephen C. Johnson en AT&T Labs [11].

Yacc es un generador automático de analizadores sintácticos de código libre el cual genera analizadores LALR(1) a partir de gramáticas en una sintaxis similar a BNF [11]. El diseño de esta herramienta es muy popular y ha sido reimplementada en multitud de lenguajes a lo largo de su historia. Entre ellas destaca Bison, desarrollada por la fundación GNU y usada en la implementación estándar de multitud de lenguajes de primer nivel, como Ruby, PHP o Go [14].

La mayoría de este tipo de herramientas generan analizadores sintácticos que implementan alguna variante del algoritmo LR, especialmente LARL(1). A pesar de ello, existen herramientas que generan otro tipo de analizadores sintácticos. Entre ellas destaca ANTLR, la cual genera analizadores LL(*) [15].

2.4 La herramienta ANTLR

ANTLR es un generador automático de analizadores sintácticos LL(*) de código libre desarrollado por Terrance Paar de la universidad University of San Francisco. A partir de la gramática formal expresada en una notación similar a BNF, ANTLR genera un Analizador para el lenguaje capaz de construir automáticamente Árboles de Sintaxis Concretos, así como el código para recorrerlos [15].

Esta herramienta es usada en proyectos de primer nivel, como los compiladores de los lenguajes HiveQL y Pig, el ORM Hibernate o Twitter search, donde ANTLR es usado para analizar más de dos millones de consultas diarias [15].

La última versión de la herramienta, ANTLR4, está programada en Java y es capaz de generar analizadores en Java, C# y C++.

2.4.1 Gramáticas en ANTLR

Las gramáticas en ANTLR están representadas mediante una notación similar a BNF o EBNF, pero con una serie de diferencias: los símbolos no terminales

empiezan por letra minúscula y los terminales por letra mayúscula; los tokens con un único carácter pueden representarse directamente como ese carácter; se han añadido los operadores '?', '+' y '*' para indicar secuencias de tokens de cero-o-uno, uno-o-n y cero-o-n, respectivamente. Además de las reglas, las gramáticas incluyen un bloque inicial de declaraciones, donde es posible, por ejemplo, importar otras gramáticas o incluir código que se añadirá al analizador sintáctico generado[15].

En cuanto a la precedencia de operadores, se declara implícitamente con el orden de las reglas, teniendo mayor precedencia los operadores en reglas situadas antes en la gramática e igual precedencia los operadores situados en la misma regla. Un ejemplo de una gramática en formato ANTLR, con precedencia de la multiplicación y la división sobre la suma y la resta es el siguiente:

```
grammar: Java
...
while: WHILE '(' exp ')' '{' (statement ';')* '}'
exp: exp ('*'|'/') exp | exp ('+'|'-') exp
...
```

Listado 2.4: Gramática en formato ANTLR

2.4.2 Bison vs ANTLR

La principal diferencia entre Bison y ANTLR es el tipo de analizador sintáctico que generan. Mientras que Bison y otros generadores herederos del clásico Yacc generan analizadores LARL(1), ANTLR genera analizadores LL(*). Estos dos tipos de algoritmos son radicalmente opuestos: LARL(1) es una variante del algoritmo ascendente LR y construye un árbol que puede ser visto como el obtenido por derivaciones a la derecha (aunque construido justo en el orden inverso), mientras que LL(*) lo hace descendentemente aplicando la derivación más a la izquierda. Además, los algoritmos LR aceptan gramáticas que son

incompatibles con algoritmos LL; concretamente, los analizadores sintácticos LL no aceptan gramáticas con reglas recursivas por la izquierda. [11][14][15].

2.4.3 Eliminación de la recursividad por la izquierda

Tal y como se indicó anteriormente, los analizadores sintácticos LL(*) generados por ANTLR no son capaces de aceptar gramáticas recursivas por la izquierda. Un ejemplo de este tipo de gramáticas es el siguiente:

```
expr : expr + expr | INT
```

Listado 2.5: Gramática para expresión recursiva por la izquierda

La recursividad está presente en un gran número de gramáticas ya que permite expresar el lenguaje a generar de una forma muy intuitiva. Por ello, la última versión de la herramienta transforma las reglas recursivas por la izquierda, generando otras equivalentes, las cuales pueden ser aceptadas por analizadores sintácticos LL. Esto se realiza de manera transparente como un paso más durante la generación del analizador sintáctico [15].

2.4.4 Entorno de Ejecución

Además de la herramienta para generar Analizadores Léxicos y Sintácticos, ANTLR incluye una librería de entorno de ejecución. Esta librería debe añadirse al código generado y en ella se encuentran las clases y métodos usados en este código, como las clases Parser, Lexer y Token, además de otras utilidades relacionadas con la construcción de compiladores. Entre ellas destacan las usadas para el manejo de los árboles construidos por el analizador sintáctico (*tree listeners* y *walkers*) [15].

2.5 Series temporales

Una serie temporal es una secuencia de datos medidos sucesivamente a lo largo del tiempo, normalmente a intervalos regulares [16]. Las series temporales pueden ser unidimensionales si para cada instante de tiempo se registra un único valor, o multidimensionales si se registran varios.

Las series temporales son usadas para comprender la estructura de los datos medidos y poder construir modelos que se utilizan, normalmente, con el fin de generar predicciones. Por ello son usadas en múltiples dominios como estadística, economía, sismología, meteorología, etc. para tareas como procesamiento de señales, reconocimiento de patrones, predicciones económicas, de terremotos, del clima o análisis del mercado de valores [1][16].

Existen múltiples técnicas para el análisis de series temporales, que van desde las técnicas estadísticas tradicionales hasta las técnicas de aprendizaje automático para clasificación y *clustering* que se utilizan en la minería de datos.

En ciertos dominios, el objetivo del análisis es la identificación de eventos que aparecen en las series estudiadas. Un evento es una subsecuencia de la serie temporal la cual se asemeja a un determinado patrón o tiene unas ciertas características. El análisis de series temporales basado en eventos es un área actual de investigación dentro del campo del análisis de series temporales [1].

2.6 Programación declarativa

La programación declarativa es un paradigma de programación en el cual, o bien los programas son descritos, o las condiciones de la solución al programa son descritas, sin especificar cómo se debe obtener esa solución, es decir, sin especificar su control de flujo [17]. El programa expresa qué debe hacerse en vez de la secuencia de pasos para hacerlo, como ocurre en paradigma de programación imperativo.

Los lenguajes declarativos, aunque menos populares que los imperativos o funcionales para uso general, son ampliamente usados en algunos dominios específicos. Probablemente los dos ejemplos más populares sean el lenguaje de consulta a bases de datos SQL y el lenguaje de marcado para declaración de interfaces HTML. Otros lenguajes populares son las expresiones regulares, el lenguaje de programación lógica Prolog o el lenguaje de especificación de Make.

2.7 El lenguaje TESL

El lenguaje TESL (Temporal Event Specification Language) es un lenguaje de programación declarativo de dominio específico para la especificación de eventos en series temporales. Es un sucesor directo del lenguaje ESL (Event Specification Language), el cual se ha ampliado y modificado para facilitar el análisis de eventos temporales complejos.

TESL surge del esfuerzo dedicado por múltiples miembros de nuestra universidad. Su predecesor, el lenguaje ESL, fue diseñado por Lara et al. [1] en 2009 , mientras que el diseño de TESL fue realizado por Álvarez [3] como tesis de fin de master en 2014. Sobre el mismo también trabajaron Martínez [2], el cual realizó un primer prototipo de traductor para el lenguaje ESL y Ojeda [4], que lo extendió incorporando las características de TESL en los módulos de análisis.

El lenguaje TESL es un lenguaje puramente declarativo. Por ello carece de estructuras de control de flujo. Un programa escrito en TESL está compuesto por declaraciones de series temporales, conjuntos de puntos, eventos y conjuntos de eventos. Estas declaraciones describen mediante expresiones las propiedades que deben cumplir los elementos que las forman. El resultado de aplicar un programa TESL a un conjunto de series temporales es el conjunto de eventos que se encuentran en las series.

Una especificación completa del lenguaje puede encontrarse en el anexo de esta memoria.

3 Estado del arte

3.1 Bases de datos para series temporales

El almacenamiento y análisis de series temporales es un problema al que deben enfrentarse muchas de las organizaciones que operan en internet. Una rama muy importante del análisis de series temporales es el análisis de los eventos que se presentan en las mismas. Por ejemplo, durante la operación de los servidores que soportan las aplicaciones web que usamos cada día se recolectan todo tipo de datos (series temporales que contienen eventos y métricas diversas) que deben ser procesados y almacenados. Además, algunos de los sistemas que conforman estas aplicaciones, como feeds y streams de novedades, son en esencia secuencias de eventos en series temporales. Por ello, han ido apareciendo en los últimos años diferentes aproximaciones para abordar este problema de manera escalable.

Para enfrentarse a este problema de la gestión y análisis de eventos y métricas a escala, estos sistemas suelen basarse en bases de datos u otros almacenes de datos distribuidos. De esta manera intentan proporcionar una gestión eficiente y de los mismos y reutilizar el esfuerzo invertido en la construcción de estas herramientas.

Estos sistemas suelen proporcionar algún tipo de interfaz o lenguaje para poder interactuar con los eventos y datos almacenados, de una manera similar a como SQL permite el acceso a los datos en bases de datos relacionales tradicionales. Entre las acciones que habitualmente proporcionan está la agregación de eventos en clases así como el cálculo de métricas de los mismos, muchas veces de una manera similar a las construcciones que proporciona el lenguaje TESL.

Por esta razón, se van a explorar una serie de sistemas basados en bases de datos los cuales o bien están específicamente diseñados para la gestión de eventos en series temporales, o bien son usados para esta tarea debido a que sus propiedades los hacen especialmente indicados para ello. El estudio de los mismos tiene como objetivo la recolección de información relevante para el diseño e implementación del lenguaje TESL.

3.1.1 InfluxDB

InfluxDB [18] es una base de datos distribuida específicamente diseñada para series temporales, eventos y métricas. Es de código libre y está programada en Go. Su desarrollo comenzó en el 2013, por lo que es un producto nuevo y surge a partir de una *startup* de la aceleradora YCombinator.

Los datos almacenados en InfluxDB están organizados en bases de datos, series temporales y eventos, los cuales equivaldrían respectivamente en una base de datos relacional tradicional a bases de datos, tablas y filas. Los datos en InfluxDB no tienen esquema, esto quiere decir, hablando en terminología tradicional, que las filas de una tabla pueden tener columnas diferentes. En estas columnas pueden almacenarse valores de tipo string, boolean, integer o float.

InfluxDB proporciona un lenguaje parecido a SQL para realizar consultas a la base de datos. Este lenguaje implementa filtros mediante *where* y agregados mediante *group_by*, *merge* y *join*. Para el análisis de los datos proporciona funciones matemáticas como *min*, *max*, *mean*, *mode*, *median* y *percentile*. Además, implementa un tipo de consultas denominadas *continuous queries,* las cuales son realizadas constantemente por la base de datos y almacenadas en la misma.

El lenguaje es primeramente analizado mediante unos analizadores Léxico y Sintáctico generados automáticamente mediante las herramientas Flex y Bison. El analizador sintáctico es usado para construir un Árbol Sintáctico Abstracto que representa las consultas de entrada, el cual es pasado al Motor de Consultas para su interpretación. El análisis semántico es realizado por el motor al mismo tiempo que el procesamiento de las consultas.

3.1.2 Roshi

Roshi [19] es un sistema de almacenamiento distribuido de alto rendimiento para eventos en series temporales. Ha sido desarrollado por SoundCloud y es usado para el almacenamiento del *stream* de su aplicación, es decir, la lista ordenada en el tiempo de novedades que un usuario recibe sobre los usuarios que está siguiendo. Es de código libre y fue presentado al público en Mayo de 2014.

Roshi esta implementado en Go como un sistema distribuido sobre Redis, haciendo uso del conjunto ordenado que Redis proporciona. Estos conjuntos están internamente implementados usando la estructura de datos *Skip List* y un diccionario, proporcionando O(nlogn) para inserciones y O(logn+m) para la obtención de los m últimos elementos.

En esencia Roshi implementa un índice de alto rendimiento para eventos ordenados por tiempo. Por ello, únicamente proporciona un interfaz para añadir datos mediante el comando *insert*, borrar datos añadidos mediante el comando *delete* y obtener los m últimos eventos mediante el comando *select*.

Los eventos almacenados en Roshi son representados por una clave, un instante de tiempo (*timestamp*) y un valor. La clave identifica a un evento concreto, el instante de tiempo sirve para ordénalos y el valor puede ser cualquier información asociada al evento y codificada en base 64.

3.1.3 Cube

Cube [20] es un sistema de código libre para el almacenamiento y análisis de eventos en series temporales. Ha sido desarrollado por Square, y es usado por ellos internamente en producción. Según afirman, el sistema puede soportar cargas de varias miles de peticiones por segundo.

Está implementado en JavaScript usando Node.js y MongoDB. Su arquitectura está compuesta por dos servicios web, un *collector* el cual recibe y almacena los eventos y un *evaluator*, el cual procesa consultas a los datos almacenados; ambos, los cuales exponen su funcionalidad mediante una API HTTP.

En el modelo de datos de Cube, los eventos son simples objetos JSON con dos campos *type* y *timestamp* y un campo arbitrario *data*. Estos eventos se envían al servicio *collector*, el cual los preprocesa y almacena en la base de datos MongoDB. Además, Cube utiliza el concepto de métricas como reducciones de un grupo de eventos de interés a un único valor numérico. Estas métricas son calculadas y guardadas en caché por el *evaluator*.

Cube proporciona un sencillo lenguaje para consultar los eventos que almacena, así como para calcular métricas sobre los mismos. Este lenguaje incluye funciones *reduce* como *sum*, *min*, *max*, *median* o *distinct*, así como funciones *filter* de comparación *eq*, *ne*, *lt*, *le*, *ge*, *gt*. Para realizar consultas, se construye una expresión y se envía al *evaluator* mediante una petición HTTP, indicando, además, la ventana de tiempos que se quiere procesar. El *evaluator* transforma estas expresiones a consultas para la base de datos MongoDB y devuelve los resultados de las mismas.

3.2 Librerías científicas

El análisis de series temporales es una tarea común en ciertos ámbitos como, por ejemplo, las Finanzas, la Neurociencia o la Economía. Por ello, es común encontrar librerías para esta tarea en los lenguajes de programación popularmente usados en esos ámbitos. Dos de los lenguajes que más suelen usarse en esos círculos son Python y R. A continuación se van a explorar dos de las opciones disponibles para estos lenguajes:, la librería para Python Pandas y la librería estándar de R.

A continuación se van a explorar dos de las opciones disponibles los lenguajes anteriormente mencionados: la librería para Python Pandas y la librería estándar de R. El estudio estas librerías tiene como objetivo la obtención de información relevante para el diseño e implementación de la librería de entorno de ejecución del TESL.

3.2.1 Pandas

Pandas [21] es una librería de código libre para análisis de datos en Python. Su desarrollo comenzó en 2008 con el objetivo de proporcionar una herramienta flexible y eficiente para el análisis cuantitativo de datos financieros. Proporciona estructuras de datos eficientes para el manejo de tablas numéricas y series temporales, así como funciones para trabajar sobre las mismas. Su última versión data de Febrero del 2014 y es popular en la comunidad de Python, tanto en la industria como en la Academia.

La librería trabaja con datos en memoria, proporcionando tres estructuras de datos denominadas *Serie*, *DataFrame* y *Panel*. Una *Serie* es una estructura

unidimensional compuesta por un array de valores, un array de índices para los mismos y un nombre. Un *DataFrame* es una estructura de datos bidimensional, similar a lo que sería una tabla en SQL y puede estar compuesta por diferentes *Series*. Por último, un *Panel* es una estructura de datos tridimensional.

Para la entrada y salida de datos, Pandas proporciona una serie de funciones las cuales pueden por ejemplo cargar archivos CSV, JSON u obtener datos de bases de datos SQL. De igual manera, existen funciones equivalentes para guardar los datos en esos formatos.

Pandas proporciona múltiples funciones de agregación para operar con las estructuras anteriormente descritas. Además de las usuales *count*, *mean*, *median*, *min*, *max*, *mode, std y var* proporciona también la desviación absoluta de la media *mad*, cuantiles *quantil*, y cálculo de histogramas *value_ count*. Por último, es posible aplicar tus propias funciones de agregación mediante *lambdas* y la función *apply*.

Para manejar los datos eficientemente, Pandas proporciona una serie de funciones *group-by*, *joins* y *merges*, con una semántica similar a las que se puede encontrar en SQL. Incluye además una funcion *where* para filtrar los datos en base a expresiones booleanas y su inversa, *mask*.

Pandas es usado habitualmente para analizar series temporales en el área de las finanzas; por ello dispone de una serie de funcionalidades específicas para ello. En Pandas, una serie temporal es modelizada como una instancia de la estructura de datos *Serie* indexada mediante instantes temporales. Estos instantes temporales pueden ser o bien *Timestamps* o periodos de tiempo denominados *Periods*.

Para poder trabajar con estas series temporales, Pandas proporciona funciones específicas que permiten, por ejemplo, muestrearlas para pasar de datos en segundos a datos en minutos, cambiar sus frecuencias o crearlas dinámicamente mediante rangos de tiempo y funciones numéricas.

3.2.2 Librería estándar de R

R [22] es un lenguaje de programación especialmente diseñado para el análisis estadístico. Junto con el lenguaje se proporciona un entorno de programación el cual incluye muchas librerías con utilidades en este dominio. Por ello, la librería estándar de R proporciona funcionalidades especialmente dedicadas para el manejo y análisis de series temporales.

La principal función para esta tarea es *ts*, la cual permite construir series temporales a partir de un conjunto de datos. Las series están modelizadas como vectores o matrices que contienen valores muestreados a instantes regulares de tiempo. Los datos pueden ser cargados desde archivos en varios formatos como CSV o DAT.

Aunque la clase *ts* solo modeliza series temporales a instantes regulares, sirve como base a otros paquetes de la librería como *zoo* o *xts* los cuales implementan infraestructura para series temporales irregulares.

El entorno de R proporciona múltiples funciones que implementan diferentes modelos de predicciones. Algunos ejemplos de ello son alisado exponencial mediante el modelo Holt-Winters, modelos autorregresivos AR o modelos ARIMA, SARIMA y ARIMAX

Uno de los puntos fuertes del entorno de programación de R son sus excelentes librerías de gráficos, las cuales pueden utilizarse para mostrar las series temporales y los análisis realizados sobre las mismas.

4 El compilador

4.1 Arquitectura

La solución que se propone en este Trabajo Fin de Grado es un compilador para el Lenguaje de Especificación de Eventos en Series Temporales TESL, que analizará un fichero de entrada que contiene una especificación en TESL y, si es correcto, generará un programa escrito en Java capaz de recorrer un conjunto de series temporales y extraer los eventos que contienen.

La arquitectura del compilador sigue la clásica para este tipo de aplicaciones, estando compuesta por cuatro fases diferenciadas: análisis léxico, análisis sintáctico, análisis semántico y generación de código. Cada una de estas fases ha sido implementada como un módulo independiente.

Los dos primeros módulos, el Analizador Léxico y el Analizador Sintáctico, han sido generados automáticamente a partir de la gramática del lenguaje mediante la herramienta ANTLR. Cuando se inicia la compilación de un programa fuente de entrada, este es procesado por el analizador léxico, generando una secuencia de componentes léxicos mínimos (tokens). Esta secuencia es procesada por el analizador sintáctico, construyendo un Árbol Sintáctico Concreto.

Una vez se ha construido el árbol que representa al programa fuente, se procede a realizar el análisis de tipos sobre el mismo. Esta es la tarea del Analizador Semántico, el cual incorpora las reglas necesarias para realizar las comprobaciones de los tipos de las distintas construcciones del lenguaje TESL. Este componente ha sido implementado siguiendo el patrón Visitante e implementa un método para cada tipo de nodo del Árbol Sintáctico Concreto. Es en esos métodos donde se comprueban las reglas semánticas correspondientes. Además de la tarea del cálculo de tipos, este módulo es responsable de controlar los ámbitos (scopes) del programa fuente para asegurar el correcto tratamiento de los símbolos (identificadores) que se definen en dicho programa.

La última fase de la compilación es realizada por el Generador de Código. Este módulo está implementado igualmente siguiendo el patrón visitante y tiene como objetivo la generación de una clase en Java la cual implementa la especificación TESL de entrada. Para ello, el Generador de Código hace uso de la información recolectada en la fase anterior por el Analizador Semántico, recorriendo los nodos del árbol en post-orden y generando el código Java correspondiente a cada uno en forma de Strings. Estos Strings son concatenados recursivamente hasta que se construye la clase final en Java.

El código generado hace uso de las clases y métodos definidos en la Librería de Entorno de Ejecución. Es en ella donde se implementan las series temporales, conjuntos y eventos así como las funciones matemáticas y estadísticas, los rangos y los operadores no primitivos de Java.

Para poder controlar el compilador se ha implementado una interfaz por línea de comandos. Esta interfaz permite compilar programas TESL, generando código fuente con las clases correspondientes. Además, debido a que el compilador está también programado en Java, puede compilar, cargar y ejecutar los programas generados dinámicamente.

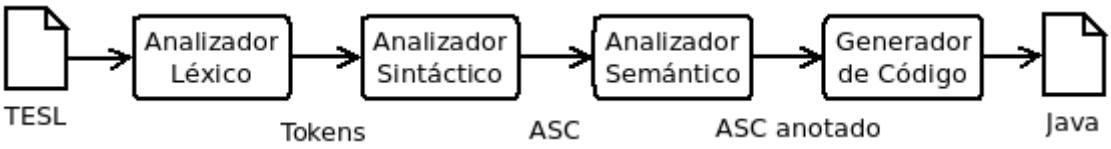

Figura 4.1: Fases del compilador TESL

4.1.1 Módulos

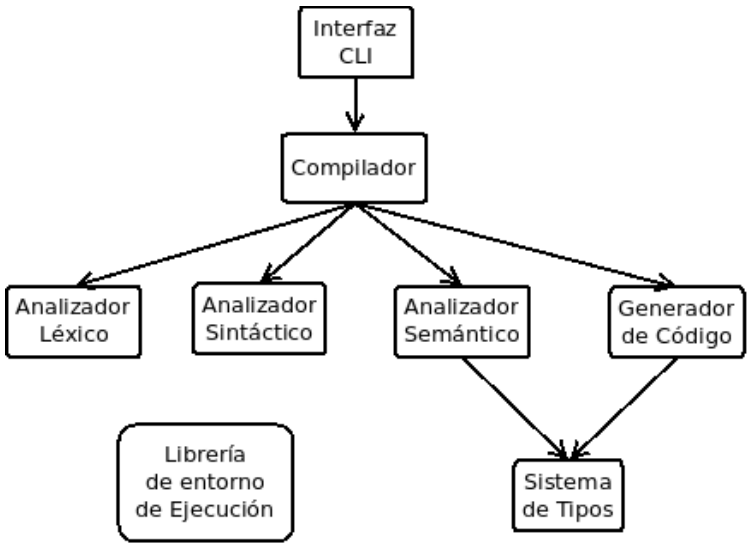

Figura 4.2: Diagrama de la arquitectura del compilador

Interfaz por línea de comandos
Permite controlar el compilador desde la línea de comandos mediante los comandos *compile*, *run*, *help* y *version*.

Compilador
Contiene la clase principal del compilador, la cual orquesta las diferentes fases de la compilación.

Sistema de tipos
Contiene las clases que representan los diferentes tipos del lenguaje y las reglas que indican cómo estos tipos se pueden combinar entre sí.

Analizador Léxico
Genera una secuencia de tokens a partir del código fuente de entrada, detectando los posibles errores léxicos. Es generado automáticamente a partir de la gramática.

Analizador Sintáctico
Genera un Árbol Sintáctico Concreto a partir de una secuencia de tokens, detectando los posibles errores sintácticos. Es generado automáticamente a partir de la gramática.

Analizador Semántico
Analiza el Árbol Sintáctico Concreto calculando los tipos de las distintas construcciones del lenguaje y construyendo los ámbitos. Detecta los posibles errores semánticos.

Generador de Código
Genera código en Java equivalente al programa fuente a partir del Árbol Sintáctico Concreto.

Librería de entorno de ejecución
Contiene las clases en java que implementan las diferentes funcionalidades del lenguaje.

4.2 Interfaz por línea de comandos

La interfaz por línea de comandos permite usar todas las funcionalidades del compilador. Para ello, la interfaz ofrece los comandos *compile*, *run*, *versión* y *help*, los cuales se detallan en la siguiente sección.

4.2.1 Comandos

Compile
Uso: tesl compile [<flags>] <program.tesl>

Compila un programa TESL generando una clase en Java. Por defecto se genera, en el directorio de trabajo, un fichero .java con el nombre especificado para él en la sentencia *eventspec* del programa TESL. El nombre es formateado para seguir las convenciones del lenguaje Java.

Opcionalmente pueden indicarse las los siguientes parámetros:

-o <output> especifica la carpeta en la cual se guardará el código generado
-p <package> especifica el *package* que se incluirá en el código Java generado

Run
Uso: tesl run <program.tesl> <data.csv>

Compila y ejecuta un programa TESL utilizando como datos el archivo .csv que se indique. Este fichero de datos debe incluir una línea por cada serie temporal especificada en el programa TESL. Para ello primero se compila el programa TESL en un archivo .java temporal. Este archivo es entonces compilado con el compilador de Java programáticamente y cargado dinámicamente mediante los mecanismos de reflexión de Java.

Opcionalmente puede indicarse el siguiente parámetro:

-l muestra una lista con los eventos encontrados

Help
Uso: tesl help [<command>]

Muestra un menú de ayuda general. Opcionalmente puede indicarse el nombre de un comando, en cuyo caso se mostrará una ayuda detallada del mismo, similar a las descripciones anteriores.

Version
Uso: tesl version

Muestra la versión del compilador.

4.2.2 Implementación

A continuación se describen las diferentes clases que conforman la implementación de la Interfaz por Línea de comandos.

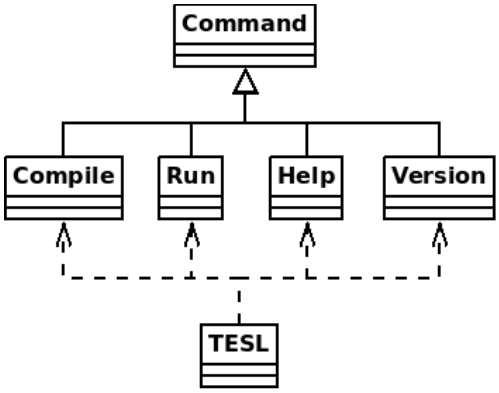

Figura 4.3: Diagrama de clases del módulo Interfaz por Línea de Comandos

TESL (es.upm.tesl.cli.TESL.java)
Clase que inicializa los comandos y orquesta su ejecución. Contiene el método *main* de la aplicación.

Command (es.upm.tesl.cli.Command.java)
Clase abstracta para los diferentes comandos. Cada uno de ellos está compuesto por un nombre *name*, una breve descripción *desc* y un manual de uso *usage*. Ofrece la siguiente interfaz:

- **public void run (String[] args):**
  Ejecuta el comando con los argumentos *args*

Compile (es.upm.tesl.cli.Compile.java)
Clase que implementa el comando Compile

Run (es.upm.tesl.cli.Run.java)
Clase que implementa el comando Run

Help (es.upm.tesl.cli.Help.java)
Clase que implementa el comando Help

Version (es.upm.tesl.cli.Version.java)
Clase que implementa el comando Version

4.3 Compilador

El compilador es el módulo encargado de orquestar la compilación, instanciando y controlando los demás módulos necesarios para cada una de las fases. Sirve como punto de acceso único al compilador TESL, permitiendo el uso del mismo programáticamente como una librería.

4.3.1 Implementación

A continuación se describe la única clase que conforma la implementación del módulo Compilador.

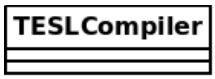

Figura 4.4 Diagrama de clases del módulo Compilador

TESLCompiler (es.upm.tesl.core.TESLCompiler)
Clase que implementa el compilador para el lenguaje. Instancia y orquesta los analizadores Léxico, Sintáctico, Semántico y el Generador de Código, los cuales están implementados en otros módulos. Ofrece la siguiente API:

- **public JavaProgram compile(String source):**

  Compila una especificación TESL en forma de String

- **public JavaProgram compile(File file):**

  Compila una especificación TESL contenida en un fichero

- **public JavaProgram compile(InputStream stream):**

  Compila una especificación TESL proveniente de un InputStream

4.4 Analizador Léxico

El analizador léxico es el módulo encargado de transformar la secuencia de caracteres del programa fuente de entrada en una secuencia equivalente de tokens del lenguaje.

El analizador léxico también es el encargado de detectar los posibles errores léxicos. Estos errores se producen cuando el programa fuente contiene una secuencia de caracteres que no encajan con el patrón de ninguno de los tokens

del lenguaje. En tal caso se ha optado por no continuar con el análisis; el analizador terminará la ejecución del compilador mediante una excepción.

4.4.1 Implementación

A continuación se describen las diferentes clases que conforman la implementación del módulo Analizador Léxico.

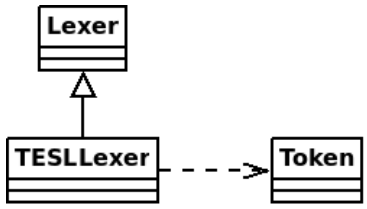

Figura 4.5: Diagrama de clases del módulo Analizador Léxico

TESLLexer (es.upm.tesl.core.TESLLexer)
Clase que implementa el analizador léxico para el lenguaje. Es generada automáticamente mediante la herramienta ANTLR a partir de la gramática del lenguaje y realiza el análisis léxico mediante un autómata finito determinista. Ofrece la siguiente API:

- **public Token nextToken():**
  Devuelve el siguiente token del programa fuente de entrada

- **public List<? extends Token> getAllTokens():**
  Devuelve todos los tokens del programa fuente de entrada

- **public int getLine():**
  Devuelve el número de línea del programa fuente en la que se encuentra el analizador léxico en ese momento.

Token (org.antlr.runtime.v4.Token)
Interfaz para los tokens del lenguaje. Es parte de la librería de entorno de ejecución de la herramienta ANTLR y usada en el código generado.

Lexer (org.antlr.v4.Lexer)
Clase abstracta para los analizadores léxicos generados mediante la herramienta ANTLR. Es parte de su librería de entorno de ejecución.

4.5 Analizador Sintáctico

El analizador sintáctico es el módulo encargado de transformar una secuencia de tokens de un programa fuente en el lenguaje TESL en un Árbol Sintáctico Concreto, cuyos nodos se corresponden con las construcciones del lenguaje. Para ello se basa en un conjunto de reglas que generan todas las construcciones que son válidas en el lenguaje.

El analizador sintáctico se encarga también de detectar los errores sintácticos, si los hubiera. Este tipo de errores se producen cuando una secuencia de tokens no corresponde con ninguna de las construcciones expresadas en las reglas gramaticales del lenguaje. De igual manera que con el analizador léxico, ante este tipo de errores se ha optado por finalizar la ejecución del compilador.

4.5.1 Implementación

A continuación se describen las diferentes clases que conforman la implementación del módulo Analizador Sintáctico.

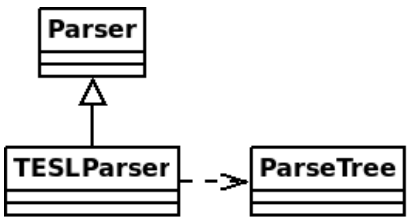

Figura 4.6: Diagrama de clases del módulo Analizador Sintáctico

TESLParser (es.upm.tesl.core.TESLParser)
Clase que implementa el analizador sintáctico para el lenguaje. Es generada automáticamente a partir de la gramática mediante la herramienta ANTLR e implementa el algoritmo de análisis sintáctico LL(*). Ofrece la siguiente API:

- **public ParseTree program():**
  Devuelve el Árbol Sintáctico Concreto construido a partir del nodo raíz *program*, usando como tokens los emitidos por el analizador léxico con el que ha sido inicializado.

Parser (org.antlr.v4.runtime.atn.Parser)
Clase abstracta usada por el analizador sintáctico generado por ANTLR. Es parte de su librería de entorno de ejecución.

ParseTree(org.antlr.v4.runtime.tree.ParseTree)
Interfaz para los Árboles Sintácticos Concretos construidos por el analizador sintáctico. Es parte de la librería de entorno de ejecución del ANTLR.

4.6 Analizador Semántico

El Analizador Semántico es módulo encargado de comprobar que se respetan las acciones semánticas del lenguaje, así como de recolectar cierta información de utilidad para la generación de código. Sus tareas son, además del cálculo del tipo de cada construcción del lenguaje, el manejo de los ámbitos del programa y el registro del tipo de los identificadores o símbolos que se declaren.

El Analizador Semántico trabaja sobre el Árbol Sintáctico Concreto construido por el Analizador Sintáctico. Por ello, es natural que sea implementado siguiendo el patrón Visitante, recorriendo los nodos del árbol en post-orden (primero los nodos hijos y después el nodo padre) y comprobando las reglas semánticas correspondientes a cada uno. El analizador implementa un método visitante para cada tipo de nodo. El resultado de la ejecución de este método es

el tipo calculado para ese nodo, el cual es usado recursivamente para calcular el tipo de su nodo padre. Esté tipo se anota en el nodo para su uso en etapas posteriores del compilador.

Cuando el analizador semántico encuentra la definición de un nuevo identificador, lo añade ámbito actual. Se ha optado por implementar los ámbitos mediante un árbol. El nodo raíz del árbol corresponde el ámbito global y sus hijos, los diferentes ámbitos locales. Cuando se entra en un nuevo ámbito, simplemente se crea un nuevo hijo y se define el actual como su padre; para salir del mismo, se define al padre del ámbito actual como el nuevo ámbito actual. Este árbol de ámbitos será usado por etapas posteriores de la compilación.

Este diseño tiene la ventaja de que todo el código relacionado con el análisis semántico está contenido en una única clase, el analizador. De igual manera pueden construirse otros visitantes que realicen otro tipo de procesamiento del árbol, como un optimizador o el Generador de Código.

Aunque es frecuente el uso de los árboles sintácticos abstractos, se optó por trabajar sobre un árbol concreto ya que era suficientemente bueno para la tarea requerida y permite utilizar más código generado mediante ANTLR.

Igualmente que en los módulos anteriores, cuando el analizador detecta un error semántico, como por ejemplo el uso de un tipo no permitido en una operación o la llamada a una función con un número de argumentos erróneos, se termina la ejecución del compilador mediante una excepción.

4.6.1 Implementación

A continuación se describen las diferentes clases que conforman la implementación del módulo Analizador Semántico

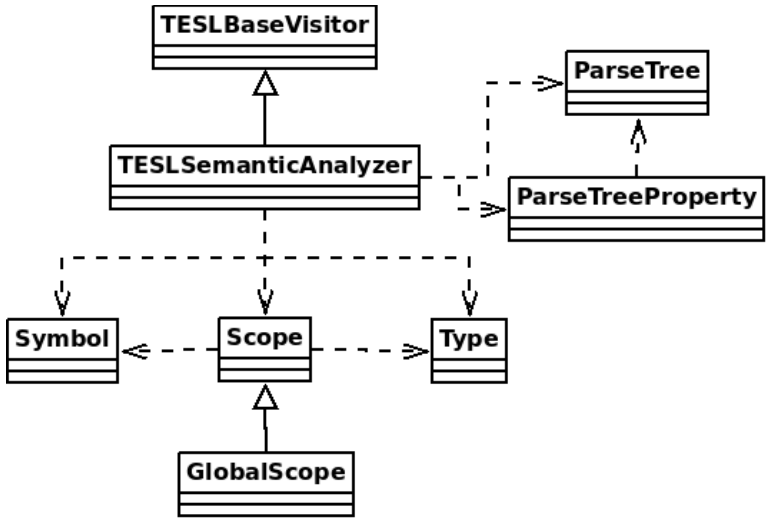

Figura 4.7: Diagrama de clases del módulo Analizador Semántico

TESLBaseVisitor (es.upm.tesl.core.TESLBaseVisitor)
Clase abstracta que implementa un visitante para los Árboles Sintácticos Concretos del lenguaje TESL. Es generada automáticamente mediante la herramienta ANTLR a partir de la gramática y su implementación sigue el patrón *External Tree Visitor*[9]. Contiene un método para cada tipo de nodo del árbol, los cuales son llamados con el nodo como argumento cuando el analizador visita esos nodos.

TESLSemanticAnalyzer (es.upm.tesl.core.TESLSemanticAnalyzer)
Clase que implementa el Analizador Semántico del lenguaje. Extiende la clase TESLBaseVisitor, sobrescribiendo los métodos para cada tipo de nodo. De esta manera, en cada uno de esos métodos se implementan las reglas semánticas correspondientes a ese nodo, devolviendo tras su ejecución el tipo TESL calculado para el mismo. Ofrece la siguiente API:

- **public Pair<ParseTreeProperty<Scope>, ParseTreeProperty<Type>> analyze(ParseTree tree)**
  Analiza semánticamente el árbol *tree*. Para ello crea y gestiona los diferentes ámbitos y calcula los tipos de los nodos, los cuales son

anotados a los mismos mediante las estructuras ParseTreeProperty. Devuelve las estructuras anotadas.

Symbol (es.upm.tesl.scope.Symbol)
Clase que implementa un identificador o símbolo del lenguaje. Cada uno de ellos está compuesto por un nombre y un tipo TESL.

Scope (es.upm.tesl.scope.Scope)
Clase que implementa los ámbitos del lenguaje TESL. Para asociar cada símbolo a un nombre de manera eficiente se usa una Tabla Hash. Ofrece los siguientes métodos:

- **public Scope getParent():**
  Devuelve el ámbito padre del mismo

- **public void define(Symbol symbol):**
  Asocia un símbolo a su nombre

- **public Symbol resolve(String name):**
  Devuelve el símbolo asociado a un nombre, resolviéndolo en el ámbito padre si no está definido en el ámbito actual.

- **public Symbol resolveLocal(String name):**
  Devuelve el símbolo asociado a un nombre, resolviéndolo únicamente en el ámbito actual.

GlobalScope (es.upm.tesl.scope.GlobalScope)
Clase que implementa el ámbito global del lenguaje TESL. Extiende la clase Scope y predefine todos los símbolos básicos del lenguaje, como por ejemplo los símbolos de las funciones matemáticas del lenguaje.

ParseTreeProperty (org.antlr.v4.runtime.ParseTreeProperty<V>)
Estructura de datos la cual permite asociar un nodo de un Árbol Sintáctico Concreto a un objeto. Es similar a una Tabla Hash. Es usada por el Analizador

Semántico para asociar un *scope* a un nodo y poder pasar esta información a fases posteriores del compilador. Ofrece la siguiente API:

- **public V get(ParseTree node):**
  Devuelve el objeto asociado a un nodo

- **public void put(ParseTree node, V val):**
  Asocia un objeto a un nodo

- **public void removeFrom(ParseTree node):**
  Elimina la asociación entre un objeto y un nodo

4.7 Sistema de Tipos

El lenguaje TESL tiene los siguientes tipos: Boolean, Integer, Float, Point, Event, EventClass, EventSpec, TimeSeries, Function, PointSet y EventSet. Los tipos EventSpec y EventClass corresponden, respectivamente, al tipo del símbolo que define el nombre del programa y al tipo del símbolo que define una clase de evento.

No todos tienen las mismas propiedades. Algunos de ellos pueden ser operados en operaciones aritméticas, transformándose de unos a otros, estos son los tipos Numeric. Otros sirven como espacio de nombres para métodos y atributos y son denominados Namespace. Por último, algunos de ellos contienen elementos en su interior los cuales pueden iterarse, son los tipos Iterable.

Debido a estas propiedades se optó por un diseño usando una clase para cada uno de los tipos, la cual implementa el interfaz Type, además de los interfaces Numeric, Namespace o Iterable según la clase de tipo que sea. Los metadatos de los tipos, como por ejemplo el tipo del valor devuelto de una función o el tipo de los elementos contenidos en un conjunto, son alojados en los tipos instanciados.

4.7.1 Conversión de tipos

En el lenguaje TESL los tipos Integer, Point y Float pueden ser usados indistintamente en operaciones aritméticas. Cuando se realiza una operación entre tipos diferentes, el tipo resultante de esta operación se calcula siguiendo la siguiente tabla de conversiones:

	Integer	**Point**	**Float**
Integer	Integer	Integer	Float
Point	Integer	Point	Float
Float	Float	Float	Float

Tabla 4.1: Tabla de conversiones de tipos

4.7.2 Implementación

A continuación se describen las diferentes clases que conforman la implementación del módulo Sistema de Tipos

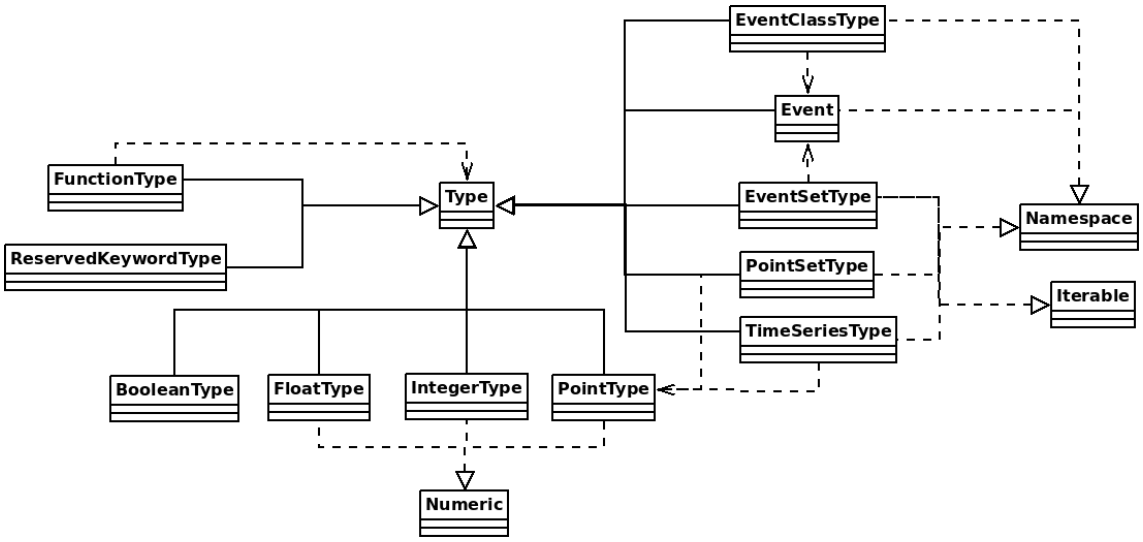

Figura 4.8: Diagrama de clases del Sistema de Tipos

Type (es.upm.tesl.type.Type)
Clase abstracta que implementa un tipo del lenguaje. Todos los heredan de esta clase y aunque no ofrece ningún método, se usa como etiqueta para agruparlos. Incluye una serie de constantes para cada uno de los tipos que son siempre idénticos. Esto tiene la finalidad de evitar la creación de objetos innecesariamente durante la ejecución del compilador. Así por ejemplo, si queremos definir un tipo Boolean, no creamos una nueva instancia de la clase Boolean si no que usamos la constante Type.Boolean. Debe tenerse en cuenta que para ciertos tipos como Function, es necesario crear una nueva instancia de los mismos pues cada uno de ellos es diferente entre sí.

Numeric (es.upm.tesl.type.Numeric)
Interfaz para un tipo numérico del lenguaje. Los tipos que lo implementan son Integer, Float y Point. Ofrece la siguiente API:

- **public Type calculate(Op op, Numeric that):**
  Devuelve el tipo resultado de operar el tipo this con el tipo that mediante el operador OP op. Para ello usa la tabla de conversión anteriormente expuesta.

Namespace (es.upm.tesl.type.Namespace)
Interfaz para un tipo el cual contiene un ámbito interno (local) donde están definidas funciones y variables. Los tipos que lo implementan son TimeSeries, EventClass, Event, y PointSet y EventSet. Ofrece la siguiente API:

- **public Symbol resolve(String name):**
  Devuelve el símbolo asociado a un nombre en el scope interno.

Iterable (es.upm.tesl.type.Iterable)
Interfaz para un tipo el cual contiene otros elementos tipados en su interior. Los tipos que lo implementan son TimeSeries y PointSet y EventSet. Expone la siguiente API:

- **public Type getElementType():**
  Devuelve el tipo de los elementos que contiene.

BooleanType (es.upm.tesl.type.Boolean)
Clase que implementa el tipo booleano. Extiende de la clase Type.

FloatType (es.upm.tesl.type.Float)
Clase que implementa el tipo en coma flotante. Extiende de la clase Type e implementa el interfaz Numeric.

IntegerType (es.upm.tesl.type.Integer)
Clase que implementa el tipo entero. Extiende de la clase Type e implementa el interfaz Numeric.

PointType (es.upm.tesl.type.Point)
Clase que implementa el tipo punto. Extiende de la clase Type e implementa el interfaz Numeric.

EventType (es.upm.tesl.type.Event)
Clase que implementa el tipo evento. Extiende de la clase Type e implementa el interfaz Namespace.

PointSetType (es.upm.tesl.type.PointSetType)
Clase que implementa el tipo conjunto de puntos. Extiende de la clase Type e implementa el interfaz Iterable.

EventSetType (es.upm.tesl.type.EventSetType)
Clase que implementa el tipo conjunto de eventos. Extiende de la clase Type e implementa el interfaz Iterable.

TimeSeriesType (es.upm.tesl.type.TimeSeriesType)
Clase que implementa el tipo serie temporal. Extiende de la clase Type e implementa el interfaz Iterable.

EventClassType (es.upm.tesl.type.EventClassType)
Clase que implementa el tipo clase de eventos. Es usada para los símbolos que identifican las diferentes definiciones de eventos. Extiende de la clase Type e implementa el interfaz Namespace.

FunctionType (es.upm.tesl.type.PointSetType)
Clase que implementa el tipo de una función o método del lenguaje. Extiende de la clase Type.

ReservedKeywordType (es.upm.tesl.type.ReserverKeywordType)
Clase usada para el tipo de las palabras reservadas del lenguaje. Además de las propias palabras reservadas de TESL, también están reservadas todas las de lenguaje Java. Esto se hace para no crear conflictos en el código generado.

4.8 Generador de código

El Generador de Código es el módulo encargado de transformar el Árbol Sintáctico Concreto que representa el programa fuente de entrada en un programa Java equivalente, el cual permite extraer, de las series temporales indicadas en el programa fuente, los eventos que contienen.

De igual manera que el Analizador Semántico, el Generador de Código está implementado siguiendo el patrón visitante. Haciendo uso de los ámbitos y tipos calculados en fases anteriores, el generador recorre los nodos del árbol en post-orden, generando la traducción en Java correspondiente a cada uno en forma de String. Estos Strings se concatenan recursivamente formando el código objeto del programa Java de salida.

Un aspecto a tener en cuenta es que debido a las diferencias entre el lenguaje fuente TESL y el lenguaje objeto Java, los algoritmos implementados en uno y otro son muy diferentes y no es posible realizar una traducción literal salvo en el caso de unas pocas expresiones.

4.8.1 Traducciones

Tal como se comentó anteriormente, los lenguajes TESL y Java son profundamente diferentes. TESL es un lenguaje declarativo mientras que Java es imperativo. Debido a ello, es necesario transformar los algoritmos usados en el programa fuente a unos equivalentes en el programa objeto. A continuación se detallan las traducciones para las diferentes construcciones del lenguaje.

Declaración EventSpec
Esta construcción define el nombre de la especificación y sirve como espacio de nombres o paquete. Es traducida a una clase en Java la cual implementa el interfaz EventSpec de la librería de entorno de ejecución. En su interior se encuentra definido el resto del programa.

```
eventspec cardio;
```

Listado 4.1: Declaración EventSpec en TESL

```
public class Cardio implements EventSpec {
    ...
    public Cardio(TimeSeries[] timeseries) {
        ...
    }
    ...
}
```

Listado 4.2: Traducción de la declaración EventSpec

Declaración de serie temporal
Esta construcción declara una serie temporal para ser usada en el resto de programa TESL. Su traducción se sitúa dentro del constructor de la clase Java generada para la declaración EventSpec. Consiste en la declaración de una variable TimeSeries, su inicialización con una de las series temporales recibida

como argumento en el constructor y la asociación de esta variable a su nombre en la tabla hash de series temporales.

```
timeseries cardio;
```

Listado 4.3: Declaración de una serie temporal en TESL

```
TimeSeries cardio = new timeseries[0];
this.timeseries.put("cardio", cardio);
```

Listado 4.4: Traducción de la declaración de una serie temporal

Expresiones

Las expresiones en TESL son similares a las expresiones en Java, por lo que algunas de ellas pueden traducirse literalmente. Las expresiones aritméticas y lógicas pueden traducirse literalmente. Las expresiones de conjuntos mediante los operadores *intersec*, *join* y *diff*, son traducidas por llamadas a métodos de la clase Set. Las expresiones de intervalos mediante los operadores *begins*, *ends*, *covers* y *exists* son traducidos por llamadas a métodos de la clase Event.

```
zero intersec mins
```

Listado 4.5: Expresión de conjuntos en TESL

```
zero.intersec(mins)
```

Listado 4.6: Traducción de la expresión de conjuntos

```
faliure begins [0, 10]
```

Listado 4.7: Expresión de intervalos en TESL

```
faliure.begins(0, 10)
```

Listado 4.8: Traducción de la expresión de intervalos

Definición de conjuntos de puntos y eventos

La definición de conjuntos en TESL puede traducirse por instancias de la clase PointSet o EventSet de la librería de entorno de ejecución, inicializadas mediante un bucle for. La cláusula de guarda de la definición puede traducirse por una simple sentencia if. La complejidad del código generado es O(n) con respecto a la longitud de la colección usada en la inicialización.

Un aspecto a destacar de la traducción es que los nombres de las variables usadas en los bucles for se modifican añadiéndoles el carácter '_'. Esto se hace para evitar problemas en código generado, debido a las diferencias entre el concepto de ámbito de TESL y Java. De no hacerse podrían producirse conflictos de redeclarado de variables en el código Java.

```
pointset Zero {
    point x in cardio such that cardio.value(x) == 0
};
```

Listado 4.9: Definición de un conjunto de puntos en TESL

```
PointSet<Integer> zero = new PointSet<Integer>();
for (Integer _x : cardio) {
    if (cardio.value(_x) == 0) zero.add(_x);
}
```

Listado 4.10: Traducción de la definición del conjunto de puntos

Definición de evento básico

Un evento básico en TESL es aquel que está definido mediante tres puntos denominados *start*, *peculiar* y *end*. Estos puntos deben respetar la invariante "start <= peculiar <= end". Además, estos puntos pueden provenir de colecciones en cualquier orden. Por esta razón se traduce por el producto cartesiano de las tres colecciones de donde provienen, guardado por una condicional para la invariante. La complejidad del mismo es por lo tanto $O(n^3)$.

Para crear los eventos en sí, primeramente se crea e inicializa y registra en la tabla hash correspondiente una instancia de EventClass con el nombre de la clase. Esta clase se usa para crear los eventos mediante el método Create.

De igual manera que en el caso de la declaración de conjuntos, los nombres de las variables *start*, *end* y *point* se modifican añadiéndoles el carácter '_'.

```
event QRS {
    point start in q,
    point peculiar in r,
    point end in s
    such that q.previous(peculiar) == start && s.next(peculiar) == end
};
```

Listado 4.11: Definición de un evento básico en TESL.

```
EventClass qrs = new EventClass("qrs");
eventClasses.put("qrs", qrs);
for (Integer _start : q) {
    for (Integer _peculiar : r) {
        for (Integer _end : s) {
            if (_start > _peculiar || _peculiar > _end) continue;
            if (q.previous(_peculiar) == _start && s.next(_peculiar) == _end) {
                qrs.create(_start, _peculiar, _end);
            }
        }
    }
}
```

Listado 4.12: Traducción de la definición de un evento básico en TESL

El motivo del uso de la clase EventClass en vez realizar la traducción creando una clase interna QRS en Java es que lo único que diferencia las clases de eventos en TESL es su nombre. Crear una clase interna para cada definición aumentaría enormemente el tamaño y duplicación de código sin aportar ningún valor. Por esta razón se optó por este diseño con la tabla hash eventClasses y la clase EventClass usada como factoría. Más detalles sobre esta implementación pueden encontrarse en el capítulo de la Librería de Entorno de Ejecución.

Definición de evento compuesto

Un evento compuesto en TESL es aquel que está definido mediante uno o más eventos. De manera similar a como se hace en la definición de eventos básicos, se traduce por el producto cartesiano de las fuentes de elementos que lo conforman. Puesto que un evento compuesto puede estar formado por n otros eventos provenientes de n fuentes diferentes, la complejidad del código traducido es $O(n^n)$.

Igualmente y por las mismas razones que en el caso de los eventos básicos, la traducción hace uso de la clase EventClass para crear y guardar los eventos.

```
event faliure {
    event e in peak.set()
    such that errors.slice(e.start, e.end).sum() > 10
};
```

Listado 4.13: Ejemplo definición de evento compuesto en TESL

```
EventClass faliure = new EventClass("faliure");
eventClasses.put("faliure", faliure);
for (Event _e : peak.set()) {
    if (errors.slice(_e.start(), _e.end().sum() > 10) faliure.create(_e);
}
```

Listado 4.14: Traducción de definición de evento compuesto

4.8.2 Implementación

A continuación se muestra la descripción de las clases que conforman la implementación del módulo Generador de Código.

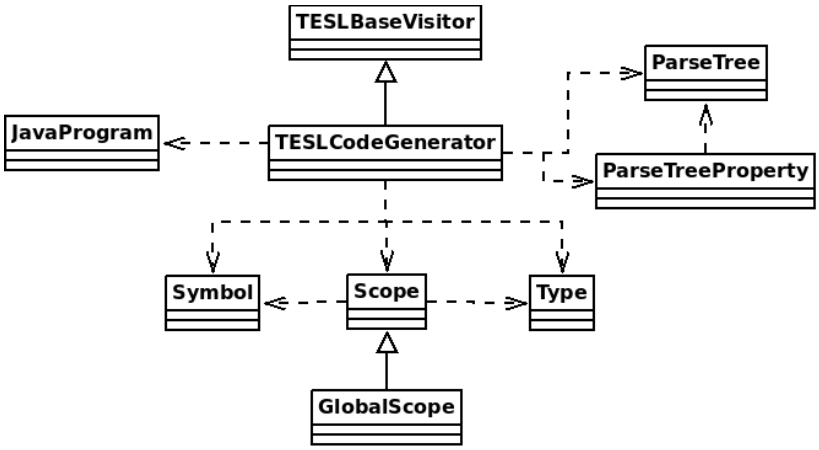

Figura 4.9: Diagrama de clases del módulo Generador de Código

TESLCodeGenerator (es.upm.tesl.core.TESLCodeGen)
Clase la cual implementa el Generador de Código encargado de realizar la traducción del Árbol Sintáctico Concreto del lenguaje al programa Java de salida. Hereda de la clase TESLBaseVisitor la cual fue descrita en el apartado del Analizador Semántico. Ofrece la siguiente interfaz:

- **public JavaProgram generate(ParseTree tree,**
  Pair<ParseTreeProperty<Scope>,
  ParseTreeProperty<Type>> metadata):
  Recibe como argumento el Árbol Sintáctico Concreto *tree* y sus ámbitos y tipos anotados, devolviendo un programa Java equivalente.

JavaProgram (es.upm.tesl.java.JavaProgram)
Clase que representa un programa en Java. En su interior contiene el código fuente en forma de String y el nombre del programa. Es usada debido a que en Java el fichero de un programa debe tener el mismo nombre que la clase que contiene en su interior.

4.9 Librería de entorno de ejecución

El código Java generado hace uso de una librería de entorno de ejecución en la cual se encuentran definidas todas las clases y métodos en Java que representan las construcciones del lenguaje TESL. Esta librería es importada al inicio de los programas generados, por lo que para compilar los mismos es necesario añadirla al *class path* a la hora de compilar el código Java generado.

El lenguaje TESL dispone además de una serie de funciones matemáticas. Estas funciones no se encuentran en la librería de entorno de ejecución debido a que el código generado hace uso de la librería estándar Math de Java.

4.9.1 Implementación

A continuación se describen las clases que conforman la implementación del módulo Librería de Entorno de Ejecución.

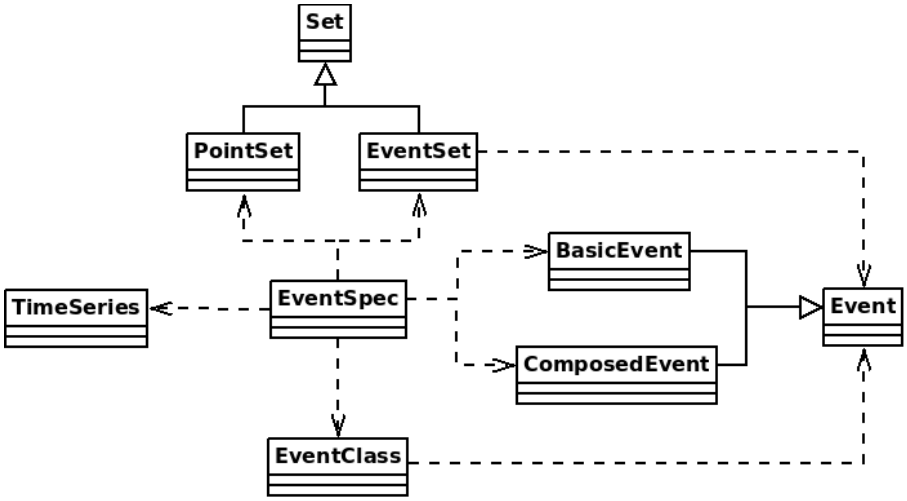

Figura 4.10: Diagrama de clases del módulo Librería de Entorno de Ejecución

EventSpec (es.upm.tesl.runtime.eventspec)
Interfaz que representa una especificación de eventos e implementan las clases generadas en el código java. Ofrece la siguiente API:

- **public EventSet getAllFrom(TimeSeries ts):**
  Devuelve un conjunto con todos los eventos encontrados en la serie temporal ts

TimeSeries (es.upm.tesl.runtime.TimeSeries)
Clase que implementa una serie temporal. Internamente contiene un vector en el cual están contenidos todos los valores de la serie temporal para cada instante de tiempo. Debe tenerse en cuenta que los instantes temporales se cuentan a partir de un mismo punto de comienzo, indexado en el punto 0. La clase ofrece la siguiente API:

- **public double value(double x):**
  Devuelve el valor de la serie temporal en el instante x

- **public PointSet maxs():**
  Devuelve los máximos locales de la serie temporal.

- **public PointSet mins():**
  Devuelve los mínimos locales de la serie temporal.

- **public double max():**
  Devuelve el valor máximo de la serie

- **public double mean():**
  Devuelve el valor medio de la serie

- **public double median():**
  Devuelve el valor mediano de la serie

- **public double min():**
  Devuelve el valor mínimo de la serie

- **public double mode():**
  Devuelve el valor modal de la serie

- **public double stdev():**
  Devuelve la desviación estándar de la serie

- **public double sum():**
  Devuelve la suma de los valores de la serie

- **public double sumsq():**
  Devuelve la suma de cuadrados de los valores de la serie

- **public double variance():**
  Devuelve la varianza de la serie

- **public TimeSeries slice(int a, int b):**
  Devuelve una nueva serie temporal formada por los valores de la serie comprendidos entre los puntos a y b.

Set (es.upm.tesl.runtime.Set)
Clase abstracta que implementa un conjunto en TESL. Debido a que en el lenguaje los conjuntos son ordenados, es una subclase de la clase estándar java.util.TreeSet. Para poder mantener el orden, los elementos que contiene deben implementar el interfaz java.lang.Comparable. Está implementada usando genéricos y ofrece la siguiente API:

- **public void add(E e):**
  Añade el elemento e al conjunto

- **public boolean isEmpty():**
  Devuelve verdadero si el conjunto está vacío

- **public boolean isFirst(E e):**
  Devuelve verdadero si e es el elemento más pequeño del conjunto

- **public boolean isLast(E e):**
  Devuelve verdadero si e es el elemento más grande del conjunto

- **public E prev(E e):**
  Devuelve el elemento previo a e en el conjunto

- **public E next(E e):**
  Devuelve el elemento siguiente a e en el conjunto

- **public E min():**
  Devuelve el elemento más pequeño del conjunto

- **public E max():**
  Devuelve el elemento más grande del conjunto

PointSet (es.upm.tesl.runtime.PointSet)
Clase que implementa un conjunto de puntos en TESL. Hereda de la clase Set.

EventSet (es.upm.tesl.runtime.EventSet)
Clase que implementa un conjunto de eventos en TESL. Hereda de la clase Set.

EventClass (es.upm.tesl.runtime.EventClass)
Clase que representa una clase de eventos. Contiene un nombre para la misma y sirve de fábrica para los eventos de su clase, creándolos y registrándolos en el conjunto que contiene. Se optó por implementar los eventos de esta manera en vez de utilizar realmente clases en Java debido a que en realidad solo hay dos clases de eventos, básicos y compuestos, y lo único que diferencia las diferentes clases es el nombre. Ofrece la siguiente interfaz:

- **public name():**
  Devuelve el nombre de la clase

- **public set():**
  Devuelve un set con todos los eventos creados a partir de esta clase

- **public create(int start, int peculiar, int end):**
  Crea y devuelve un evento básico de esta clase

- **public create(Event ... events):**
  Crea y devuelve un evento compuesto de esta clase

Event(es.upm.tesl.runtime.Event)
Clase abstracta que implementa las funcionalidades comunes a los eventos en TESL. Ofrece la siguiente API:

- **public String name():**
  Devuelve el nombre del evento

- **public int start():**
  Devuelve el punto de inicio del evento

- **public int end():**
  Devuelve el punto final del evento

- **public boolean begins(int a, int b):**
  Devuelve verdadero si el evento comienza entre los instantes a y b

- **public boolean ends(int a, int b):**
  Devuelve verdadero si el evento termina entre los instantes a y b

- **public boolean covers(int a, int b):**
  Devuelve verdadero si el evento cubre completamente el rango de tiempos entre los instantes a y b

- **public boolean exists(int a, int b):**
  Devuelve verdadero si en alguno de los instantes de tiempo contenidos entre el comienzo y el final del evento se encuentra entre los instantes de tiempo a y b

BasicEvent (es.upm.tesl.runtime.BasicEvent)
Clase que implementa un evento básico en TESL y hereda de la clase Event. Un evento básico es aquel que se define por tres puntos denominados *start*, peculiar y *end*. Además de los métodos heredados de Event, ofrece la siguiente API:

- **public int peculiar():**
  Devuelve el punto peculiar del evento

ComposedEvent (es.upm.tesl.runtime.ComposedEvent)
Clase que implementa un evento compuesto en TESL y hereda de la clase Event. Un evento compuesto es aquel que está definido por uno o más eventos. Además de los métodos heredados de Event, ofrece la siguiente API:

- **public Event[] events():**
  Devuelve los eventos que lo forman

5 Caso de uso

En este capítulo se van a demostrar algunas de las principales funcionalidades del compilador mediante su uso en la ejecución de una especificación TESL para el dominio de las métricas de servidores.

Se han generado artificialmente tres tipos de métricas que simulan algunas de las que se podrían recolectar durante 5 minutos de operación de un servidor HTTP. Estas métricas son el número de peticiones por segundo (Figura 5.1), número de errores (HTTP 500) por segundo (Figura 5.2) y tiempo de respuesta medio por petición (Figura 5.3). Cada una de estas series temporales tiene 300 valores, correspondientes a cada segundo en la ventana de tiempo de 5 minutos anteriormente mencionada.

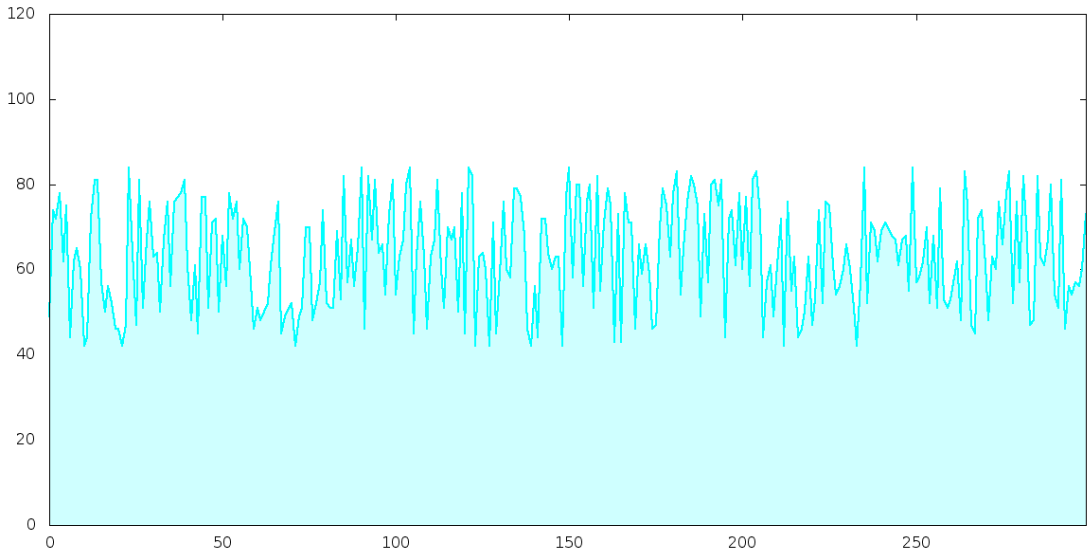

Figura 5.1: Número de peticiones por segundo

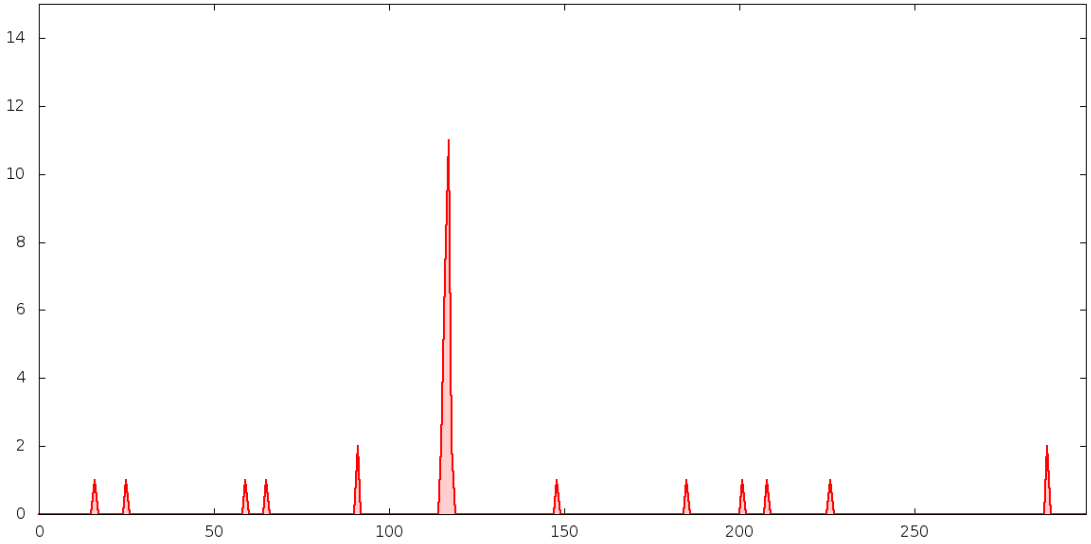

Figura 5.2: Número de errores (HTTP 500) por segundo

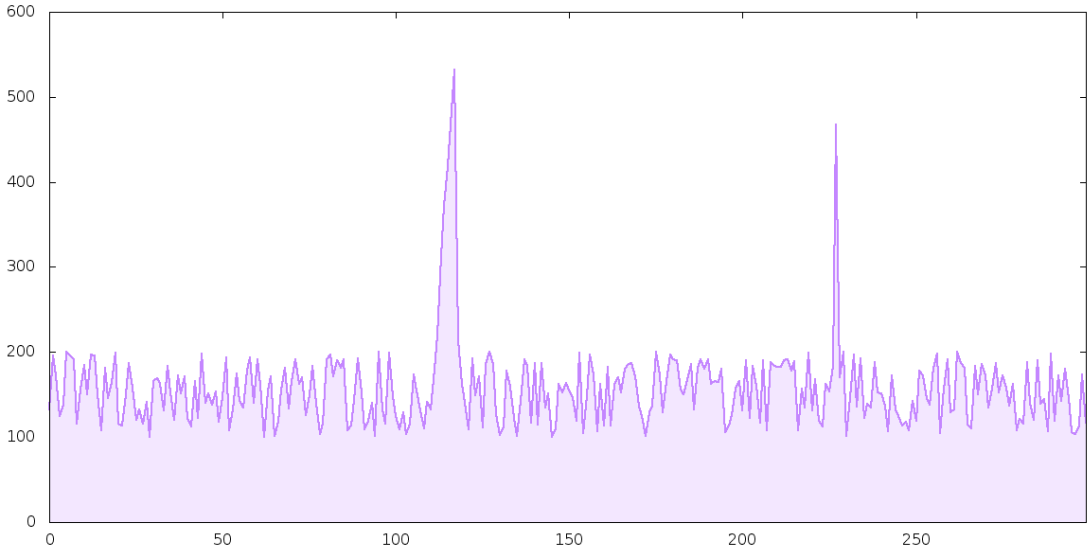

Figura 5.3: Tiempo medio de respuesta (ms)

El objetivo de la especificación en TESL para el caso propuesto es detectar posibles problemas en la calidad del servicio del servidor. Para ello se van a identificar los picos en el tiempo de respuesta durante los cuales se produce un ratio anormal de errores. A continuación se muestra la especificación TESL que realiza la tarea propuesta:

```
// se declara el nombre de la especificación
eventspec metrics;

// se declaran las series temporales usadas
timeseries numRequests;
timeseries requestTimes;
timeseries numErrors;

// se obtienen los máximos y mínimos relevantes
pointset maxs {
    x in requestTimes
    such that
    requestTimes.value(x) > 2*(requestTimes.mean() + requestTimes.stdev())
};

poinset mins {
    x in requestTimes.localmins()
};

// se obtienen los picos en el tiempo de respuesta. Para ello buscamos
// los mínimos más cercanos al máximo correspondiente
event time_peak {
    start in mins, peculiar in maxs, end in mins
    such that start == mins.prev(peculiar) && end == mins.next(peculiar)
};

// se obtienen aquellos picos en los que se produce un ratio anormal de errores
event malfunction {
    event e in time_peak
    such that
        (numErrors.slice(e.start, e.end).sum()
        / numRequests.slice(e.start, e.end).sum()) >= 0.2
};
```

Listado 5.1: Especificación metrics.tesl

Para ejecutar la especificación anterior se va a hacer uso del comando *Run* del compilador. Este comando permite ejecutar una especificación de eventos indicando sobre qué conjunto de datos se ejecutará, siendo este conjunto un fichero en formato csv que contiene las series temporales. Tal y como se describió en el subcapítulo dedicado a la interfaz por línea de comandos, el comando *Run* hace uso del compilador de Java y sus mecanismos de reflexión para compilar y cargar dinámicamente el código generado.

```
$ tesl run -l metrics.tesl metrics.csv
requesttimepeak: 2
  [110, 121]
  [225, 228]
failure: 1
  [110, 121]
```

Listado 5.2: Ejecución del compilador para la especificación metrics.tesl

El resultado de la ejecución es un sumario con los eventos encontrados. Como era de esperar, el programa en TESL ha sido capaz los dos momentos en los cuales se produce un pico en el tiempo de respuesta (*requesttimepeak*) y en cual de ellos se produce además un ratio anormal de errores (*failure*).

6 Conclusiones

En este trabajo se ha descrito el diseño e implementación de un compilador para el lenguaje de dominio específico TESL y su librería de entorno de ejecución. Se han documentado las decisiones de diseño que han llevado a la arquitectura del mismo, así como los diferentes detalles de su implementación. Además, se ha documentado la versión actual del lenguaje TESL y se ha colaborado en el diseño del mismo.

El compilador ha sido diseñado con la idea en mente de permitir la fácil extensión del lenguaje, incluyendo unos Analizadores Léxico y Sintáctico autogenerados mediante la herramienta ANTLR y siguiendo las mejores prácticas en cuanto a su arquitectura. Su generador de código de alto nivel permite el fácil desarrollo de nuevas construcciones del lenguaje, además de simplificar la posible extensión y mejora de su librería de entorno de ejecución.

Este compilador se ha concebido como un prototipo que sienta las bases para la continuación del desarrollo del lenguaje. Permite probar especificaciones TESL y ver cómo funcionan los diferentes aspectos del lenguaje, así como identificar posibles puntos de mejora o debilidades del mismo. Al mismo tiempo, es una herramienta que facilita en gran medida el análisis de series temporales, al permitir transformar, automáticamente, especificaciones de alto nivel en código ejecutable capaz de analizar las series temporales y extraer los eventos que contienen.

El análisis de series temporales es un área de gran relevancia en la cual hay necesidad de soluciones específicas y espacio para la innovación, como demuestran los numerosos productos comerciales que han aparecido en los últimos años. El lenguaje TESL tiene la oportunidad de encontrar un nicho de mercado en este área y el presente compilador es la herramienta para ello.

7 Líneas de trabajo futuras

Hay diferentes direcciones en las cuales puede mejorarse el presente compilador, principalmente enfocadas en el sistema de tipos y el generador de código, siempre teniendo en cuenta la posible extensión y mejora del lenguaje TESL.

El sistema de tipos actual podría hacerse más avanzado y flexible. La versión actual del lenguaje TESL no lo requiere, pero la previsión es que el lenguaje siga desarrollándose y acabe requiriendo un sistema de tipos más sofisticado.

En cuanto al generador de código de alto nivel a Java, es ideal para el rápido desarrollo del lenguaje pero, una vez el mismo se estabilizase, podrían considerarse otro tipo de generadores más avanzados. Por ejemplo, un generador de código intermedio IR para la herramienta LLVM permitiría el desarrollo de un intérprete JIT. Esto eliminaría la dependencia del compilador con el del lenguaje Java, así como permitiría realizar todo tipo de optimizaciones que no son posibles generando código de alto nivel.

Otra posibilidad sería el uso de una base de datos para almacenar las series temporales y adaptar el lenguaje TESL a consultas a esta base de datos. Esto permitiría el uso de TESL junto con grandes volúmenes de datos y posibilitando así su uso en más dominios.

Por último, podría rediseñarse la librería de entorno de ejecución para intentar mejorar la eficiencia de la misma o para añadir más funcionalidades. La práctica en el uso del lenguaje TESL llevará probablemente al descubrimiento de necesidades que no estén cubiertas por la librería actual.

8 Bibliografía

[1] J. A. Lara, A. López-Illescas, A. Pérez, J. P. Valente, “A Language for Defining Events in Multi-Dimensional Time Series: Application to a Medical Domain”. 1st International Workshop on Mining of Non-Conventional Data, Sevilla, España, 2009

[2] D. Martínez, “VIIP: Lenguaje de definición de eventos para series temporales. Traductor a C#”. Trabajo de Fin de Carrera, Facultad de Informática, Universidad Politécnica de Madrid, 2011.

[3] J. Álvarez, “Aplicación de lógica temporal a la definición, descubrimiento y análisis de sistemas software complejos”. Tesis de Máster, Facultad de Informática, Universidad Politécnica de Madrid. Pendiente de publicación.

[4] M. Ojeda, “Procesador de un lenguaje de especificación de eventos con extensión temporal”. Trabajo de fin de carrera, Facultad de Informática, Universidad Politécnica de Madrid, 2014.

[5] T. Norvell, A Short Introduction to Regular Expressions and Context Free Grammars. Typeset, 2002

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques & Tools, 2nd ed. Addison Wesley, 2006.

[7] K. B. Williams, Grace Hopper: Admiral of the Cyber Sea. Naval Institute Press, 2012

[8] Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L. & Nutt, R. "The FORTRAN automatic coding system." Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability. ACM, 1957.

[9] T. Parr, Language implementation patterns. Pragmatic Bookshelf, 2010.

[10] The Python Software Foundation, The Python Language Reference. The Python Software [En línea]. Disponible: https://docs.python.org/2/reference/

[11] S. C. Johnson. Yacc: Yet Another Compiler-Compiler. AT&T Bell Laboratories. [En línea]. Disponible: http://dinosaur.compilertools.net/yacc/

[12]GNU, GCC Front Ends. GNU, 2013. [En línea]. Disponible: http://gcc.gnu.org/frontends.html

[13] LLVM Developer Group, LLVM Users. LLVM Developer Group, 2009 [En línea]. Disponible: http://llvm.org/Users.html

[14] The GNU Project, Bison 3.0.2. The GNU Project. [Internet]. Disponible: http://www.gnu.org/software/bison/manual/bison.html

[15] T. Parr, The Definitive ANTLR4 Reference, 2nd ed. Pragmatic Bookshelf, 2013.

[16] NIST (2013), Handbook of Statistical Methods. NIST [En línea] Disponible: http://www.itl.nist.gov/div898/handbook/

[17] J. W. Lloyd"Practical advantages of declarative programming." Joint Conference on Declarative Programming, GULP-PRODE. Vol. 94. 1994.

[18] The InfluxDB Team, InfluxDB [En línea] Disponible: http://influxdb.com/

[19] SoundCloud Ltd, Roshi. [En línea] Disponible: https://github.com/soundcloud/roshi

[20] Square Inc, Cube [En línea] Disponible: https://github.com/square/cube

[21] Pandas Team, Pandas. [En línea] Disponible: http://pandas.pydata.org/

[22] R Development Core Team, R. [En línea] Disponible: http://www.r-project.org/

Anexo A: El lenguaje TESL

A.1. Notación

La notación usada para las definiciones de la sintaxis en esta especificación es la Notación de Backus-Naur Extendida (EBNF)

A.2. Tipos

En TESL un tipo se utiliza para identificar un conjunto de valores así como las operaciones que se pueden realizar sobre los mismos. El lenguaje está compuesto por 8 tipos predefinidos y no permite la definición de tipos por los usuarios. Los tipos son los siguientes:

Entero
El tipo Entero representa el conjunto de valores enteros de 32 bits. Se usa en el lenguaje mediante la palabra clave *int*.

Punto
El tipo es semánticamente equivalente al tipo entero y representa un instante de tiempo en una serie temporal. Se usa mediante la palabra clave *point*.

Coma flotante
El tipo Coma Flotante representa el conjunto de valores en coma flotante de 32 bits. Se usa en el lenguaje mediante la palabra clave *float*.

Booleano
El tipo Booleano representa los valores Booleanos verdadero y falso. Se usa en el lenguaje mediante la palabra clave *bool*.

Evento
El tipo Evento representa un evento de interés en una serie temporal. Un evento está compuesto por un nombre que lo identifica, así como por tres puntos denominados *start*, *peculiar* y *end* en caso de tratarse de un evento simple, o un conjunto de eventos en caso de tratarse de un evento compuesto. Se usa en el lenguaje mediante la palabra clave *event*.

Conjunto de puntos
El tipo Conjunto de Puntos representa un conjunto de valores de tipo Punto. Se usa en el lenguaje mediante la palabra clave *pointset*.

Conjunto de Eventos
El tipo Conjunto de Eventos representa un conjunto de valores de tipo Evento. Se usa mediante la palabra clave *eventset*.

Serie Temporal
El tipo Serie Temporal representa una secuencia de valores de tipo Coma Flotante indexados mediante valores de tipo Punto. Se usa mediante la palabra clave *timeseries* o alternativamente *ts*.

Función
El tipo Función representa una función del lenguaje. Está compuesto por unos argumentos tipados y un tipo devuelto. El lenguaje TESL no permite la declaración de funciones por parte de los usuarios pero dispone de múltiples funciones y métodos predeclarados.

A.3. Conversión de tipos

El lenguaje TESL tiene conversión de tipos automática entre operandos numéricos. A continuación se detallan las reglas de la misma:

	Integer	**Point**	**Float**
Integer	Integer	Integer	Float
Point	Integer	Point	Float
Float	Float	Float	Float

Tabla A.1: Tabla de conversiones de tipos

A.4. Elementos léxicos

Identificadores

Un identificador es una secuencia de caracteres formada por una o más letras y números y cuyo primer carácter debe ser una letra. Los identificadores sirven para nombrar clases de eventos, funciones, especificaciones y series temporales.

```
identifier = ( “a” ... “z” | “A ... “Z” ) { “a” ... “z” | “A ... “Z” | “0” ... “9” }
```

Listado A.1: Regla gramatical para identificadores

Palabras reservadas

Son las siguientes identificadores: *point, float, bool, event, eventset, pointset, timeseries, ts, eventspec, true, false, join, intersec, diff*

Literales de tipo punto

Un literal de tipo punto es una secuencia de uno o más dígitos.

```
point = “0” ... “9” {“0” ... “9”}
```

Listado A.2: Regla gramatical para literales de tipo punto

Literales de tipo coma flotante

Un literal de tipo coma flotante es una secuencia de uno o más dígitos denominada parte entera y separada mediante un punto de otra secuencia de uno o más dígitos denominada parte decimal.

---

float = “0” ... “9” { “0” ... “9” } “.” “0” ... “9” { “0” ... “9” }

Listado A.3: Regla gramatical para literales tipo coma flotante

Literales de tipo booleano

Un literal de tipo Booleano es la palabras reservadas *true* o *false*.

---

bool = “true” | “false”

Listado A.4: Regla gramatical para literales tipo booleano

Operadores aritméticos

Los operadores aritméticos son los siguientes: +, -, *, /, %

Operadores booleanos

Los operadores booleanos son los siguientes: &&, //, !

Operadores relacionales

Los operadores relacionales son los siguientes: ==, != <, <=, >=, >

Comentarios

Un comentario es una secuencia de caracteres que comienza con los caracteres // y terminal al final de la línea.

Declaraciones
Una declaración asocia un identificador a una especificación, una serie temporal, un conjunto de puntos, un conjunto de eventos o un tipo de evento.

Declaración de especificación
Una declaración de especificación asocia un identificador a una especificación e identifica el nombre del programa TESL.

```
eventspec = "eventspec" identifier ";"
```

Listado A.5: Regla gramatical para la declaración de especificación

Declaración de serie temporal
Una declaración de serie temporal asocia un identificador a una serie temporal.

```
timeseries = ("timeseries" | "ts") identifier ";"
```

Listado A.6: Regla gramatical para la declaración de serie temporal

Declaración de conjunto
Una declaración de conjunto asocia un identificador a un conjunto de puntos o eventos. Está compuesta por una fuente iterable de puntos o eventos denominada *source* y una cláusula de guarda opcional denominada *guard*. La cláusula de guarda permite filtrar los elementos que pertenecen al conjunto en en función de a alguna condición booleana. Semánticamente, el tipo de la fuente de elementos debe coincidir con el tipo de elementos del conjunto.

```
setdecl = ("pointset" | "eventset") identifier "{" source [guard] "}" ";"
source = ("point" | "event") identifier "in" expression
guard =  "such" "that" expression
```

Listado A.7: Regla gramatical para la declaración de conjunto

Declaración de clase de evento

Una declaración de clase de evento asocia un identificador a una clase de eventos. Está compuesta por una o más fuentes *sources* y una cláusula de guarda *guard*. Semánticamente, una clase de evento es declarada o bien mediante 3 puntos o bien mediante uno o varios eventos.

---

eventdelc = “event” identifier “{“ sources [guard] “}”

sources = sources {“,” source}

source = (“point” | “event”) identifier “in” expression

guard = “such” “that” expression

Listado A.8: Reglas gramaticales para la declaración de clase de evento

Expresiones

Una expresión es el cálculo de un valor mediante literales, operadores y llamadas a funciones. El tipo de una expresión es el tipo de este valor calculado.

---

expr = (“(“ expr “)”)

| boolean_expr

| arithmetic_expr

| relational_expr

| call_expr

| set_expr

| interval_expr

| literal_expr

| identifier_expr

Listado A.9: Reglas gramaticales para expresiones

Expresiones Booleanas

Una expresión Booleana es aquella en la cual se utilizan operadores Booleanos. Los operandos deben ser de tipo Booleano, detectándose un error TypeError en caso contrario. El tipo resultado es igualmente Booleano.

```
boolean_expr = expr ( "&&" | "||" ) expr
             | "!" expr
```

Listado A.10: Regla gramatical para expresiones booleanas

Expresiones aritméticas

Una expresión aritmética es aquella en la cual se utilizan operadores aritméticos. Los operandos deben ser de tipo Entero o Coma Flotante, detectándose un error TypeError en caso contrario. El tipo del resultado se calcula siguiendo la tabla de conversión de tipos.

```
arithmetic_exp = expr ( "+" | "-" | "*" | "\" | "%" ) expr
               | ("+" | "-") expr
```

Listado A.11: Regla gramatical para expresiones aritméticas

Expresiones relacionales

Una expresión relacional es aquella en la cual se utilizan operadores relacionales. Los operandos "==" y "!=" se aplican a tipos equivalentes y los demás a tipos numéricos, detectándose un error TypeError en caso contrario. El tipo del resultado es Booleano.

```
relational_expr = expr ("==" | "!=" "<=" | "<" | ">" | ">=" ) expr
```

Listado A.12: Regla gramatical para expresiones relacionales

Expresiones de conjuntos

Una expresión de conjuntos es aquella en la cual se usan operadores de conjuntos. Ambos operandos deben ser de tipo conjunto de eventos o conjunto de puntos, detectándose un error TypeError en caso contrario. El tipo del resultado coincide con el de los operandos.

```
set_expr = expr ( “intersec” | “diff” | “join” ) expr
```

Listado A.13: Regla gramatical para expresiones de conjunto

Expresiones de intervalos

Una expresión de intervalos es aquella en la cual se utilizan operadores de intervalos junto a tres operandos. El primero de ellos debe ser de tipo Evento, mientras que los últimos corresponden a los puntos de inicio y fin del intervalo y deben ser de tipo Punto, detectándose un error TypeError en caso contrario. El tipo del resultado es Booleano.

```
interval_expr = expr ('begins' | 'ends' | 'covers' | 'exists') '[' expr ',' expr ']'
```

Listado A.14: Regla gramatical para expresiones de intervalo

Expresiones de llamada a función

Una expresión de llamada a función es aquella en la cual se llama a una función aplicando unos argumentos. El tipo de la expresión que recibe la llamada debe ser tipo Función, detectándose un error TypeError si no es así. El número y tipo de los argumentos deben coincidir con los de la función, detectándose en caso contrario un error ArgumentError o TypeError respectivamente. El tipo resultado será el tipo devuelto por la función.

```
call_expr = expr “(“ [args] “)”
args = expr {“,” expr}
```

Listado A.15: Reglas gramaticales para expresiones de llamada a función

Expresiones literales

Una expresión literal es aquella que está compuesta por una constante. El tipo del resultado de la misma será el de la constante.

```
literal_expr = integer
             | boolean
             | float
```

Listado A.16: Reglas gramaticales para expresiones literales

Expresiones de identificador

Una expresión identificador es aquella compuesta por un identificador. El tipo del resultado será el asociado a ese identificador. En caso de usarse un identificador no declarado previamente se detectará un error NameError.

```
identifier_expr = identifier
```

Listado A.17: Regla gramatical para expresiones de identificador

A.5. Precedencia de operadores

La precedencia de los operadores binarios en el lenguaje sigue el siguiente orden:

Precedencia	Operador
5	* / %
4	+ -
3	!= == < <= > >=
2	&&
1	||

Tabla A.1: Tabla de precedencia de los operadores

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Jun 06 22:36:47 CEST 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)