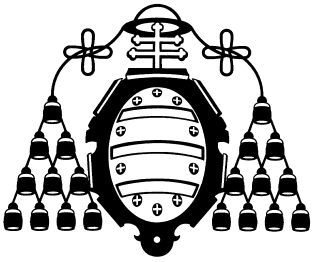
UNIVERSIDAD DE OVIEDO

Departamento de Informática



TESIS DOCTORAL

Desarrollo y evaluación de técnicas de construcción de Procesadores de Lenguaje para Máquinas Abstractas Orientadas a Objetos

Presentada por

María Cándida Luengo Díez

Para la obtención del Título de Doctor por la Universidad de Oviedo

Dirigida por el

Profesor Doctor D. Juan Manuel Cueva Lovelle

Oviedo, Febrero de 2002

Resumen

En este documento se realiza un estudio de las principales técnicas de construcción de procesadores de lenguajes y se proponen nuevas técnicas de desarrollo que permitan utilizar tecnologías orientadas a objetos.

La mayoría de los sistemas existentes, mezclan especificaciones y mantienen un procesamiento tradicional basado en técnicas procedimentales. Las carencias más importantes, atribuibles a los sistemas actuales de construcción de procesadores de lenguajes, sobre las que se ha realizado el estudio son: reusabilidad, modularidad, extensibilidad, mantenimiento y entornos visuales de desarrollo.

Una aproximación interesante para solucionar los problemas encontrados es la que se presenta en este trabajo. La idea se basa en construir un sistema que utilice técnicas orientadas a objetos, combinando marcos de aplicación (frameworks) y patrones.

Para comprobar la viabilidad del sistema diseñado, se implementa un prototipo y como aplicación práctica, se desarrollan todos los procesadores de lenguajes utilizados en la implementación del propio prototipo.

Palabras clave

Tecnologías orientadas a objetos, lenguajes de programación, procesadores de lenguajes, generadores de procesadores de lenguajes, máquinas abstractas, marcos de aplicación, patrones.

Abstract

In this document, we include a study of the main tecniques for the construction of languages processors and we purpose new techniques of development which allow make use of object oriented technologies.

In the majority of the current systems, the specifications are combined and they have a tradicional processing based on procedimental technique. The most important deficiendy associated with the current systems for the construction of languages processors over the study has been made are: reusability, modularity, extensibility, maintainability and graphics user interfaces.

An interesting approach to solve this problems is presented in this work. The idea is based on building an system which provide direct suport for object oriented techniques, working with frameworks and patterns.

In order to obtain a proof of the feasibility of the system designed, we have implemented a prototype and as a practical application, we have developed all languages processors used for the implementation of the own prototype.

Keywords

Object Oriented technologies, programming languages, languages processor, languages processor generators, abstract machines, frameworks, patterns.

Agradecimientos

Este trabajo no habría sido posible sin la contribución de muchas personas que de forma directa e indirecta ofrecieron su apoyo.

Agradezco especialmente, el apoyo, los consejos y las constantes dosis de ánimo recibidas a lo largo de estos años de mi director de tesis Juan Manuel Cueva.

Gracias a David Basanta por los valiosos intercambios de opinión durante el desarrollo e implementación del prototipo. A José Emilio Labra y Néstor García por sus acertadas sugerencias y su contribución en las revisiones del texto así como de los artículos que precedieron a esta tesis. También a los compañeros del grupo de investigación que en todo momento mostraron su solidaridad (especialmente a Belén Martínez) y a las personas que escribieron a raíz de las publicaciones (Stephen C. Johnson, Udo Hafermann, Bernd Kuehl, ...), mostrando su interés en el trabajo que estábamos realizando.

Por último, mi más sincero agradecimiento a mi familia por apoyarme y mostrar su confianza en los momentos más difíciles. Especialmente a las dos personas que más han notado mis ausencias y han hecho posible la finalización de esta tarea, *Néstor*, que siempre estuvo a mi lado, y *Óscar*, que con tan solo dos años me proporcionó la energía necesaria con sus altas dosis de cariño y alegría.

Tabla de Contenidos

CAPÍTULO 1

INTRODUCCIÓN

1.1 Objetivos

El objetivo principal de esta tesis es la descripción de un sistema de soporte en la construcción de procesadores de lenguajes, que facilite el desarrollo e implementación de los lenguajes de programación, y la reutilización de diferentes tareas relacionadas con la programación de una forma rápida y sencilla. En concreto, en esta tesis se describen las características que debe tener este sistema, así como la estructura de una arquitectura software que lo soporta.

Cada vez está tomando más fuerza la utilización de procesadores de lenguajes, de manera extensiva, en numerosas disciplinas, y el uso de sistemas que faciliten su construcción resulta muy ventajoso. A partir de una o varias especificaciones, que como mínimo incluirán la gramática del lenguaje que se va a procesar, es posible obtener analizadores léxicos y sintácticos eficientes, e incluso todas las fases de un compilador sin demasiado esfuerzo.

Los sistemas existentes en estos momentos mantienen un procesamiento tradicional y ofrecen pocas variaciones respecto a las técnicas utilizadas inicialmente por los sistemas clásicos. Las soluciones aportadas no son adecuadas para proporcionar características relacionadas con la *reusabilidad*, *modularidad*, *extensibilidad y mantenimiento*. Por otro lado, mantienen un *ciclo de desarrollo largo*, carecen de *entornos visuales de desarrollo* y el *tratamiento de errores* es poco eficaz.

En esta tesis se realiza un estudio comparativo de las principales técnicas utilizadas para la generación de procesadores de lenguajes, en base a las características anteriores, y se proponen nuevas técnicas. El sistema propuesto se basa en la utilización de técnicas orientadas a objetos integrando marcos de aplicación (frameworks) [JF88, FS97] y patrones [GHJ+95] como métodos de diseño.

Se consideran objetivos fundamentales: establecer un modelo de objetos uniforme y homogéneo, para estructurar la funcionalidad del sistema por medio de conjuntos de objetos, y además, conseguir un diseño que integre la mayor parte de los aspectos relacionados con la usabilidad. Como objetivo final, se busca la

valoración cualitativa del modelo diseñado y las principales técnicas utilizadas para la generación de procesadores de lenguajes.

1.2 Fases de desarrollo

En este apartado se describen las fases de trabajo en las que se divide el desarrollo de la tesis. En él no se hace mención al contenido desarrollado en cada fase, sino a los aspectos funcionales de cada una.

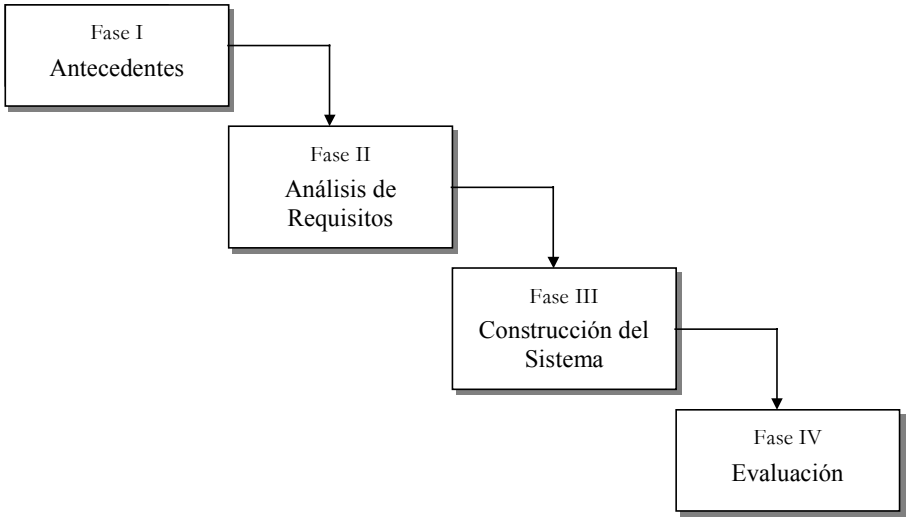


Figura 1.1: Fases de desarrollo

1. En la primera fase, se trata de analizar qué ofrecen los sistemas actuales que permiten generar procesadores de lenguajes, tanto en entornos comerciales como de investigación. En este estudio se conseguirá un conocimiento profundo de la forma de operar y las características que ofrecen. Asimismo, una visión conjunta permitirá realizar comparativas que lleven a la obtención explícita de críticas de los sistemas revisados y de los fallos existentes, se podrán extraer conclusiones que resulten de utilidad para los nuevos planteamientos.

2. En la segunda fase, se realiza un análisis de los requisitos deseables para el diseño del nuevo sistema. Los aspectos generales del sistema y los aspectos relacionados con la arquitectura del sistema servirán de base para la elección de los requisitos de partida.

3. Como consecuencia de los requisitos impuestos y los conocimientos adquiridos en la primera fase se elabora el diseño del sistema. Esta fase se concibe como resultado del proceso de crítica, reflexión y selección de posibilidades iniciado después del desarrollo de la primera fase. A ello hay que añadir un proceso de síntesis y consecución de un nuevo sistema que se ajusta mejor a los requisitos de partida y que tiene considerables ventajas sobre los existentes. Finalmente, dentro de esta fase, se formalizará este

sistema, analizando sistemáticamente todos los aspectos que intervienen en él.

4. Una vez formalizado el sistema, se procede a la descripción de las características que corroboran la bondad de éste, y su mayor adecuación respecto a otros ya existentes. La implementación de un prototipo es fundamental y sirve para apoyar esta demostración.

1.3 Organización de los contenidos

La estructura de esta tesis está organizada según las fases de desarrollo descritas anteriormente.

El capítulo 2, *Lenguajes de programación*, está dedicado a ofrecer una introducción de los conceptos básicos de los lenguajes. En él se definen los aspectos sintácticos y semánticos, se analizan algunos principios de diseño y los principales paradigmas de programación, destacando la creciente importancia de los lenguajes de dominio específico. En definitiva, se trata de delimitar el ámbito de trabajo y determinar las relación existente entre lenguajes y procesadores.

El capítulo 3, *Procesadores de lenguajes de programación*, se dedica a analizar fundamentalmente a los sistemas generadores de procesadores de lenguajes. Se parte de un conjunto de definiciones y se describen brevemente todos los aspectos generales que deben ser considerados, como su estructura, clasificación y características.

El capítulo 4, *Panorama de diversas técnicas de construcción de procesadores de lenguajes,* realiza una visión general del estado del arte de los sistemas utilizados. Para ello, se han seleccionado un conjunto de sistemas que son representativos de las diferentes técnicas de construcción de procesadores de lenguajes existentes. De cada uno de estos sistemas, se lleva a cabo una pequeña descripción de sus características generales y una valoración de aspectos a favor y en contra.

El capítulo 5, *Tecnologías orientadas a objetos*, analiza los problemas derivados de un procesamiento tradicional, y justifica la utilización de técnicas orientadas a objetos para solucionar problemas comunes detectados en los sistemas revisados.

En el capítulo 6, *Máquinas abstractas orientadas a objetos*, se justifican las ventajas del uso de las máquinas abstractas orientadas a objetos en el desarrollo de procesadores.

El capítulo 7, *Requisitos del sistema O2C2*, realiza una descripción de aquellos requisitos que se consideran importantes para la consecución del sistema propuesto. Los requisitos planteados en este capítulo afectan principalmente al diseño y a la construcción de dicho sistema.

El capítulo 8, *Diseño del sistema O2C2*, desarrolla una arquitectura para el sistema. En él se describen los módulos básicos que proporcionan los servicios del sistema y las principales características ofrecidas por esta arquitectura. La aplicación de técnicas orientadas a objetos aportan un conjunto de propiedades que van a servir de base para proporcionar la funcionalidad básica del sistema. El resultado obtenido es la idea de un mecanismo nuevo, que no ha sido usado con anterioridad y que aporta ventajas adicionales a los sistemas existentes.

El capítulo 9, *Diseño de un prototipo para el sistema O2C2*, describe en términos generales el diseño de un prototipo para el sistema atendiendo a las características especificadas en los dos capítulos anteriores.

El capítulo 10, *Implementación del prototipo*, describe los pasos seguidos para la obtención de una versión inicial del prototipo, y las ampliaciones y mejoras realizadas sobre la versión inicial. Las mejoras, sirvieron para solventar los problemas detectados en la versión inicial, y las ampliaciones, para dotar al prototipo de la funcionalidad descrita en el diseño del sistema O2C2. Este desarrollo inicial será tomado como base para la construcción del prototipo final mediante la aplicación de la técnica conocida como bootstraping. Para comprobar la bondad del sistema diseñado, se usó el propio prototipo para construir cada uno de los analizadores utilizados en la implementación, probándose de esta forma, su capacidad inmediata para generar procesadores de lenguajes de forma rápida y sencilla.

En el capítulo 11, *Ventajas del sistema diseñado y aplicaciones*, se justifican las principales ventajas que aporta el sistema diseñado, que corroboran la utilidad de éste, así como sus posibles aplicaciones.

En el capítulo 12, *Conclusiones*, se valoran y comparan las diferentes técnicas de construcción de procesadores de lenguajes, incluyendo dos trabajos relacionados con temas tratados en este documento. También se resumen las principales conclusiones obtenidas como consecuencia de la elaboración de este trabajo, y se perfilan algunas líneas de investigación futuras que marquen el camino a seguir en la investigación empezada con esta tesis.

El anexo A incluye una descripción del manual de usuario del entorno visual de desarrollo del prototipo. En el anexo B se presenta una tabla de conversión de términos en inglés y por último, aparece la bibliografía utilizada en este trabajo.

CAPÍTULO 2

LENGUAJES DE PROGRAMACIÓN

En este capítulo se revisarán algunos conceptos de lenguajes de programación y se analizarán los principales paradigmas de programación, destacando los aspectos sintácticos y semánticos de los lenguajes en general.

2.1 Introducción

Los lenguajes de programación constituyen una herramienta fundamental en la informática. Basta considerar que cualquier producto software es desarrollado utilizando uno o varios lenguajes de programación. De hecho, la elección de lenguajes de programación adecuados puede ser la clave del éxito de muchos proyectos informáticos.

Existe una enorme variedad de lenguajes de programación[1]. La descripción de estos lenguajes requiere identificar el formato de los programas que se pueden escribir (**sintaxis**) así como su comportamiento (**semántica**). Mientras que para la descripción sintáctica, la notación BNF se utiliza de forma prácticamente universal, para la descripción semántica, no existe un formalismo comúnmente aceptado [Wat96]. En numerosas ocasiones, la semántica es especificada a través del formalismo que impone el propio lenguaje de programación.

Se han propuesto diversos formalismos para la descripción semántica de lenguajes de programación, como la semántica operacional estructurada, natural, denotacional, algebraica, axiomática, de acción, etc. Cada uno de ellos tiene sus ventajas e inconvenientes sin que se haya encontrado todavía un formalismo superior a los demás. En [Lab01] se describe una nueva técnica que combina la semántica monádica modular con los nuevos desarrollos en el campo de la programación genérica.

La especificación o descripción del lenguaje debe ser aplicada de forma clara en la *sintaxis* del lenguaje (se debe especificar qué forma tiene un programa exactamente), en la *semántica estática* del lenguaje (por ejemplo se debe conocer

[1] En [Ulo01] se incluye una lista de más de 2000 lenguajes de programación

qué restricción debería ser aplicada durante el uso de entidades de diferentes tipos) y en la *semántica dinámica* de los programas que satisfacen las reglas sintácticas (debe ser capaz de predecir el resultado de algunos programas cuando son ejecutados).

En esta tesis se hace un estudio de las diversas técnicas de construcción de procesadores de lenguajes y se desarrollan nuevas técnicas orientadas a objetos, centrándose en la descripción sintáctica expresada en notación EBNF [Iso96], y en la descripción semántica especificada mediante el propio lenguaje de implementación.

2.2 Lenguajes como instrumentos de comunicación

Un lenguaje natural es un instrumento de comunicación utilizado de forma común entre personas. Para que exista comunicación, debe existir una comprensión mutua de cierto conjunto de símbolos y reglas del lenguaje.

Los lenguajes de programación tienen como objetivo la construcción de programas, normalmente escritos por personas. Estos programas se ejecutarán por un ordenador que realizará las tareas descritas. El programa debe ser comprendido tanto por personas como por ordenadores. La utilización de un lenguaje de programación requiere, por tanto, una comprensión mutua por parte de personas y máquinas. Este objetivo es difícil de alcanzar debido a la naturaleza diferente de ambos.

En un lenguaje natural, el significado de los símbolos se establece por la costumbre y se aprende mediante la experiencia. Sin embargo, los lenguajes de programación se definen habitualmente por una autoridad, que puede ser el diseñador individual del lenguaje o un determinado comité.

Para que el ordenador pueda comprender un lenguaje humano, es necesario diseñar métodos que traduzcan tanto la estructura de las frases como su significado a código máquina. Los diseñadores de lenguajes de programación construyen lenguajes que saben cómo traducir o que creen que serán capaces de traducir. Si los ordenadores fuesen la única audiencia de los programas, estos se escribirían directamente en código máquina o en lenguajes mucho más mecánicos. Sin embargo, el programador debe ser capaz de leer y comprender el programa que está construyendo y las personas no son capaces de procesar información con el mismo nivel de detalle que las máquinas.

Los lenguajes de programación son, por tanto, una solución de compromiso entre las necesidades del emisor (programador – persona) y del receptor (ordenador – máquina).

Las declaraciones, tipos, nombres simbólicos, etc. son concesiones de los diseñadores de lenguajes para que los humanos podamos entender mejor lo que se ha escrito en un programa. Por otro lado, la utilización de un vocabulario limitado y de unas reglas estrictas son concesiones para facilitar el proceso de traducción.

El estudio lingüístico de un lenguaje suele descomponerse en sintaxis, semántica y pragmática. La sintaxis estudia las reglas de formación de frases, la semántica su significado y la pragmática la relación entre el lenguaje y los usuarios del lenguaje.

2.2.1 Aspectos sintácticos

2.2.1.1 Sintaxis concreta

Convencionalmente, la **sintaxis concreta** se separa en análisis léxico y análisis sintáctico propiamente dicho o *parsing*.

El objetivo del análisis léxico es agrupar los caracteres del texto de un programa en un conjunto de símbolos legales o *tokens*. Con el análisis sintáctico se intenta agrupar estos símbolos en frases, construyendo un árbol sintáctico o árbol de derivación que tiene a los símbolos como hojas. Los principales tipos de símbolos reconocidos en el análisis léxico suelen ser:

- *Delimitadores*: marcas de puntuación, signos matemáticos, etc.
- *Palabras*: secuencias alfanuméricas, las cuales pueden subdividirse en un conjunto de palabras reservadas como *begin* o *end*, y un conjunto de identificadores utilizados para dar nombre a entidades del programa.
- *Literales numéricos*: secuencias de dígitos, puntos decimales y signos indicando bases y/o exponentes.
- *Literales que representan caracteres o cadenas de caracteres*: suelen utilizarse comillas simples o dobles.
- *Comentarios*: secuencias arbitrarias de caracteres, delimitados por secuencias especiales, que se utilizan para ayudar a la comprensión del programa por parte de los programadores, pero que son ignorados por el ordenador.
- *Separadores*: espacios, caracteres fin de línea, etc.

Los comentarios y los separadores aparecen generalmente de forma libre entre el resto de símbolos léxicos. Resulta tedioso describir dicha libertad como parte de la estructura de las frases, por lo que en general, una vez reconocidos por el analizador léxico, son eliminados de la secuencia de símbolos que se obtiene del análisis léxico.

El análisis sintáctico obtiene a partir del conjunto de símbolos anteriores una estructura de las frases que forman el programa. Habitualmente, se utilizan varias categorías como por ejemplo:

- *Declaraciones*: Permiten introducir identificadores que dan nombre a entidades del programa.

- *Expresiones*: Se asemejan a términos matemáticos
- *Enunciados*: simples o compuestos, que pueden contener a su vez declaraciones o expresiones.
- *Programas*: combinaciones adecuadas de declaraciones, expresiones y enunciados.

La clasificación anterior puede realizarse mediante lo que se denomina *análisis libre de contexto*, ya que el sistema puede reconocer las diferentes categorías independientemente del contexto en el que aparezcan.

Para describir la sintaxis libre de contexto suele utilizarse la notación BNF (Backus-Naur Form), introducida para la descripción del lenguaje Algol-60 [Nau60].

Ejemplo 2.1 *La siguiente gramática describe la sintaxis concreta de expresiones aritméticas simples en notación BNF. Para resolver precedencia y asociatividad de los operadores se utilizan 3 categorías sintácticas*.

```
<exp>    := <term> '+' <exp>
          | <term> '-' <exp>
          | <term>
<term>   := <factor> '*' <term>
          | <factor> '/' <term>
          | <factor>
<factor> := <number>
          | '(' <exp> ')'
```

A la expresión 3 + 4 * 5 le correspondería el siguiente árbol sintáctico concreto de la *figura 2.1*

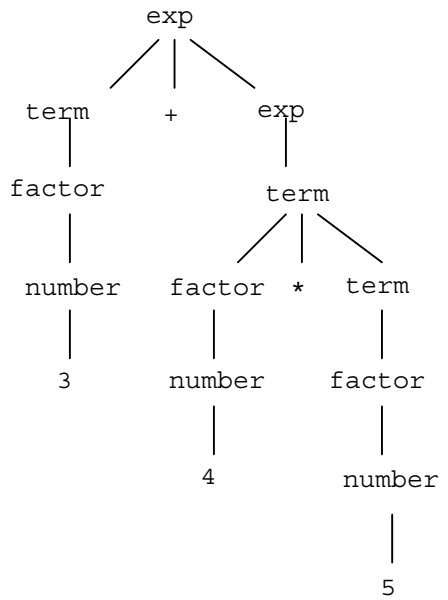


Figura 2.1 Árbol sintáctico concreto de la expresión 3 + 4 * 5

2.2.1.2 Sintaxis abstracta

La **sintaxis abstracta** [McC63] proporciona una interfaz adecuada entre la sintaxis concreta y la semántica. Habitualmente se obtiene de forma simple, ignorando aquellos detalles del árbol sintáctico que no tienen relevancia semántica, dejando árboles sintácticos que representan únicamente la estructura composicional esencial de los programas.

Ejemplo 2.2 *La siguiente gramática define la sintaxis abstracta del mismo lenguaje formado por expresiones aritméticas simples del ejemplo 2.1*

```
<exp> := <exp> '+' <exp>
       | <exp> '-' <exp>
       | <exp> '*' <exp>
       | <exp> '/' <exp>
       | <number>
```

A partir de dicha gramática, el árbol de la sintaxis abstracta de la expresión 3 + 4 * 5 se representa en la *figura 2.2*

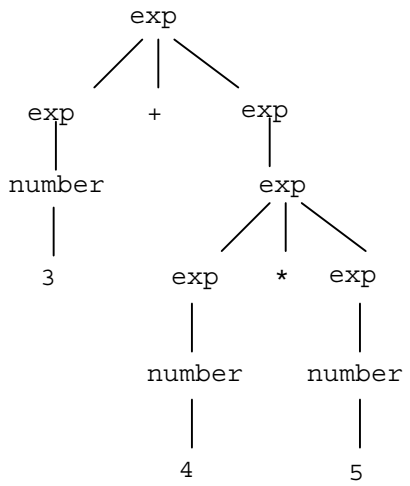


Figura 2.2 Árbol sintáctico abstracto de la expresión 3 + 4 * 5

2.2.1.3 Sintaxis sensible al contexto

La **sintaxis sensible al contexto** se enfrenta a aquellos aspectos que no pueden ser tratados por la sintaxis libre de contexto, como *la declaración de identificadores antes de su utilización, las expresiones con el tipo adecuado*, etc.

Los sistemas de análisis estático de tipos han alcanzado enorme popularidad ya que el programador puede especificar restricciones sobre el dominio de valores que toman las expresiones en el lenguaje. De esa forma, el sistema puede comprobar que las declaraciones coinciden con la implementación realizada antes de la ejecución, eliminando posibles errores. Además, se evita la realización de

comprobaciones de tipos durante la ejecución del programa, aumentando la eficiencia [Car97].

Al realizar el análisis de lenguajes con sintaxis sensible al contexto, es necesario encajar partes de un programa distantes entre sí. Existen varias alternativas:

Las gramáticas atribuidas introducidas por D. Knuth en 1968 [Knu90] toman una gramática libre de contexto y le añaden un conjunto de atributos y una serie de ecuaciones. De esta forma es posible enviar información sobre los valores de los atributos entre los diferentes nodos del árbol sintáctico, facilitando el análisis contextual del lenguaje. Sin embargo, es importante destacar que las gramáticas atribuidas tradicionales tienen algunos problemas de modularidad.

Otra posibilidad es considerar la sintaxis sensible al contexto como un tipo especial de semántica, conocida como *semántica estática*. Se denomina así porque depende únicamente de la estructura del programa, no de la entrada del usuario. El comportamiento de un programa que depende de la entrada se conoce como *semántica dinámica*[1].

La forma más sencilla de considerar la semántica estática es como una restricción sobre los programas: si un programa no cumple la restricción, entonces se considera ilegal. Tal semántica podría considerarse como una función que asignase valores booleanos a los programas. En la práctica, un tratamiento sistemático suele ser más complicado, ya que deben tenerse en cuenta los mensajes de error sobre los motivos del incumplimiento.

En esta tesis se tiene en cuenta únicamente las descripciones de la semántica estática para el tratamiento de los lenguajes de programación.

2.2.2 Aspectos semánticos

Dado un programa en un lenguaje determinado, ¿cuál es la naturaleza de su semántica?

En primer lugar, deben apartarse los efectos que el programa pueda tener sobre un lector humano, evocando, por ejemplo, sentimientos de admiración, o quizás, y más a menudo, de disgusto. A diferencia de la filología, la lingüística de la programación, no trabaja con cualidades subjetivas. El significado de un programa depende solamente del *comportamiento objetivo* que el programa cause cuando es ejecutado por un ordenador.

Los ordenadores son sistemas complejos y, cuando se ejecuta un programa, pueden observarse efectos muy diversos: movimientos de cabezas del disco, flujos de corrientes en los circuitos, caracteres que aparecen en una pantalla o en una impresora, etc. Con el fin de considerar programas que controlan de forma

[1] En algunos textos tradicionales que sólo se ocupan del desarrollo de procesadores de lenguaje, se denomina semántica a la semántica estática, lo cual puede llevar a cierta confusión.

específica tales comportamientos físicos, es necesario afrontar estos fenómenos en su semántica.

Sin embargo, se considerarán programas en lenguajes de alto nivel, los cuales no realizan un control directo sobre detalles de comportamiento físico del ordenador. La semántica apropiada de estos programas es independiente de la implementación, que consiste únicamente en aquellas características que son comunes a todas las implementaciones. Esto incluye habitualmente propiedades sobre la terminación de un programa, pero ignora consideraciones electrónicas.

Puede utilizarse la siguiente definición (adaptada de [Mos92]).

Definición 2.1 (Semántica) *La semántica de un programa es una entidad abstracta que modeliza el comportamiento de forma independiente de la implementación particular*.

Definición 2.2 (Semántica composicional) *Una descripción semántica es composicional si el significado de una frase compuesta viene determinado únicamente por el significado de sus componentes y es independiente de otras características*.

2.2.3 Aspectos pragmáticos

La pragmática se refiere a la relación entre el lenguaje y sus usuarios. En el caso de lenguajes de programación la pragmática estudiará las técnicas y tecnologías empleadas en la construcción de programas.

A la hora de desarrollar grandes proyectos informáticos, aspectos pragmáticos como la organización y planificación, el análisis de requerimientos, la gestión de recursos humanos, el trabajo en equipo, etc. tienen una importancia incuestionable.

No obstante, se considera fundamental el conocimiento y comprensión de la herramienta básica de trabajo, en este caso, el lenguaje de programación. El comportamiento del software está determinado, en último momento, por el comportamiento especificado del lenguaje en el que se construye.

En la presente tesis se prescinde de los detalles pragmáticos de la construcción de software y se tratan exclusivamente los aspectos de descripción sintáctica y semántica de lenguajes de programación.

2.3 Diseño de lenguajes

Al diseñar lenguajes de programación, a menudo es necesario tomar decisiones sobre las características que se incluyen de forma permanente, las características que no se incluyen pero existen mecanismos que facilitan su inclusión y las que no se permiten. Estas decisiones pueden afectar al diseño final del lenguaje y posiblemente entrar en conflicto con otros aspectos del lenguaje.

A continuación se resumen algunos principios de diseño de lenguajes de programación recogidos de diversas fuentes [Ste98, FG93, Kam90, McL87, PZ96].

- *Concisión notacional*. El lenguaje debe permitir describir algoritmos con el nivel de detalle adecuado.

- *Integridad conceptual*. Se proporcionan un conjunto de conceptos simple claro y unificado. Lo ideal es disponer de un conjunto de conceptos diferentes mínimo, con una reglas simples y regulares para su combinación.

- *Ortogonalidad*. Dos características de un lenguaje son ortogonales si pueden ser comprendidas y combinadas de forma independiente.

- *Generalidad*. Todas las características de un lenguaje son generadas a partir de conceptos básicos.

- *Abstracción*. Evitar que algo deba ser enunciado más de una vez. Permitiendo la factorización de patrones repetitivos.

- *Extensiblidad*. El lenguaje debe admitir la creación de nuevas características no previstas en el momento de su creación.

- *Seguridad*. Deben existir medios adecuados para comprobar cuándo los programas no cumplen con la definición del lenguaje o con la propia estructura pretendida por el programador. Por ejemplo, si el lenguaje admite declaraciones de tipos, debe ser posible comprobar que el tipo declarado de una función coincide con el tipo de su implementación.

- *Automatización*. El lenguaje debe admitir la automatización de tareas mecánicas, tediosas o susceptibles de producir errores.

- *Portabilidad*. La definición del lenguaje debe facilitar que sus programas funcionen en diferentes máquinas y clases de máquinas.

- *Eficiencia*. Es conveniente estudiar la eficiencia, tanto de los programas al ejecutarse, como de las herramientas de procesamiento del lenguaje.

- *Entorno*. Aunque el entorno no forma parte del lenguaje, muchos lenguajes débiles técnicamente son ampliamente utilizados debido a que disponen de un entorno de desarrollo potente. De la misma forma, la disposición de documentación, ejemplos de programas e incluso programadores, pueden ser factores clave del desarrollo de un lenguaje de programación.

2.3.1 Familias de lenguajes

El concepto de paradigma fue utilizado por T. S. Kuhn [Kuh93] para justificar las revoluciones científicas. Un paradigma engloba un conjunto de normas,

prácticas y creencias de una comunidad científica en un momento dado. Las personas de esa comunidad aceptan y comparten dichas creencias sin discusión, hasta que aparece un nuevo paradigma que rebate alguna característica fundamental.

En informática, también es posible observar varias comunidades de ese tipo, cada una hablando su propio lenguaje y utilizando sus propios paradigmas. De hecho los lenguajes de programación suelen fomentar el uso de ciertos paradigmas y disuadir el uso de otros.

A continuación se resumen los principales paradigmas de programación, aunque existen lenguajes de programación híbridos:

- La *programación imperativa* presta especial atención a la secuencia de órdenes que el programador debe comunicar para resolver un problema. *Cobol, Fortran, Pascal, C*, etc. son lenguajes comúnmente aceptados dentro de este paradigma. En oposición al paradigma imperativo, surgen los paradigmas declarativos[1].

- La *programación funcional* toma como elemento central las funciones que intervienen en el problema a resolver. Cuando se permiten exclusivamente funciones matemáticas sin efectos laterales, los lenguajes se denominan puramente funcionales. Una característica común de este paradigma es la utilización de funciones de orden superior, es decir, funciones que pueden tener como argumentos otras funciones y devolver funciones como resultado. *Lisp*, *Scheme*, *ML*, *Haskell* son ejemplos de lenguajes funcionales.

- La *programación lógica* se centra en la descripción de las relaciones que intervienen en el problema. El lenguaje más conocido sería el lenguaje *Prolog*, aunque existen otros lenguajes como *Mercury*, *Goedel*, *λProlog*, etc.

- En la *Programación Orientada a Objetos* se resuelve el problema definiendo los objetos que intervienen y el envío de mensajes entre ellos. Los lenguajes *Simula*, *Smalltalk*, C++, *Java* etc. se consideran lenguajes Orientados a Objetos.

Existen otros paradigmas, como la programación dirigida por eventos, la programación visual, etc.

2.3.2 Lenguajes de Dominio Específico

En todas las ramas de la ingeniería, aparecen técnicas genéricas junto con técnicas específicas. La aplicación de técnicas genéricas proporciona soluciones generales para muchos problemas, aunque tales soluciones pueden no ser óptimas.

[1] El paradigma declarativo (programación funcional y programación lógica).

Las técnicas específicas suelen estar optimizadas y aportan una solución mejor para un conjunto reducido de problemas.

En los lenguajes de programación también se produce esta dicotomía, lenguajes de dominio específico respecto a lenguajes de propósito general.

Recientemente, hay un interés creciente en el estudio de las técnicas de implementación de lenguajes de dominio específico. Una posible definición, adaptada de [DKV00], podría ser.

Definición 2.3 (Lenguaje de dominio específico) *Un Lenguaje de Dominio Específico es un lenguaje de programación o un lenguaje de especificación ejecutable que ofrece potencia expresiva enfocada y restringida a un dominio de problema concreto*.

Entre las principales características de estos lenguajes están:

- Estos lenguajes suelen ser pequeños, ofreciendo un conjunto limitado de notaciones y abstracciones. En ocasiones, pueden contener un completo sublenguaje, proporcionando además de las capacidades generales, las del dominio del problema concreto. Esta situación aparece cuando el lenguaje es empotrado en un lenguaje general.

- Suelen ser lenguajes declarativos, pudiendo considerarse lenguajes de especificación, además de lenguajes de programación. Muchos contienen un compilador que genera aplicaciones a partir de los programas, en cuyo caso el compilador se denomina generador de aplicaciones.

- Un objetivo común de muchos de estos lenguajes es la programación realizada por el usuario final. Estos usuarios no suelen tener grandes conocimientos informáticos pero pueden realizar tareas de programación en dominios concretos con un vocabulario cercano a su especialización.

- La utilización de Lenguajes de Dominio Específico ha supuesto un avance en gran variedad de dominios.

2.3.3 Desarrollo de Lenguajes de Dominio Específico

Existen varias técnicas para desarrollar lenguajes de dominio específico.

- La primera posibilidad es crear un *lenguaje específico independiente* y procesarlo como cualquier lenguaje de programación ordinario. Esta opción requiere el diseño completo del lenguaje y la implementación de un intérprete o un compilador.

- Otra posibilidad es *empotrar* el lenguaje específico en un lenguaje de propósito general. En [Kam98] se resumen varios lenguajes empotrados de dominio específico.

- Mediante *Preprocesamiento* o proceso de macros se empotra un lenguaje de propósito específico en otro lenguaje incluyendo una fase intermedia que convierte el lenguaje específico en el lenguaje anfitrión. Por ejemplo, el preprocesador de C++ puede utilizarse para crear un completo lenguaje específico para tareas concretas.

- *Intérprete extensible*. Consiste en añadir elementos que permitan modificar el intérprete para que analice lenguajes diferentes.

2.4 Evolución de los lenguajes de programación

Las bases teóricas de los lenguajes de programación se emparejan con las de los lenguajes formales. Los cuales surgen a partir de los desarrollos de la lógica matemática de finales del siglo XIX.

Aristóteles (384/3–322a.C) plantea el primer sistema lógico riguroso, con una estructura de tipo axiomático. Durante mucho tiempo, la lógica se desarrolla en el campo filosófico, utilizando el lenguaje natural como instrumento de comunicación.

G. W. Leibniz (1646–1716) propone dar un rigor matemático a la lógica que permita resolver las argumentaciones mediante procedimientos de cálculo, inspirándose en las operaciones aritméticas, e iniciando la corriente de lógica automatizada.

Durante la segunda mitad del s. XIX se produce un enorme interés en el estudio de la lógica, destacando los trabajos de G. Frege que dan lugar a la lógica de predicados y el programa logicista que pretende demostrar que los teoremas y principios matemáticos pueden obtenerse por cálculos deductivos a partir de unos axiomas, constituyendo el primer sistema axiomático de lógica de primer orden. Martin Davis lo considera el principal precedente clásico de la prueba automática de teoremas [Dav00]. El teorema de Herbrand resultó una pieza fundamental para la prueba automática de teoremas, base de la programación lógica.

En 1910, Russell y Whitehead publican los Principia Mathematica, en los que recopilan los conocimientos de lógica matemática de la época. En su obra se hace mención a la paradoja de las clases cuya solución dará lugar a la teoría de tipos, base de los sistemas de tipos de los lenguajes de programación actuales.

A principios de los años 20, Hilbert desarrolla la teoría de la prueba con el objetivo de axiomatizar los conceptos matemáticos básicos, y poder demostrar mediante métodos constructivos, todos los teoremas matemáticos. En sus desarrollos aparecen las definiciones de lenguajes formales y de métodos constructivos, que deben estar formados por una secuencia finita de pasos. Aunque el teorema de incompletud de Gödel da al traste con las expectativas de Hilbert, durante la primera mitad de los años 30 se desarrollaron métodos constructivos como el cálculo lambda de Church y las máquinas de Turing, que formarán la base teórica de los lenguajes de programación actuales. En esa época,

A. Tarski desarrolla la teoría de modelos, que fundamentará el estudio semántico de lenguajes formales, y por tanto, de lenguajes de programación.

Con la aparición de los primeros ordenadores, surge la necesidad de utilizar notaciones que faciliten la programación. El lenguaje ensamblador consiste en una serie de macros mnemotécnicas que se corresponden de forma más o menos directa con las instrucciones de la máquina. En 1954, J. Backus lidera un equipo para implementar el lenguaje Fortran, que ofrece construcciones de control de alto nivel y se impondrá como lenguaje de cálculo científico.

En 1959, J. McCarthy desarrolla el lenguaje Lisp, que incluye facilidades para tratamiento de listas y la utilización de funciones de orden superior. El lenguaje Lisp se convertirá en uno de los lenguajes más utilizados en aplicaciones de inteligencia artificial, considerándose el primer lenguaje funcional.

A finales de los años 50, existen numerosos lenguajes de programación para arquitecturas específicas e incompatibles entre sí. Se desarrolla un comité internacional para el desarrollo de un lenguaje algorítmico universal dirigido por P. Naur. El lenguaje desarrollado se denominará Algol y contiene las principales características conocidas en la época. El lenguaje tendrá cierto éxito en ambientes académicos como instrumento de comunicación de algoritmos, pero su complejidad impide la construcción de implementaciones y de aplicaciones prácticas.

En la descripción sintáctica del lenguaje Algol, P. Naur adopta una notación previamente utilizada por J. Backus, que posteriormente se denominará notación BNF (Backus-Naur Form). En la misma época, N. Chomsky propone las gramáticas libres de contexto para la descripción del lenguaje natural. El descubrimiento de la equivalencia descriptiva entre ambos formalismos favorece la posterior evolución de la lingüística computacional.

Algol marcó un hito en el diseño de lenguajes y tuvo una enorme influencia en los lenguajes posteriores. Esta influencia puede resumirse en tres líneas.

- Por un lado, N. Wirth desarrolla el lenguaje Pascal (1971) con propósitos educativos. Posteriormente, desarrollará los lenguajes Modula-2 (1983) y Oberon (1988).

- Por otro lado, C. Strachey, pionero en la creación de la semántica denotacional de lenguajes, diseña el lenguaje CPL (Combined Programming Language) en 1966. A partir de dicho lenguaje, Richards diseña BCPL o Basic CPL (1966) y posteriormente, D. Ritchie diseña el lenguaje C (1972) como lenguaje de implantación de programas asociados al sistema operativo Unix. Este lenguaje tendrá una enorme popularidad por la combinación entre características de alto nivel con la libertad de acceso a elementos de bajo nivel.

- Finalmente, K. Nygaard y O. J. Dahl diseñan el lenguaje Simula para el desarrollo de aplicaciones de simulación. La principal novedad del lenguaje es la combinación de datos y procedimientos en una entidad,

dando lugar al concepto de objeto y clase. A partir de dicho lenguaje, A. C. Kay diseña Smalltalk como un sistema uniforme orientado a objetos.

El lenguaje Simula 6, influyó en el diseño del Prolog. En 1972 A. Colmerauer y R. Kowalski desarrollan el primer intérprete de Prolog basándose en los procedimientos de prueba automática de teoremas, con el propósito de desarrollar sistemas de tratamiento de lenguaje natural. Este lenguaje dará lugar al paradigma de la programación lógica.

En 1974, se produce un nuevo intento de desarrollo de lenguaje universal. El Departamento de Defensa de Estados Unidos realiza un concurso para el diseño de un nuevo lenguaje para sistemas empotrados que se denominará a Ada. De la misma forma que en el caso de Algol, el lenguaje es excesivamente complejo aglutinando muchos de los conceptos de la ingeniería del software del momento.

En el campo de la programación funcional, la popularidad del lenguaje Lisp para aplicaciones de Inteligencia Artificial favoreció el crecimiento incontrolado del lenguaje incorporando características imperativas. Como reacción, en 1975, G. Steele Jr. y Sussman diseñan Scheme, un lenguaje tipo Lisp más sencillo.

El sistema de demostración de teoremas LCF incluía el metalenguaje funcional ML para la programación de estrategias. ML contenía un sistema de chequeo e inferencia estática de tipos cuya utilidad le permitió independizarse y comenzar a ser utilizado como lenguaje de propósito general.

Los programas ML pueden ser compilados consiguiendo competir con programas imperativos en eficiencia. No obstante, el lenguaje ML no es puramente funcional.

Por otro lado, en 1976 D. Turner desarrolla el lenguaje SASL incluyendo evaluación perezosa. Este lenguaje dará lugar al lenguaje comercial Miranda que alcanzaría cierta popularidad en ambientes académicos.

En 1989 se crea un comité internacional con el objetivo de diseñar un lenguaje puramente funcional que aglutine las características existentes. A este lenguaje se denomina Haskell.

La programación Orientada a Objetos alcanza gran popularidad con los lenguajes C++ y Java, y posee entre otras ventajas, la de simplificar el mantenimiento de las aplicaciones software. El interés creciente en la construcción de procesadores de lenguajes utilizando el lenguaje Java está motivado por el uso de una plataforma independiente, su robustez desde el punto de vista de la ingeniería del software, y su popularidad frente a otros lenguajes como C++.

CAPÍTULO 3

PROCESADORES DE LENGUAJES DE PROGRAMACIÓN

En este capítulo se describen las características comunes de los procesadores de lenguajes de programación, así como algunos de los aspectos de implemntación más relevantes que deben tenerse en cuenta en la creación de los mismos.

3.1 Introducción

Los procesadores de lenguajes de programación, son programas que *procesan* a otros programas. Aunque la mayor parte de las aplicaciones de los procesadores de lenguajes están en el desarrollo de compiladores e intérpretes, también están presentes en los analizadores XML de la mayoría de los sistemas de comercio electrónico, editores con sintaxis resaltada como los que usan los programadores, analizadores de ficheros con configuraciones complejas como Sendmail o Apache, analizadores de gramáticas como los que utilizan las aplicaciones para procesar textos o en entornos de desarrollo basados en componentes, en los que se usan pequeños lenguajes para unir dichos componentes.

Como se ha visto en el capítulo 2, los lenguajes de programación juegan un papel muy importante en el campo de las ciencias de la computación. Se pueden considerar como las herramientas fundamentales con las que trabajan los desarrolladores de software que dependen completamente de la calidad de los procesadores que usen.

Existe una interacción entre el diseño de los lenguajes de programación y el conjunto de instrucciones de un ordenador: son los procesadores los que deben unir el salto semántico entre los distintos lenguajes y el código máquina como se verá en el capítulo 4.

3.2 Clasificación y estructura

A los traductores, compiladores, intérpretes, ensambladores y otros programas que realizan operaciones con lenguajes se les aplica el nombre genérico de

procesadores de lenguajes. Por tanto, el término *procesadores de lenguajes* engloba a todas las aplicaciones informáticas en las cuales uno de los datos fundamentales de entrada es un lenguaje.

Puede utilizarse la siguiente definición formal [Ter00].

Definición 3.1 (Procesador de lenguajes) *Un procesador de lenguaje es una función cuyo dominio es un lenguaje fuente y cuyo rango está contenido en un lenguaje objeto*.

La definición anterior afecta a una gran variedad de herramientas software. La siguiente tabla recoge algunas de ellas.

Término en castellano	Término en inglés
Traductores	Translators
Compiladores	Compilers
Intérpretes	Interpreters
Ensambladores	Assemblers
Montadores de enlaces	Linkers
Cargadores	Loaders
Desensambladores	Dissemblers
Decompiladores	Decompilers
Depuradores	Debuggers
Analizadores de rendimiento	Profilers
Optimizadores de código	Code optimizers
Compresores	Compressors
Preprocesadores	Preprocessors
Formatedores	Formatters
Editores	Editors

Tabla 3.1 Herramientas software que procesan lenguajes

3.2.1 Traductores

Definición 3.2 (Traductor) *Un traductor transforma un programa escrito en un lenguaje (denominado lenguaje fuente) en otro programa escrito en otro lenguaje (denominado lenguaje objeto).*

La ejecución de un programa mediante un traductor (*figura 3.1*), requiere dos fases. Una primera fase de traducción al lenguaje objeto y una segunda fase de ejecución del programa resultante.

El *lenguaje fuente* es el lenguaje origen que transforma el traductor (por ejemplo C, C++, Pascal, Fortran, PL/I, ADA, MODULA-2, ...). También pueden ser lenguajes de bajo nivel.

El *lenguaje objeto* es el lenguaje al que se traduce el texto fuente. Los lenguajes objeto pueden ser de alto nivel, o bien un lenguaje ensamblador o lenguaje máquina de un procesador determinado.

El *lenguaje de implementación* del traductor o lenguaje host puede ser cualquier lenguaje de programación, desde un lenguaje de alto nivel a un lenguaje máquina.

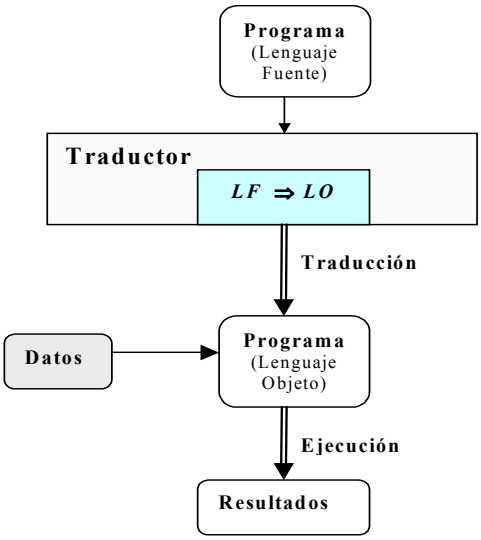


Figura 3.1: Esquema general de un traductor

3.2.2 Compiladores

Definición 3.3 (Compilador) *Un compilador es un traductor que transforma un programa escrito en un lenguaje de alto nivel (lenguaje fuente) en otro programa escrito en un lenguaje de bajo nivel (lenguaje objeto).*

En la *figura 3.2* se puede observar el esquema general de un compilador.

3.2.2.1 Compilador incremental

Algunos lenguajes, como *Lisp* o *Prolog*, permiten realizar modificaciones del propio programa fuente durante su ejecución. Esta característica limita la posibilidad de implementar un compilador íntegro ya que durante la ejecución de un programa no sería posible acceder al código fuente de dicho programa.

Los programas escritos en este tipo de lenguajes pueden dividirse en una parte estática, que no es susceptible de modificaciones durante la ejecución, y una parte dinámica, que puede sufrir modificaciones durante la ejecución.

La compilación incremental es una técnica que compila únicamente las partes estáticas del programa. Posteriormente, al ejecutar el programa, las partes dinámicas no compiladas son interpretadas.

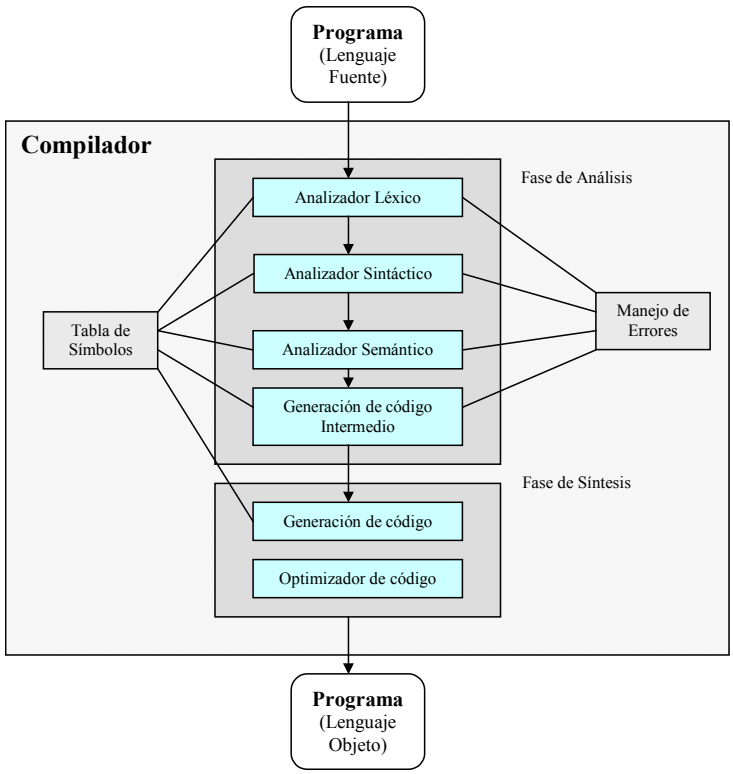


Figura 3.2: Esquema general de un compilador

3.2.2.2 Compilación *Just in time*

La compilación *Just in Time* es una técnica que se aplica fundamentalmente a entornos distribuidos en los que un cliente desea ejecutar un programa que reside en un servidor.

Con la compilación tradicional, sería necesario solicitar todo el programa fuente, compilarlo y ejecutarlo. Con la compilación *Just in time* se solicita un módulo del programa fuente, se compila y se ejecuta dicho módulo. Si el módulo realiza una llamada a otro módulo, se solicita entonces dicho módulo, se compila y se ejecuta, y así sucesivamente.

La ventaja para el usuario es que puede comenzar a obtener resultados parciales de la ejecución antes que con la compilación tradicional. Además, en el caso de programas grandes, no siempre es necesario compilar todos los módulos, ya que algunos pueden no utilizarse.

La compilación continua [Ple97] es una mejora de la compilación *Just in time* en la que se solicitan módulos y se ejecutan de forma paralela.

3.2.3 Intérpretes

Definición 3.4 (Intérprete) *Un intérprete es un programa que analiza y ejecuta de forma simultánea un programa.*

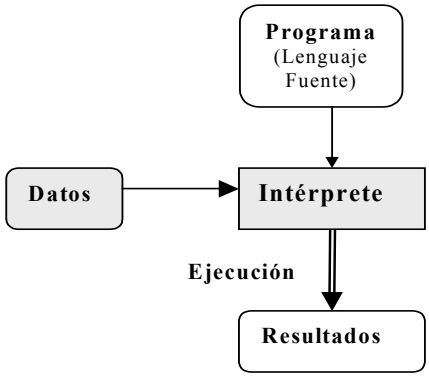


Figura 3.3: Esquema general de un intérprete

Los intérpretes (*figura 3.3*) tienen las siguientes características:

- La implementación de un intérprete es relativamente sencilla y flexible, siguiendo fielmente la semántica del lenguaje a interpretar y adaptándose a cambios con facilidad.

- En ocasiones, un intérprete puede utilizarse como modelo para la descripción semántica del lenguaje. Dicho intérprete se denomina implementación prototipo del lenguaje. La semántica operacional de un lenguaje puede considerarse un caso especial de implementación prototipo en el que el intérprete se desarrolla en un lenguaje suficientemente abstracto.

- El intérprete no requiere disponer de todo el código del programa a ejecutar al principio de la ejecución. Esta característica se aplica en los intérpretes de comandos en los que el código a interpretar es introducido de forma interactiva por el usuario. Otra aplicación son los sistemas distribuidos: al ejecutar desde un cliente programas que residen en un servidor no es necesario traer todos los programas completos, sino que es posible ir ejecutando porciones.

- Los intérpretes permiten acceder al propio código del programa fuente durante la ejecución. Lenguajes como *Lisp, Prolog* o *Smalltalk* explotan esta característica permitiendo construir programas que se manipulan a sí mismos en tiempo de ejecución.

3.3 Construcción de procesadores de lenguajes

Un procesador debe estar escrito en un lenguaje de programación, que es conocido con el nombre de lenguaje *host* o *lenguaje de implementación*. Hoy en día es difícil encontrar procesadores escritos en lenguaje máquina, incluso los procesadores que se desarrollan para un nuevo hardware se implementan mediante el uso de lenguajes de alto nivel, aplicando técnicas de *bootstrapping* y compilación cruzada (*cross-compiler)*.

El primer procesador desarrollado con éxito se produjo a finales de los 50 para el lenguaje FORTRAN [BGH57]. Fueron necesarios varios años para desarrollar el compilador, ya que el lenguaje se fue diseñando al mismo tiempo que se iba implementando.

En la actualidad el diseño y la implementación de procesadores se lleva a cabo de forma rápida y eficaz debido fundamentalmente a la rápida evolución de los lenguajes, el uso de algoritmos y estructuras de datos y la existencia de herramientas que generan de forma automática la fase de análisis del procesador.

3.3.1 Analizadores léxicos, sintácticos y semánticos

Definición 3.5 (Analizador léxico) *Un analizador léxico para un lenguaje es un programa que reconoce los componentes léxicos (tokens) del programa fuente.*

La misión del analizador léxico consiste en enviar al analizador sintáctico los tokens y sus atributos.

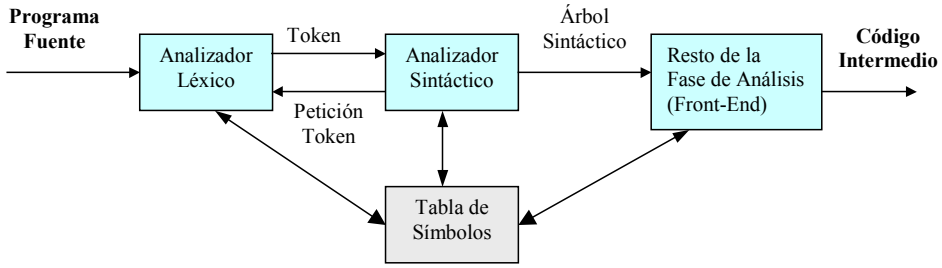


Figura 3.4: Esquema general del análisis sintáctico

Definición 3.6 (Analizador sintáctico) *Un analizador sintáctico para un lenguaje es un programa que recibe como entrada los elementos individuales o componentes léxicos (tokens) del programa fuente y determina la estructura de las sentencias o instrucciones que lo forman.*

La misión del analizador sintáctico consiste en:

- Detectar e informar de los errores sintácticos.
- Generar un árbol sintáctico a partir del cual pueda ser generado el código intermedio.

El esquema general del análisis sintáctico se puede observar en la figura 3.4. Para que un programa expresado en un lenguaje de programación pueda ser analizado, es necesario describir dicho lenguaje mediante una **gramática** y para la mayoría de los lenguajes de programación es suficiente realizar la descripción mediante **gramáticas libres de contexto (Context Free Grammar - CFGs)**.

Una CFG contiene un conjunto de reglas que definen cómo está estructurado el lenguaje. Las reglas son patrones específicos de datos que aparecen en la entrada, pueden ser recursivas y hacer referencias a otras reglas denominadas subreglas. Las producciones, son las diferentes alternativas por las cuales la regla puede ser satisfecha.

Ventajas derivadas del uso de gramáticas

- Una gramática da una especificación sintáctica precisa de un lenguaje.

- A partir de determinadas clases de gramáticas, es posible construir de forma automática y con los generadores existentes, analizadores sintácticos eficientes. Estos generadores pueden detectar también ambigüedades sintácticas y otros problemas de forma automática.

- Un procesador basado en una descripción gramatical de un lenguaje es más fácil de mantener y extender.

3.3.1.1 Tipos generales de analizadores sintácticos

Analizadores Universales

Los algoritmos más referenciados para reconocer las gramáticas libres de contexto son el algoritmo CYK introducido en 1963 por Cocke, Younger and Kasami [You67], y el algoritmo de Earley [Ear70] introducido en in 1970 como una extensión del método LR(K) de Knuth [Knu65]. El primero opera solamente con gramáticas en la forma normal de Chomsky mientras que el segundo trata las gramáticas libres de contexto en general. Ambos algoritmos no se consideran demasiado eficientes para ser utilizados de forma generalizada.

Analizadores Descendentes

Construyen el árbol sintáctico de la sentencia a reconocer desde el símbolo inicial (raíz), hasta llegar a los símbolos *terminales* (hojas) que forman la sentencia, usando derivaciones más a la izquierda.

Los principales problemas que se plantean son dos: el retroceso (*bakctraking*) y la recursividad a izquierdas.

El análisis dirigido por sintaxis, en la forma de análisis recursivo descendente, fue propuesto por primera vez de forma explícita en 1961 para describir un

compilador simplificado de ALGOL 60, mediante un conjunto de funciones recursivas que correspondían a la notación BNF. El problema fundamental para el desarrollo de este método fue el *retroceso*, lo que hizo que su uso practico fuera restringido. *La elegancia y comodidad de la escritura de compiladores dirigidos por sintaxis, fue pagada en tiempo de compilación por el usuario*.

La situación cambió cuando comenzó a realizarse el análisis sintáctico sin retroceso, por medio del uso de gramáticas LL(1), obtenidas independientemente por Foster (1965) y Knuth (1967), generalizadas posteriormente por Lewis, Rosenkrantz y Stearns en 1969, dando lugar a las gramáticas LL(k), que pueden realizar un análisis sin retroceso, en forma descendente y sin recursividad a izquierdas, reconociendo la cadena de entrada de izquierda a derecha (**L**eft to rigth), tomando las derivaciones más a la izquierda (**L**eftmost) y examinando en cada paso, todos los símbolos procesados anteriormente y los k símbolos de la entrada.

Los algoritmos que realizan el análisis descendente deben cumplir al menos dos condiciones:

- Saber en todo momento dónde se encuentra dentro del árbol sintáctico.
- Poder elegir la regla de producción que se aplica de forma correcta.

Analizadores Ascendentes

Construyen el árbol sintáctico de la sentencia a reconocer, desde las hojas hasta llegar a la raíz. Son analizadores del tipo *reducción-desplazamiento* (*shift-reduce*), parten de los distintos *tokens* de la sentencia a analizar, y por medio de reducciones llegan al símbolo inicial de la gramática.

El principal problema que se plantea en el análisis ascendente es el retroceso. Para solventar este inconveniente, se definieron distintos tipos de gramáticas entre las cuales las más utilizadas son las **LR(k)** ya que realizan eficientemente el análisis ascendente sin retroceso. Los analizadores sintácticos ascendentes que reconocen lenguajes descritos por gramáticas LR(k), toman sus decisiones en función de los **k** *tokens* inspeccionados por delante (*lookaheads*) en la cadena de entrada, tomados de izquierda a derecha (**L**eft to right), realizándose el análisis por derivaciones más a la derecha en sentido inverso (**R**ightmost).

Las gramáticas LR(k) describen la mayor parte de los lenguajes libres de contexto, y son menos restrictivas que las gramáticas LL(k). Sin embargo, la complejidad de los analizadores sintácticos LR(k) a la hora de construir las tablas de análisis LR (SLR, LR canónico y LALR), hace que la mayor parte de éstos se construyan mediante herramientas software.

	Analizadores S.	Analizadores S. Ascendentes
Ventajas	Fácil de codificar Fácil de depurar Código más pequeño	Permite la recursividad a izquierdas
Desventajas	No puede tratar la recursividad a izquierdas El retroceso es lento	Más difíciles de depurar El código puede ser bastante largo La ejecución de las acciones puede ser impredecible La estructura de datos que gestiona la recursividad no es aceptable

Tabla 3.2 Ventajas y desventajas de los analizadores sintácticos ascendentes y descendentes

Definición 3.6 (Analizador semántico) *Un analizador semántico detecta la validez semántica de las sentencias aceptadas por el analizador sintáctico.*

En el análisis semántico se realizan fundamentalmente las comprobaciones de tipos, y se convierte el árbol sintáctico en una representación intermedia denominada código intermedio.

No siempre se establece la división entre la sintaxis y la semántica de forma clara y precisa. Existen determinadas tendencias a especificar producciones usando nombres con connotaciones semánticas, y a utilizar formas que reflejan la precedencia de los operadores por ejemplo.

El tratamiento formal de la semántica se está llevando a cabo mediante el uso de diversas técnicas, como la semántica operacional, semántica denotacional, semántica axiomática y las gramáticas atribuidas.

En esta tesis no se realiza un estudio de las distintas aproximaciones pero sí se usa una técnica formal para su tratamiento. Se proporciona a través de su descripción mediante la utilización de lenguajes de programación, aplicando el formalismo que impongan dichos lenguajes.

3.3.2 Generadores de procesadores

Aplicando la definición utilizada en [GH98] podemos decir que:

Definición 3.7 (Generador de procesadores) *Un generador de procesadores es un programa que transforma una especificación en un procesador para el lenguaje de programación descrito en la especificación* .

En la sección 3.3.1.1 se argumentó que los analizadores sintácticos ascendentes son difíciles de implementar "manualmente" puesto que la construcción de las tablas de análisis LR es una tarea ardua y tediosa. En contraposición, los analizadores recursivos descendentes, son relativamente fáciles de implementar de forma manual puesto que el código tiende a seguir las directrices marcadas por

la gramática. De forma similar, la construcción manual de analizadores léxicos para muchas aplicaciones resulta ser algo relativamente fácil de conseguir siguiendo las técnicas habituales para el reconocimiento de los tokens.

Sin embargo, como en muchos proyectos reales de programación, cuando se trata de desarrollar procesadores de cierta envergadura, la complejidad aumenta. Incluso aplicando técnicas en las que las fases de construcción del procesador puedan ser encapsuladas y separadas en módulos, el software resultante es difícil de entender y mantener, especialmente de forma que sea portable.

3.3.2.1 Estructura

La *figura 3.5* muestra cómo a partir de la entrada al generador de procesadores, un *metalenguje,* que es un lenguaje que describe otro lenguaje, se genera un procesador de forma automática.

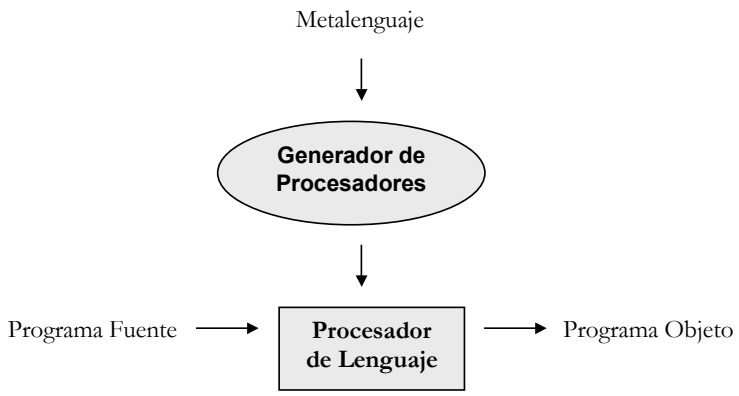


Figura 3.5: Esquema general de un generador de procesadores de lenguajes

En el proceso de generación de procesadores, se consideran tres tipos de tiempos.

Tiempo de generación.

Cuando el generador es ejecutado. A partir de una entrada en forma de metalenguaje, se obtiene como salida el procesador de lenguaje correspondiente.

Tiempo de compilación.

Cuando la salida obtenida en tiempo de generación es ejecutada. Se parte de un programa fuente como entrada al procesador de lenguaje y se traduce a un programa objeto.

Tiempo de ejecución.

Cuando el código generado por el procesador de lenguaje es ejecutado por el sistema operativo de la máquina.

3.3.2.2 Características

Los generadores de procesadores de lenguajes se diseñan generalmente con un modelo particular de lenguaje, así la herramienta está más indicada para generar procesadores de lenguajes similares a ese modelo.

Para conseguir un software satisfactorio, los generadores ocultan los detalles de la implementación al usuario. Debido a que casi todas las herramientas generan solamente una parte del procesador, es muy importante que pueda ser integrada con las diferentes partes del procesador que no hayan sido generadas. Se puede decir que el grupo más importante de generadores automatizan la fase de análisis del procesador, la fase de síntesis se construye de forma manual.

Podríamos destacar las siguientes características.

Metalenguajes

Un metalenguaje describe a otro lenguaje o algunos aspectos de otro lenguaje. Existen diversos metalenguajes que son muy conocidos, las *expresiones regulares* para describir tokens, la *notación BNF* (Backus-Naur-Form) para describir la sintaxis de los lenguajes de programación o las *gramáticas atribuidas* para describir la semántica de los lenguajes añadiendo atributos y funciones semánticas a la notación BNF. Algunos metalenguajes pueden usarse para describirse a ellos mismos.

Principales propiedades que debería tener un metalenguaje:

- Fácil de leer, aprender y usar.
- Integrado, que tenga notaciones similares para las funciones a través de las diferentes partes del generador. Por ejemplo, el símbolo "=" debería ser usado como operador de asignación para la parte de las expresiones regulares, la sintaxis y las funciones semánticas.
- De forma análoga a los lenguajes de programación, no debería incorporar construcciones que permitan cometer errores fácilmente.

Funcionalidad

Las características funcionales de un generador de procesadores es describir cómo trabaja el sistema, sus capacidades y los aspectos normales del software como la extensibilidad, robustez e integración.

Para conseguir una buena funcionalidad es necesario que:

- La herramienta sea robusta. Los errores cometidos por el usuario deben ser gestionados de forma adecuada.

- La herramienta sea eficiente en tiempo y espacio. Es necesario utilizar las estructuras adecuadas en todo momento.

- La herramienta se pueda extender e integrar con las diferentes partes del procesador.

Documentación

La documentación describe la herramienta, y al igual que para cualquier aplicación software a gran escala, se divide en dos partes: el manual de usuario y la documentación del sistema donde se describen los aspectos técnicos de la herramienta. En este caso, los límites entre los dos está un poco confuso ya que con los detalles técnicos es necesario conocer como se usan, y para usarlo es necesario conocer los aspectos técnicos.

3.3.2.3 Clasificación

Los generadores de procesadores se encuentran en general separados en distintas fases. De forma análoga a la división establecida en la fase de análisis (*front-end*) de un compilador, se establece la siguiente clasificación:

- Generadores de analizadores léxicos (*scanner*)

- Generadores de analizadores sintácticos (*parser*)

- Generadores de analizadores semánticos

Para la *fase de síntesis* (back-end) de un compilador se están llevando a cabo investigaciones para la obtención de *generadores de generadores de código*.

El la *figura 3.6* se indica cómo un generador de procesadores puede ser parametrizado. Las cajas rectangulares muestran cada una de las fases estándar de un procesador y las cajas redondeadas muestran los correspondientes generadores. La fase de análisis se encuentra agrupada porque por lo general existe un único fichero de entrada conteniendo toda la información necesaria para generar la *fase de análisis* (front-end) de un compilador.

La mayor parte de los generadores existentes hoy en día, no son tan ambiciosos como para poder resolver todos los problemas que afronta el diseñador de procesadores. De hecho, de todas las tareas, algunas de ellas se prestan más fácilmente a automatización que otras. Dentro de la *fase de análisis*, por ejemplo, las tareas de análisis léxico y análisis sintáctico han sido estudiadas ampliamente y pueden ser formalizadas, por lo que parece obvio que sea posible construir herramientas que las automaticen. Sin embargo, la tarea de análisis semántico es mucho más compleja de automatizar, no existen mecanismos formales tan populares como en el caso anterior, por lo que es difícil encontrar herramientas que puedan ayudar en esta tarea.

Como consecuencia de lo anterior, nos encontramos con multitud de herramientas que automatizan la creación de analizadores léxicos y sintácticos, y muy pocas que intenten resolver de manera efectiva la automatización del análisis semántico, lo cual es lógico, puesto que esta etapa es la que más cambia según las circunstancias del lenguaje que soporte el compilador. Por esta razón en lo sucesivo, vamos a centrarnos de manera exclusiva en herramientas del primer tipo.

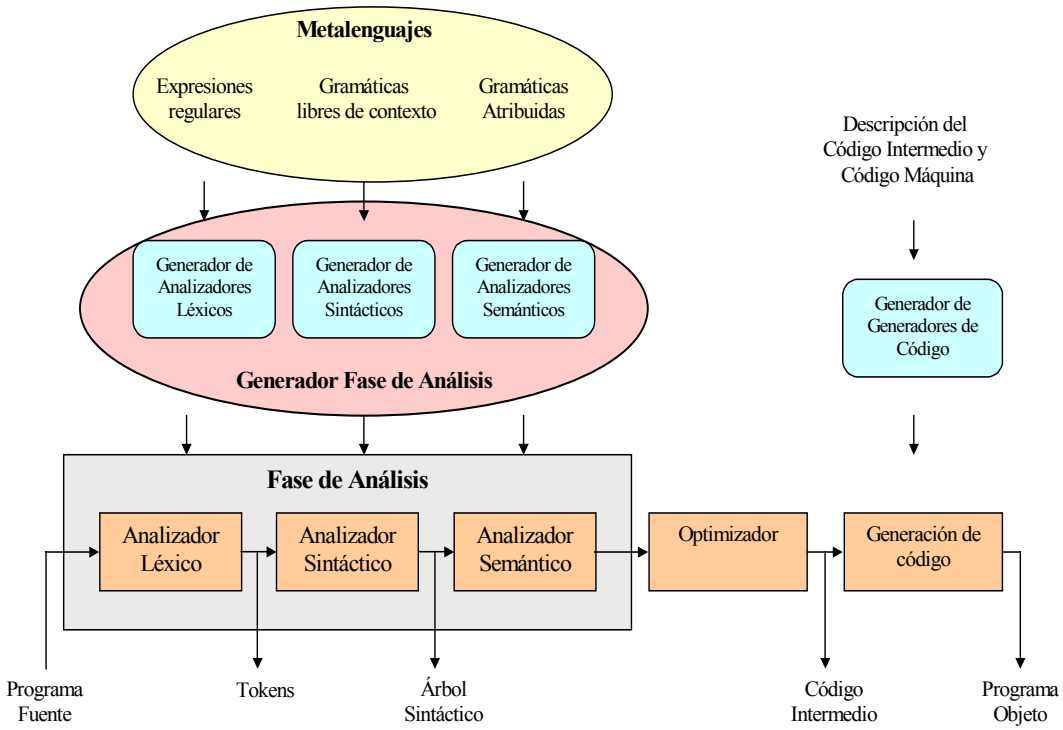


Figura 3.6: Parametrización de un generador de procesadores de lenguajes

Si bien es cierto que hay un amplio número de herramientas pensadas para facilitar el trabajo del diseñador de lenguajes de programación, si hay una que tenga especial importancia por su difusión y trascendencia es YACC, Yet Another Compiler Compiler, obtenida en 1975 [Joh75]. En principio fue creada para el sistema operativo UNIX y está asociada con otra herramienta llamada LEX [Les75] que genera analizadores léxicos. Esta herramienta ha sentado un precedente de tal forma que la mayoría de las que han aparecido posteriormente, se presentan siempre en comparación con YACC, como se verá posteriormente en el capítulo 4.

3.3.2.4 Generadores de analizadores léxicos

Definición 3.7 (Generador de analizadores léxicos) *Un generador de analizadores léxicos, es un programa que convierte una notación formal por medio de expresiones regulares, en un programa (analizador léxico) que puede reconocer los componentes léxicos de un programa fuente.*

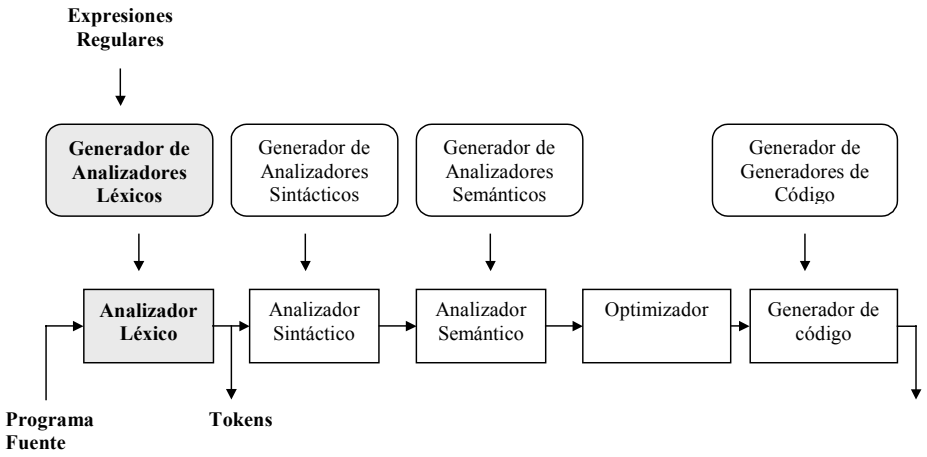
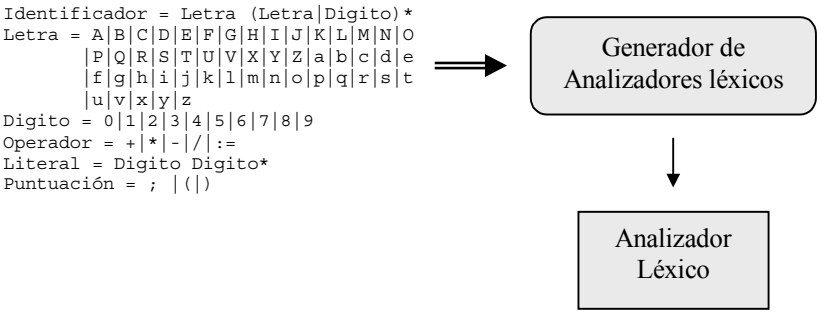


Figura 3.7: Esquema de un generador de analizadores léxicos

En la *figura 3.7* se muestra cómo la entrada al generador es un metalenguaje formado por expresiones regulares. Estas expresiones describen los componentes léxicos que el analizador léxico deberá reconocer del programa fuente.

Ejemplo 3.1 *En el siguiente ejemplo se muestran las expresiones regulares de los componentes léxicos de una sentencia de asignación típica.*



Ejemplo 3.2 *En este ejemplo se muestra una posible entrada y una salida para el analizador léxico generado anteriormente.*

3.3.2.5 Generadores de analizadores sintácticos

Definición 3.8 (Generador de analizadores sintácticos) *Un generador de analizadores sintácticos, es un programa que convierte una gramática en un programa (analizador sintáctico) que puede reconocer el lenguaje que describe dicha gramática.*

En la *figura 3.8* se muestra cómo la entrada al generador es un metalenguaje representado mediante una gramática libre de contexto. Esta gramática describe las construcciones sintácticas del programa fuente que el analizador deberá reconocer.

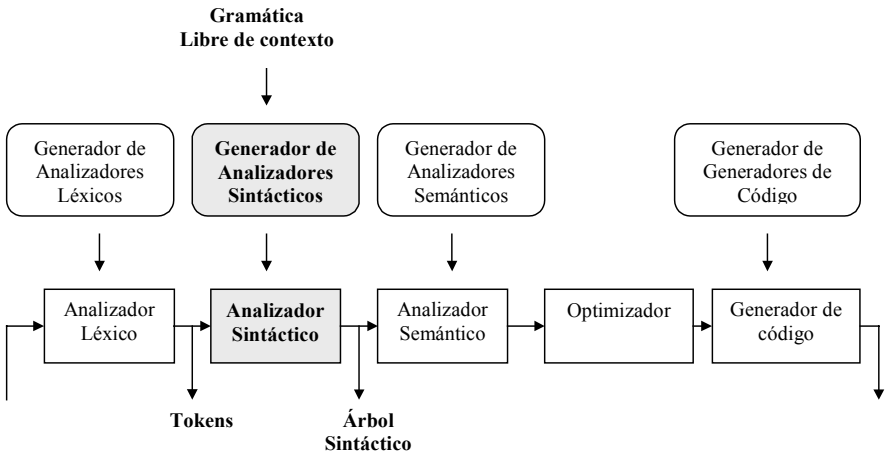
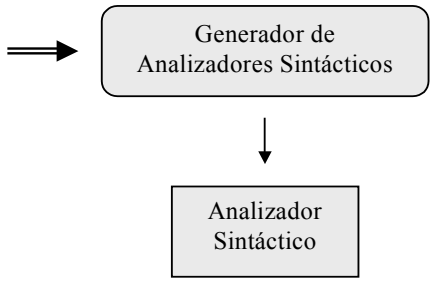


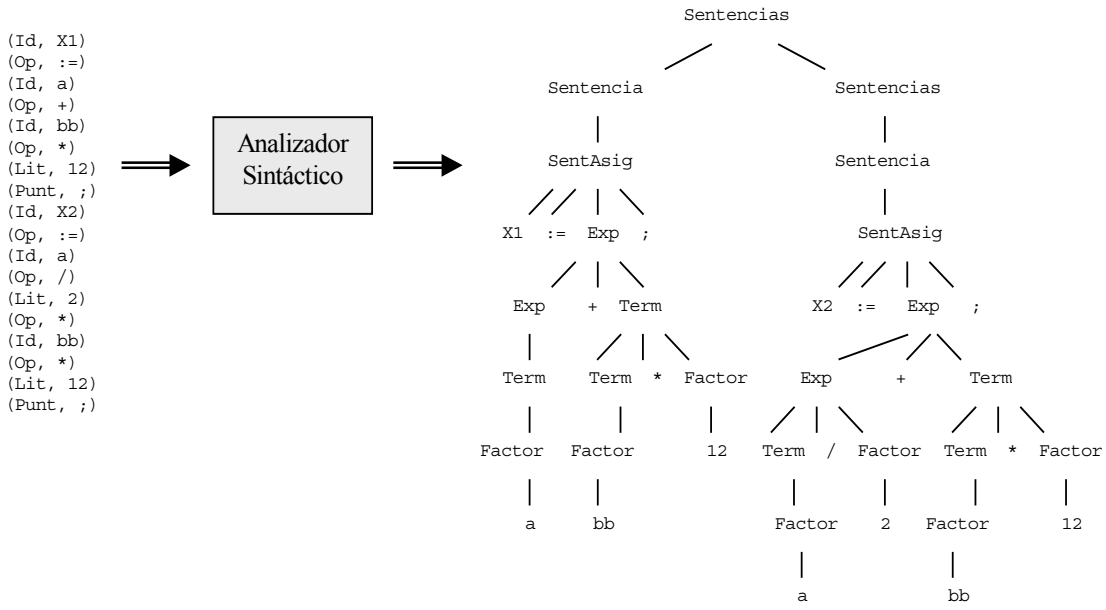
Figura 3.8: Esquema de un generador de analizadores sintácticos

Ejemplo 3.3 *En el siguiente ejemplo se muestra la gramática expresada en notación BNF de un programa para reconocer sentencias formadas por expresiones aritméticas.*

```
Programa → Sentencias
Sentencias → Sentencia Sentencias | Sentencia
Sentencia → SentAsig
SentAsig → Identificador := Exp
Exp → Exp + Term | Exp - Term | Term
Term → Term * Factor | Term / Factor | Factor
Factor → ( Exp) | Identificador | Literal
```



Ejemplo 3.4 *En este ejemplo se muestra el árbol sintáctico obtenido después de que los componentes léxicos hayan sido analizados.*



3.3.2.6 Generadores de analizadores semánticos

La entrada a un generador de analizadores semánticos es por norma general una gramática atribuida (*figura 3.9*). Esta gramática integra una gramática libre de contexto y un conjunto de ecuaciones denominadas funciones que permiten realizar una especificación del lenguaje.

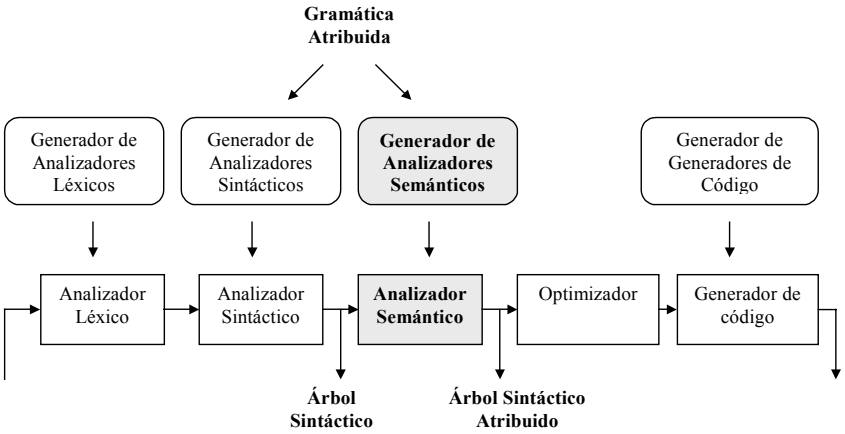
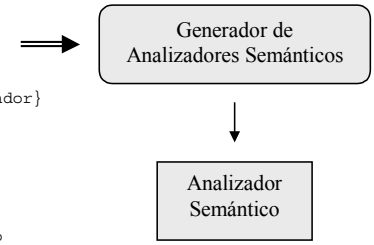


Figura 3.9: Esquema de un generador de analizadores semánticos

Ejemplo 3.5 *En el siguiente ejemplo se muestra la entrada al generador de analizadores semánticos, una gramática atribuida para reconocer sentencias formadas por expresiones aritméticas.*

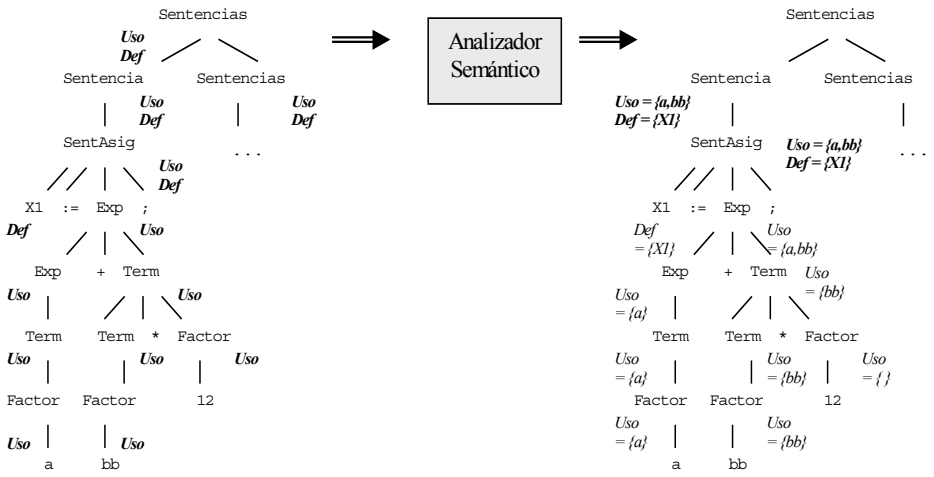
Programa → Sentencias	Programa.Uso = Sentencias.Uso Programa.Def = Sentencias.Def
$Sentencias_1$ → Sentencia $Sentencias_2$	$Sentencias_1$.Uso = Sentencia.Uso ∪ $Sentencias_2$.Uso $Sentencias_1$.Def = Sentencia.Def ∪ $Sentencias_2$.Def
| Sentencia	$Sentencias_1$.Uso = Sentencia.Uso $Sentencias_1$.Def = Sentencia.Def
Sentencia → SentAsig	Sentencia.Uso = SentAsig.Uso Sentencia.Def = SentAsig.def
SentAsig → Identificador := Exp	SentAsig.Def = {VolorLex(Identificador} SentAsig.Uso = {Exp.Uso}
Exp_1 → Exp_2 + Term	Exp_1.Uso = Exp_2.Uso ∪ Term.Uso
| Exp_2 – Term	Exp_1.Uso = Exp_2.Uso ∪ Term.Uso
| Term	Exp_1.Uso = Term.Uso
$Term_1$ → $Term_2$ * Factor	$Term_1$.Uso = $Term_2$.Uso ∪ Factor.Uso
| $Term_2$ / Factor	$Term_1$.Uso = $Term_2$.Uso ∪ Factor.Uso
| Factor	$Term_1$.Uso = Factor.Uso
Factor → (Exp)	Factor.Uso = Exp.Uso
| Identificador	Factor.Uso = {VolorLex(Identificador}
| Literal	Literal.Uso = { }



Los atributos *uso* y *def* están asociados a distintos símbolos *no terminales* y sirven para describir información sobre el uso y la definición de las variables usadas en un programa. Esta información puede ser utilizada para finalizar el análisis, mostrando los correspondientes mensajes al usuario, o bien ser utilizada por la fase de optimización, eliminando variables que fueron definidas pero nunca usadas, etc.

Durante el análisis sintáctico, los nodos del árbol serán etiquetados con esos atributos y durante el análisis semántico, los valores de esos atributos serán computados mediante la propagación de la información a través de todo el árbol sintáctico.

Ejemplo 3.6 *En este ejemplo se muestra (de forma parcial) el resultado obtenido con el analizador semántico generado anteriormente .*



Aquí los valores de *Uso* y *Def* en el nodo raíz pueden ser consultados para obtener la mayor información posible. En el ejemplo, los identificadores `X1`, `X2` estarían definidos pero no se habrían usado, y `a`, `bb` si se habrían usado pero no estarían definidos.

La mayoría de los generadores permiten integrar las acciones semánticas de forma intercalada entre las producciones de la gramática en notación BNF. En este caso, la evaluación de los atributos se realiza durante el proceso de análisis sintáctico.

3.3.2.7 Ventajas

Las *ventajas* de utilizar herramientas específicas para construir procesadores, se podrían resumir en los siguientes puntos:

- Atenuar la complejidad.
- Favorecer su calidad
- Disminuir el tiempo y los costes necesarios para su desarrollo.

3.3.2.8 Resumen

Como se ha visto, los generadores son herramientas que ayudan a construir procesadores de lenguajes generando alguna o todas las fases del procesador de forma automática. A partir de una especificación, algunos generadores construyen procesadores completos (realizan el análisis y la generación-optimización de código para máquinas reales) mientras que otros, solamente construyen la fase de análisis de un compilador (generan analizadores léxicos y/o sintácticos). En el plano teórico, puede parecer que un generador que construya todas las fases del procesador será más potente, sin embargo, en la práctica, los generadores que construyen únicamente analizadores léxicos y/o sintácticos están integrados normalmente con un lenguaje de programación de propósito general. De esta forma, la implementación de estructuras de datos complejas, optimizaciones y análisis de código, son fáciles de realizar ya que se utiliza el lenguaje de programación nativo del programador.

Los primeros generadores de la *fase de análisis* (front-end) aparecieron a principios de los 60 y hasta la actualidad se han venido desarrollando numerosas herramientas. Existen generadores de analizadores léxicos y sintácticos, para la mayoría de los lenguajes de programación utilizados hoy en día.

En la descripción de la arquitectura del sistema, presentada en el capítulo 8, se ha optado por una solución intermedia entre los generadores parciales y los generadores totales. Se va a utilizar un marco de trabajo (*framework*) para crear procesadores de lenguaje de manera automática, de forma que no sólo se va a realizar un análisis léxico, sintáctico y semántico del lenguaje fuente, sino que además, se va a permitir añadir el código necesario para realizar una interpretación o traducción del mismo a lenguaje objeto (*generación de código*).

Nos centraremos en la generación de código para máquinas abstractas orientadas a objetos, no máquinas reales (su justificación se presenta en el capítulo 6). Esto permite al programador trabajar con lenguajes de programación sin un marcado salto semántico.

CAPÍTULO 4

PANORAMA DE DIVERSAS TÉCNICAS DE CONSTRUCCIÓN DE PROCESADORES DE LENGUAJES

En este capítulo se realiza una descripción general de las técnicas utilizadas en la construcción de procesadores de lenguajes que se han venido desarrollando en los últimos años y que están relacionadas con los objetivos del sistema propuesto en la presente tesis. Se trata de analizar sus características, destacando sus ventajas e inconvenientes y detectar posibles estrategias que sean de utilidad para el diseño de una arquitectura para el sistema de desarrollo de procesadores de lenguajes descrito en el capítulo 8.

Actualmente existen muchos sistemas que permiten automatizar la construcción de procesadores de lenguajes. Los sistemas que se revisan a continuación, intentan ser representativos de las diferentes tendencias y son sólo una selección de los sistemas examinados.

4.1 Lex/YACC

Lex [Les75] y YACC [Joh75] (*Yet Another Compiler Compiler*) son una pareja de herramientas que pueden ser usadas juntas para generar un compilador completo o únicamente su fase de análisis (*front-end*).

Existen numerosas variaciones y su influencia se deja sentir en otras muchas herramientas de construcción de procesadores de lenguajes en uso actualmente. Aunque la versión más popular es la del proyecto GNU (*Free Software Foundation*) con las herramientas Flex [Pax95] y Bison [DS95] existen una larga lista, citando a modo de ejemplo las herramientas PCYACC [Pcy00] y PCLEX [Pcl00] así como las versiones realizadas para el lenguaje Java denominadas Jlex [Ber97] y CUP (Constructor of Useful Parsers) [Hud97].

Esta popularidad se debe en parte a que YACC se incluye de serie desde hace bastante tiempo en el sistema operativo Unix junto con otras herramientas de programación que también se han vuelto muy populares. No obstante, es indudable que buena parte de su éxito se debe sobre todo, a las cualidades técnicas de la aplicación.

4.1.1 Características generales

El generador de analizadores léxicos (Lex) es usado para dividir una secuencia de caracteres en componentes léxicos (*tokens*). A partir de una especificación que asocia expresiones regulares con acciones, Lex construye una función implementando un autómata finito determinístico (DFA) que reconoce expresiones regulares. En tiempo de ejecución, cuando una expresión regular es reconocida, su acción asociada es ejecutada.

El generador de analizadores sintácticos es YACC. A partir de una especificación que contiene la gramática del lenguaje para el cual se desea generar el procesador y las acciones asociadas a las alternativas de las producciones de la gramática, genera un analizador que ejecutará el código de acción asociado a cada alternativa tan pronto como sean reconocidos los símbolos de dicha alternativa.

YACC genera analizadores ascendentes (*figura 4.1*). Estos analizadores se basan en la construcción de tablas de análisis LALR(1).

Es interesante notar que tanto YACC, como buena parte de las herramientas compatibles, generan analizadores sintácticos en el lenguaje de programación C. Existen varias razones para que esto sea así, incluyendo el hecho de que YACC es una herramienta que apareció en el mundo Unix, sistema operativo implementado en dicho lenguaje. C tiene entre otras importantes ventajas, un buen grado de portabilidad y eficiencia, sobre todo para los analizadores sintácticos generados. Sin embargo como veremos más adelante, otras herramientas de aparición más reciente han optado por otros lenguajes de programación.

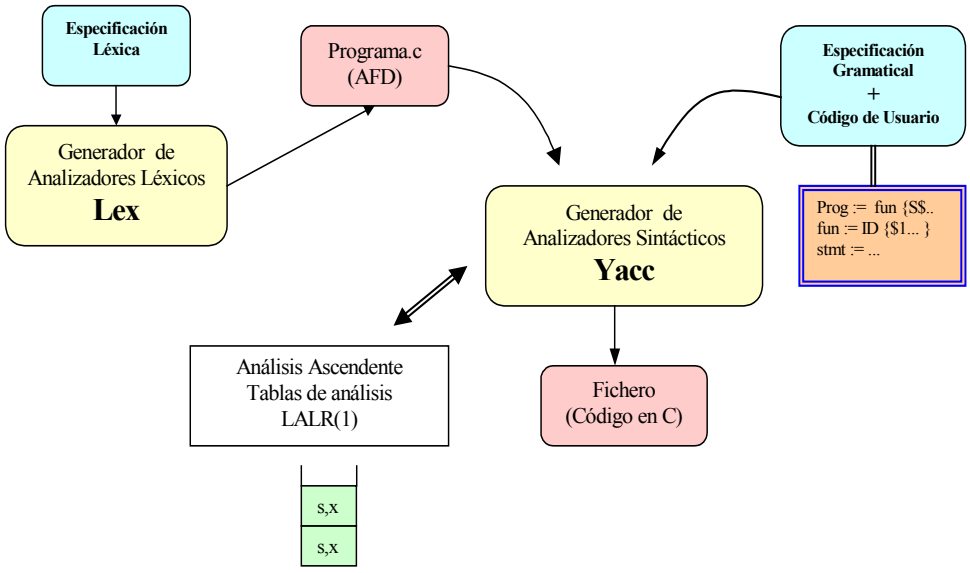


Figura 4.1: Arquitectura de Lex/YACC

4.1.2 La entrada de YACC

YACC espera como entrada una especificación que se compone de tres partes separadas por dos símbolos de tanto por ciento (%%) y cuya estructura es la siguiente:

```
Definiciones previas en C
%%
Definición gramatical
%%
Funciones en C
```

La primera parte, la de definiciones previas en C contiene exclusivamente declaraciones de variables, funciones y librerías que van a ser empleadas en alguna de las dos partes siguientes.

La segunda parte es la más importante. En ella se introduce la especificación gramatical del lenguaje del analizador sintáctico utilizando una notación similar a la expresada en BNF.

La tercera parte, la de funciones en C, consiste en un conjunto de funciones que complementarán a las que genera YACC a partir de la especificación gramatical, para poder obtener un programa funcionalmente completo. Entre las funciones necesarias se encuentra la inevitable función main() de todo programa C, que se encarga de iniciar el proceso junto con la función yyparse(), y una función yylex() que se encarga de obtener los tokens del analizador léxico para que sean usados por el analizador sintáctico.

Tanto la primera como la última sección del fichero de entrada serán copiadas de manera integra y sin ningún tratamiento intermedio en el fichero generado a la salida por YACC.

Ejemplo 4.1 *El siguiente ejemplo muestra la especificación completa para YACC de una pequeña calculadora. Contiene 26 registros nombrados de la 'a' a la 'z' y acepta expresiones aritméticas, con los operadores +, -, *, /, % (módulo), & (and), | (or) y asignación.*

```
%{
# include <stdio.h>
# include <ctype.h>
int regs[26];
int base;
%}
%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* precedencia para el menos unario */
```

```
%% /* Comienza la sección de reglas */
list : /* empty */
     | list stat '\n'
     | list error '\n' { yyerrok;} ;

stat : expr
            { printf( "%d\n", $1 ); }
     | LETTER '=' expr
            { regs[$1] = $3; } ;

expr : '(' expr ')'
            { $$ = $2; }
     | expr '+' expr
            { $$ = $1 + $3;}
     | expr '-' expr
            { $$ = $1 - $3; }
     | expr '*' expr
            { $$ = $1 * $3; }
     | expr '/' expr
            { $$ = $1 / $3; }
     | expr '%' expr
            { $$ = $1 % $3; }
     | expr '&' expr
            { $$ = $1 & $3; }
     | expr '|' expr
            { $$ = $1 | $3; }
     | '-' expr %prec UMINUS
            { $$ = - $2; }
     | LETTER
            { $$ = regs[$1]; }
     | number ;

number : DIGIT
            { $$ = $1; base = ($1==0) ? 8 : 10; }
       | number DIGIT
            { $$ = base * $1 + $2; } ;

%% /* Sección de funciones */
     ...
```

En la definición gramatical, cada uno de los símbolos *no terminales* que aparecen deben estar definidos en alguna de las reglas existentes. También se debe especificar qué símbolos son *terminales*, cuáles son *no terminales* y los tipos de datos de estos símbolos.

Cada símbolo no terminal puede dar lugar a varias producciones, obteniéndose así una definición diferente de ese no terminal. Estas producciones, al igual que en

la notación BNF se separan por símbolos "|". Para conocer cuando se ha puesto la última definición de una producción se incluye al final el carácter punto y coma ";".

Al lado de cada una de las producciones y entre llaves, está incluida la semántica asociada a dicha producción que se ejecutará cuando el analizador sintáctico la haya reconocido. Este código estará escrito en lenguaje C utilizando los símbolos $$, $1, ... en las acciones para hacer referencia al contenido semántico de los elementos *terminales* y *no terminales* incluidos en la producción.

Para completar la explicación de la especificación gramatical, es necesario comentar las declaraciones que hay que incluir previas a la aparición de las reglas de la gramática.

```
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* precedencia para el menos unario */
```

La palabra clave *%token* denota que `DIGIT` y `LETTER` son símbolos *terminales*. Entre *%token* y `DIGIT` se podría haber colocado el nombre de un tipo de dato que se asocia a ese terminal. En la misma línea es posible colocar varios *terminales* siempre que el tipo que tengan, si es que se ha especificado alguno, sea el mismo. De igual forma se permite colocar varias líneas aunque cada terminal sólo puede tener una definición.

De forma análoga a los símbolos *terminales*, existen líneas equivalentes, comenzando con la palabra clave *%type*, para los símbolos *no terminales*. A diferencia de los símbolos anteriores, el que los símbolos *no terminales* aparezcan definidos es opcional y sólo es recomendable cuando se necesita que el analizador considere que el símbolo no terminal es de un tipo de dato determinado.

Las líneas que comienzan por *%left*, *%right* o *%nonassoc*, hacen referencia a la precedencia y asociatividad de los operadores. Es necesario trabajar con gramáticas no ambiguas, porque si es posible reconocer una sentencia mediante distintas secuencias de reglas gramaticales, se producirán diferentes resultados semánticos dependiendo de qué acciones y en qué orden se ejecuten. Por ejemplo, en una expresión como 2 + 2 * 4, se puede evaluar reduciendo primero la suma y después el producto produciéndose como resultado el valor 16. Si se evalúa reduciendo primero el producto y después la suma se obtiene como resultado el valor 10. Los resultados son diferentes porque en el segundo caso, se ha considerado que el operador multiplicación tiene mayor precedencia que el de la suma.

La precedencia y asociatividad es usada por YACC para resolver los posibles conflictos con los que se puede encontrar a la hora de construir el analizador. Formalmente se resuelven de la siguiente manera [Joh75]:

1. La precedencia y asociatividad asociada a cada regla de la gramática se corresponde a la del último token o literal contenido en la regla.

2. Cuando se produce un conflicto reduce/reduce o shift/reduce y el símbolo de entrada no tiene asignado una precedencia y asociatividad, YACC invoca dos reglas para tratar de soluciona la ambigüedad por defecto.

   - En un conflicto shift/reduce, por defecto se elige cargar (shift).

   - En un conflicto reduce/reduce, por defecto se elige reducir (reduce) por la regla de la gramática que aparece antes siguiendo la secuencia de entrada.

3. Si existe un conflicto shift/reduce y tanto la regla de la gramática como el símbolo de entrada tienen precedencia y asociatividad asociada, entonces se resuelve el conflicto a favor de la acción (shift o reduce) asociada con la precedencia más alta. Si poseen la misma precedencia, la elección se basa en la asociatividad: si es de izquierda a derecha se elige reducir, si es de derecha a izquierda se elige cargar y si no es asociativo se producirá un error.

4.1.3 La salida de YACC

A partir del fichero de especificaciones de entrada, YACC genera un analizador sintáctico ascendente. Este analizador viene determinado por un complejo fichero en lenguaje C cuyo contenido es difícil de leer y su comportamiento una vez compilado es difícil de depurar. Forman parte del fichero de salida el analizador y las funciones y declaraciones en C especificadas en el fichero de entrada.

Este fichero con código en lenguaje C, una vez compilado con éxito, dará lugar a un ejecutable con el que podremos empezar a efectuar las primeras pruebas para verificar su comportamiento.

El propósito de YACC se reduce de manera exclusiva a la generación de analizadores sintácticos por lo que si entre las funciones que hemos incluido en el fichero de especificaciones no se encuentra una función específica, conocida como yylex() que se encargue de hacer las veces de analizador léxico, el procesador de lenguaje generado por la herramienta no funcionará. Esta función, yylex(), puede ser creada directamente de forma manual por el usuario si la especificación léxica es sencilla, o se puede delegar en generadores de analizadores léxicos específicos.

4.1.4 Tratamiento de errores

El tratamiento de errores con YACC se podría calificar de sencillo y pobre. En general, cuando el analizador encuentra un error, se llama a la rutina *yyerror* para

su tratamiento y finaliza el proceso. No obstante también incorpora mecanismos que pueden hacer que el analizador se recupere de los errores y sea capaz de seguir adelante con el proceso.

Para permitir al usuario algún control en el tratamiento de errores dispone de un token llamado *error* que está reservado únicamente para el manejo de errores. Puede ser usado en la reglas de la gramática y se debe colocar donde se espera pueda ocurrir algún error del que sea posible su recuperación ejecutándose las acciones asociadas para dejar el proceso de análisis en estado estable.

Estas estrategias de recuperación de errores se basan en suposiciones, el programador las utilizará en aquellos sitios en que "*supone*" que se pueden producir errores. Si luego las cosas no suceden de esta forma, un error podría llevar a otro. Para solucionar el problema de los errores en cascada, después de detectar un error YACC se queda en el estado *error* hasta que tres token hayan sido leídos y procesados de forma satisfactoria. Si ocurre un error cuando el analizador se encuentra en el estado *error*, no se muestra ningún mensaje y el token de entrada es ignorado.

También es posible utilizar la macro *yyerrok* para informar al analizador de que el error ha sido completamente superado y hacer que vuelvan a salir mensajes de error, aunque aún no se hayan cargado con éxito tres tokens. Después de la recuperación de un error, el componente léxico actual es nuevamente analizado, si ya no es válido, se puede utilizar la macro *yyclearin*, que se ocupa de que sea desechado. Tanto *yyerrok* como *yyclearin* son sentencia en el lenguaje C válidas, pudiendo ser utilizadas en las acciones de las reglas de producción.

4.1.5 Crítica

4.1.5.1 Necesidad de mejorar el generador de analizadores léxicos

Lex soporta solamente caracteres de 8 bits. Esto supone un problema para lenguajes como Java que ha adoptado como conjunto de caracteres nativo el sistema de codificación Unicode.

El tratamiento que Lex realiza con las macros es análogo al que se realiza en el lenguaje C. Esto significa que son textualmente sustituidas en las expresiones regulares, lo que puede dar lugar a errores difíciles de localizar. Por ejemplo, si una macro M es definida como a|b, entonces la expresión regular aMb será interpretada como aa|bb ≡ (aa)|(bb) y no como a(a|b)b que podría ser lo deseado.

4.1.5.2 Falta de integración entre las dos herramientas

Las herramientas Lex y YACC no han sido especialmente diseñadas para trabajar juntas. Para que el procesador de lenguaje pueda ser generado, es necesario proveer al analizador sintáctico los componentes léxicos del lenguaje de entrada. Es tarea del programador establecer los enlaces necesarios con el código generado por las dos herramientas.

4.1.5.3 Mezcla de código y especificación gramatical

Las acciones semánticas asociadas a las producciones de los *no terminales* de la gramática son difíciles de depurar. Puesto que este código pasa directamente al código del analizador generado por YACC, es necesario comprender el funcionamiento de un analizador ascendente para saber donde falla algo. Esto requiere un nivel de abstracción bastante elevado y hace que no sea adecuado para muchos usuarios de la herramienta.

4.1.5.4 Ciclo de desarrollo largo

El hecho de que las acciones estén mezcladas con la especificación de la gramática supone un grave problema desde el punto de vista de la ingeniería del software. Esto significa que el tamaño de las especificaciones resultantes son a menudo muy grandes y contenidas en un único fichero.

Por otra parte la duplicación del código de acción en la especificación y en el programa generado resulta nefasto. Deja al programador la responsabilidad de mantener actualizados el fichero de especificaciones y el programa resultante. El ciclo de depurado del código de las acciones semánticas sigue los siguientes pasos:

1. Escribir o modificar el código de acción en el fichero de especificaciones.
2. Procesar el fichero de especificaciones.
3. Compilar y ejecutar el fichero generado para localizar posibles errores.
4. Localizar los errores en el programa.
5. Editar nuevamente el fichero de especificaciones.
6. Volver al paso 1.

Se incrementa la dificultad del trabajo de depuración puesto que los errores se cometen en el fichero de especificación pero son sólo visibles en el analizador generado. Acortar este ciclo resolviendo los errores únicamente en el programa generado, implica tener un fichero de especificaciones no actualizado con los últimos cambios realizados.

4.1.5.5 No permite generar ASTs

No genera árboles sintácticos abstractos ASTs (*Abstract Syntax Trees*) con lo cual no es fácil conseguir compiladores que realicen sus tareas en varias pasadas. Los analizadores generados por YACC, en general, realizan sus tareas leyendo el fichero fuente una sola vez. En determinadas circunstancias, dependiendo de la

complejidad del lenguaje, es necesario que los analizadores operen a través de varias pasadas para recoger diversa información del fichero fuente.

4.1.6 Características interesantes

4.1.6.1 Eficiencia

Los analizadores generados por YACC son muy eficientes desde el punto de vista del rendimiento y del tamaño del código del analizador generado. Incluso se podría decir que más que los construidos de forma manual en general.

Por otra parte, el conjunto de lenguajes que pueden ser reconocidos por los analizadores ascendentes, es mayor que los que pueden reconocer los analizadores descendentes.

4.2 LEMON

4.2.1 Características generales

LEMON [Lem01] es un generador de analizadores sintácticos escritos en leguaje C (aunque también admite código C++) y usa para ello tablas de análisis LALR(1). Es similar a otras herramientas como YACC o BISON pero aporta algunas diferencias. Utiliza una sintaxis gramatical diferente que permite cometer menos errores, genera analizadores que son más rápidos que los generados por esos programas e incluye el concepto de un símbolo destructor.

El objetivo fundamental de esta herramienta es traducir una gramática libre de contexto para un lenguaje determinado a código C. Este código resultante será el que implemente al analizador para ese lenguaje.

LEMON tiene **dos entradas**, la especificación gramatical y un fichero de plantillas para el analizador sintáctico. Solamente es obligatorio que el usuario describa la gramática, puesto que se crea por defecto un fichero de plantillas, aunque también puede proporcionar su propio fichero de plantillas si así lo desea.

En **la salida** se obtendrán dos o tres ficheros, dependiendo de las opciones en la línea de comandos. Un programa en código C que implementa al analizador sintáctico, un fichero cabecera asociando un entero para cada no terminal presente en la gramática, y un fichero de información que describe los estados del autómata del analizador generado.

4.2.2 Características relativas a su funcionamiento

LEMON no crea un programa analizador completo sino que genera unas subrutinas que deben ser invocadas por el programador de una forma apropiada para producir el sistema completo.

Ejemplo 6.2 *El siguiente ejemplo describe el núcleo del código que debe proporcionar el usuario y la interfaz de las subrutinas generadas.*

```
ParseFile() {
    PParser = ParseAlloc( malloc );
    While( GetNextToken(pTokenizer, &hTokenId, &sToken) ) {
            Parse(pParser, hTokenId, sToken);
     }
     Parse (pParser, 0, sToken);
     ParseFree (pParser, free);
}
```

Lo primero que debe hacer un programa para usar un analizador sintáctico generado por esta herramienta es crear el analizador (el *parser)*. Las estructuras de datos usadas para representar a este analizador no son visibles desde el programa.

A continuación, se obtendrían los *tokens* del fichero de entrada por medio de la función `GetNextToken()`.Cuando se detecta el final del fichero de entrada, se informa al analizador y se libera la memoria.

La herramienta mezcla las acciones semánticas, mediante bloques de código C, con la especificación gramatical. Estos bloques deben colocarse al final de las reglas de la gramática y son ejecutados cuando el analizador realiza las reducciones correspondientes.

El acceso a los valores semánticos de los símbolos de la gramática, no se realiza según la posición que ocupen en la regla sino a través de los nombres simbólicos asignados. Esta notación, permite localizar de forma más sencilla errores de omisión en los bloques de código y facilita el uso de destructores para liberar la memoria ocupada por los valores asociados a los símbolos (*terminales* y *no terminales*).

4.2.3 Tratamiento de errores

Cuando el analizador generado por LEMON encuentra un error de sintaxis, invoca al código especificado por la directiva `%syntax_error`, si existe. La estrategia de recuperación de errores consiste en extraer símbolos de la pila hasta que el sistema pasa a un estado en el que está permitido cargar el símbolo especial *error,* a partir del cual puede seguir analizando. Este código no será llamado nuevamente hasta que al menos se hayan cargado con éxito tres nuevos *tokens*.

4.2.4 Crítica

LEMON intenta mejorar desde el punto de vista del usuario final, la forma de trabajar de otras herramientas análogas como YACC o BISON, usando una sintaxis gramatical ligeramente diferente para reducir los errores cometidos en el código de las acciones semánticas, eliminando el uso de variables globales para pasar la información entre el analizador léxico y sintáctico o permitiendo que varios analizadores sintácticos puedan ser ejecutados simultáneamente, pero aún mantiene la misma estructura y modo de operar de YACC, por lo tanto, es aplicable la misma crítica vista en el apartado 4.1.5.

4.3 ACCENT

4.3.1 Características generales

ACCENT [Acc01] es una herramienta que sirve para generar procesadores de lenguajes a partir de la descripción de la gramática de un determinado lenguaje, sin imponer restricciones sobre su especificación. Es decir, puede procesar una gramática libre de contexto sin que su diseño se vea influenciado por la técnica de implementación (LALR o LL) del analizador sintáctico.

El formato de la especificación gramatical que utiliza es similar al de YACC aunque con ligeras diferencias. Admite la notación EBNF, usa nombres simbólicos para referirse a los valores de los símbolos, y permite operar con gramáticas ambiguas, estableciendo para ello una notación que resuelve las ambigüedades a nivel de abstracción de la gramática, sin quedar reflejado en el algoritmo de análisis.

La representación de los símbolos *terminales* no están definidos en la especificación gramatical. Utiliza la función `yylex()` generada por *Lex* para obtener la secuencia de *tokens* del fichero de entrada.

4.3.2 Características relativas a su funcionamiento

Las ambigüedades se detectan en tiempo de ejecución y para resolverlas, se dividen en dos grupos: ambigüedades entre alternativas y ambigüedades dentro de las alternativas.

En ambos caso, la herramienta visualiza un análisis detallado y muestra una indicación de cómo resolver la ambigüedad mediante una notación concreta. Para el primer grupo, la notación `%prio number` se usa para dar a una alternativa cierta prioridad. Siempre se selecciona la alternativa con prioridad más alta. Para el segundo grupo, la notación `%long` indica que se quiere reconocer la cadena más larga y con la notación `%short` la cadena más corta.

Las acciones semánticas (bloques de código en lenguaje C) pueden estar situadas en cualquier posición dentro de la gramática y serán ejecutadas cuando la alternativa particular sea procesada. Este código pasa directamente al programa generado.

Arquitectura

Es necesario seguir varios pasos hasta la obtención del procesador de lenguaje. En la *figura 4.2* se muestran los ficheros que deben ser generados y suministrados por el usuario.

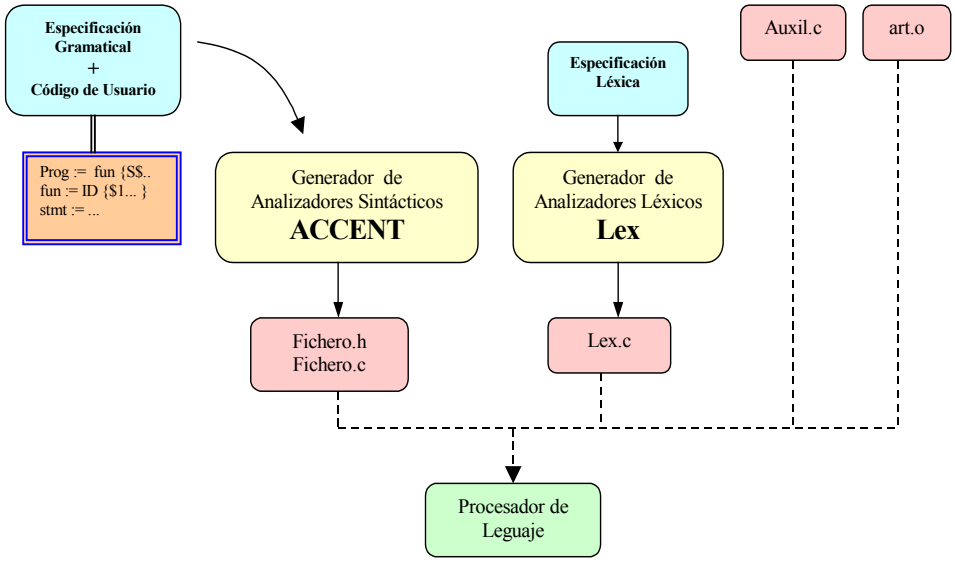


Figura 4.2: Arquitectura de ACCENT

Su arquitectura no difiere de la de YACC. El usuario tiene que proporcionar algunas funciones auxiliares como `main()`, `yyerror()` que deben estar incluidas en el fichero `auxil.c` y el fichero `art.o` que contiene el código necesario para crear un analizador operativo.

Técnica de implementación del analizador

ACCENT utilizan dos tipos de enfoques para solucionar los problemas de análisis, cuando es posible aplicar más de una alternativa:

- **Análisis exhaustivo** que examina todas las posibilidades en paralelo. Lo utiliza para conseguir generalidad.

- **Análisis predictivo** que inspecciona el siguiente símbolo de entrada y lo utiliza para elegir la alternativa correcta, mejorando así su eficiencia.

Para el análisis exhaustivo utiliza el *algoritmo de Earley* [Ear70], y con el análisis predictivo usa los conjuntos de símbolos directores para excluir una alternativa que no sería viable.

4.3.3 Crítica

Se trata de una herramienta que sigue manteniendo la misma estructura que los generadores clásicos y por lo tanto sus defectos son: falta de integración de los analizadores léxicos y sintácticos, las acciones semánticas se mezclan con las reglas gramaticales, no permite generar ASTs, el tratamiento y recuperación de errores es muy pobre y mantiene un ciclo de desarrollo largo.

Es destacable el hecho de que permita una descripción gramatical más cómoda, que abarca al conjunto de las gramáticas libres de contexto, pero esta generalidad tiene un precio, los procesadores de lenguajes obtenidos son menos eficientes que los generados por YACC.

4.3.4 Características interesantes

La especificación de la gramática no está influenciada por la técnica de implementación del analizador sintáctico, permitiendo además el uso de gramáticas ambiguas.

4.4 COCKTAIL

4.4.1 Características generales

El sistema Cocktail [Coc93] ha sido desarrollado hasta 1993 por el centro de investigación nacional de Alemania. Engloba a un conjunto de herramientas para generar parte de las fases de un compilador. Estas herramientas son:

- **Rex** (Regular EXpresión tool). Es un generador de analizadores léxicos cuya especificación está basada en expresiones regulares y acciones semánticas escritas en lenguaje C o Modula-2.

- **Lalr**. Es un generador de analizadores sintácticos basado en tablas de análisis LALR(1). Acepta gramáticas escritas en notación BNF junto con acciones semánticas en lenguaje C o Modula-2. Cuando se producen conflictos LR, además de la información sobre el estado interno, proporciona un árbol de derivación que es más útil para analizar el problema. Los analizadores generados incluyen tratamiento y recuperación de errores.

- **Ell**. Es un generador de analizadores sintácticos que acepta gramáticas que cumplan la condición LL(1) junto con acciones semánticas en lenguaje C o Modula-2. Estos analizadores son implementados siguiendo el método recursivo descendente e incluyen tratamiento y recuperación de errores.

- **Ast**. Es un generador de árboles sintácticos abstractos. La información se almacena como una secuencia de registros enlazados, generando también tipos de datos abstractos para su recorrido.

- **Ag.** Permite procesar gramáticas atribuidas.

4.4.2 Crítica

Aunque existe un estudio comparativo de estas herramientas con Lex/YACC en el que se muestra una mejora en eficiencia y tratamiento de errores, el sistema es muy similar y mantiene la mayoría de los problemas encontrados en estos generadores tradicionales: falta de integración de los analizadores léxicos y sintácticos, falta de un mecanismo de tratamiento y recuperación de errores robusto, mezcla de código y especificación gramatical y ciclo de desarrollo largo.

4.5 ANTLR

El sistema PCCTS (*Purdue Compiler Construction Tool Set*) [PDC91, Par97] fue escrito inicialmente en el lenguaje C++ para generar compiladores escritos también en C++. Posteriormente fue portado al lenguaje Java pasándose a llamar ANTLR 2.x.x [Ins01]

ANTLR es sólo una de las tres partes del sistema que está compuesto por:

- **Antlr** *(Another Tool for Language Recognition)*. Una herramienta para construir procesadores de lenguajes a partir de una descripción gramatical.
- **DLG** *(DFA-based Lexical analyzer Generator)*. Es el generador de analizadores léxicos basados en autómatas finitos deterministas.
- **SORCERER.** Herramienta para construir árboles sintácticos abstractos (ASTs).

4.5.1 Características generales

Es una herramienta creada para el mismo propósito que Lex/YACC. Ambas tienen bastantes cosas en común, aunque se podría decir que ANTLR es más fácil de usar y se integra mejor con lenguajes orientados a objeto como C++ y Java.

La facilidad de uso de ANTLR se fundamenta en que utiliza analizadores descendentes recursivos LL(k) cuyo funcionamiento interno es mucho más predecible y fácil de depurar que las tablas de análisis LALR(1).

Las características más importantes que aporta esta herramienta son:

1. Predicados semánticos.
2. Predicados sintácticos
3. Construcción de árboles sintácticos abstractos.

Predicados semánticos

Un *predicado semántico*, especifica una condición que debe ser cumplida en tiempo de ejecución, antes de que el analizador pueda continuar. Existen dos tipos de predicados semánticos, los que realizan la función de validación lanzando una excepción si no se cumplen las condiciones necesarias cuando se analiza una producción, y los que resuelven problemas de ambigüedades. Los predicados semánticos se representan mediante acciones semánticas.

Ejemplo 6.2 *El siguiente ejemplo muestra el uso de un predicado semántico para validar si un identificador es semánticamente un tipo "nombre".*

```
decl: "var" ID ":" t:ID
       { isTypeName(t.getText()) }?
    ;
```

A través del predicado semántico se genera una excepción si no se cumple la condición.

Ejemplo 6.3 *El siguiente ejemplo muestra el uso de un predicado semántico para distinguir entre una declaración de variables y una sentencia de asignación.*

```
Statement:
          {isTypeName(LT(1))}? ID ID ";"  // Declaración
          ID "=" expr ";";                // Asignación
         ;
```

El analizador generado realizará la siguiente interpretación:

```
if(LA(1)==ID && isTypeName(LT(1)))
{
  match alternative one
}
else if (LA(1)==ID)
{
  match alternative two
}
else error
```

Se demuestra así como se puede influir en el proceso de análisis con información semántica mediante los predicados semánticos.

Predicados sintácticos

Los *predicados sintácticos* son usados durante el análisis cuando no es posible resolver la ambigüedad producida en dos o más alternativas de una producción. Un predicado sintáctico ejecuta las llamadas de función del predicado, pero no ejecuta el código de acción asociado a la regla. Si se satisface el predicado, se elige la alternativa correspondiente y se ejecuta el código, sino se procesa la siguiente alternativa (o predicado) que lo cumpla. Los predicados sintácticos son implementados usando las excepciones del lenguaje de implementación.

Ejemplo 6.4 *El siguiente ejemplo muestra una regla que distingue entre una lista y una lista asignada a otra lista.*

```
stat: ( list "=" )=> list "=" list
| list
;
```

Si se encuentra una lista seguida por un operador de asignación, entonces se elige la primera producción, sino se intenta el análisis eligiendo la segunda alternativa. Los predicados sintácticos aportan una forma de realizar un análisis con retroceso (*backtraking*) selectivo en el que no se ejecutan las acciones durante el proceso de evaluación.

Árboles sintácticos abstractos

La construcción de ***árboles sintácticos abstractos*** (ASTs) se realiza aumentando la especificación gramatical con operadores, reglas y acciones. ANTLR también permite especificar la estructura gramatical de los ASTs, mediante una colección de reglas EBNF mezcladas con acciones, predicados semánticos y predicados sintácticos, de forma análoga a la herramienta SORCERER cuando se usa de forma separada.

Ejemplo 6.5 *El siguiente ejemplo muestra la construcción de un analizador para una pequeña calculadora de expresiones, generando un AST como representación intermedia y un analizador que recorre dicha representación intermedia evaluando el resultado.*

```
class CalcParser extends Parser;
options {
buildAST = true;   // usa CommonAST por defecto
}
expr: mexpr (PLUS^ mexpr)* SEMI!
    ;                     //!Operador para indicar que el
AST                       // token será ignorado para el AST
mexpr: atom (STAR^ atom)*
   ;
atom:  INT
  ;
```

Los tokens `PLUS` y `STAR` representan operadores y se indica que serán nodos raíz del árbol que se va a generar mediante el carácter '`^`'. El símbolo '`!`' se usa para indicar que el token `SEMI`, que representa el carácter '`;`', no se incluirá en el AST.

Para construir un analizador que permita recorrer los árboles generados a partir de las expresiones de entrada, es necesario especificar la estructura gramatical del AST.

```
class CalcTreeWalker extends TreeParser;
expr : #(PLUS expr expr)
     | #(STAR expr expr)
     | INT
     ;
```

Una vez especificada la estructura, es necesario añadir acciones que permitan evaluar el resultado de las expresiones.

```
class CalcTreeWalker extends TreeParser;
expr returns [float r]
{
float a,b;
r=0;
}
: #(PLUS a=expr b=expr) {r = a+b;}
| #(STAR a=expr b=expr) {r = a*b;}
| i:INT       {r = (float)Integer.parseInt(i.getText());}
;
```

Los símbolos de entrada son almacenados como nodos en el árbol, pero la estructura es codificada teniendo en cuenta la relación de esos nodos.

Por último, el código necesario para iniciar el proceso es el siguiente:

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;

class Calc {
    public static void main(String[] args) {
        try {
            CalcLexer lexer =
                new CalcLexer(new
DataInputStream(System.in));
            CalcParser parser = new CalcParser(lexer);
                     // Analiza la expresión de la entrada
            parser.expr();
            CommonAST t = (CommonAST)parser.getAST();
                    // Imprime el AST usando la notación LISP
            System.out.println(t.toStringList());
            CalcTreeWalker walker = new CalcTreeWalker();
                   // Recorre el árbol creado por el
analizador
            int r = walker.expr(t);
            System.out.println("value is "+r);
        } catch(Exception e) {
            System.err.println("exception: "+e);
        }
    }
}
```

4.5.2 La entrada de Antlr

La descripción de entrada en Antlr consiste en una colección de reglas léxicas y sintácticas describiendo el lenguaje que va a ser reconocido, y un conjunto de acciones semánticas definidas por el usuario para realizar el tratamiento correspondiente cuando las instrucciones de entrada son reconocidas. Formalmente el fichero de especificación contiene la siguiente información:

```
Sección de cabecera
Definición de reglas léxicas
Definición de reglas sintácticas
Definición de reglas para generar ASTs (Esta sección es
opcional)
Acciones
```

Sección de cabecera

Es la primera sección del fichero y contiene el código fuente que debe estar colocado en la salida, antes que el código generado por Antlr. Sirve por ejemplo, para especificar las librerías y las clases importadas en el lenguaje Java.

Definición de reglas léxicas

Las reglas léxicas deben comenzar con una letra mayúscula, pueden especificar argumentos, devolver valores y usar variables locales. Son procesadas de la misma forma que las reglas sintácticas.

Definición de reglas sintácticas

Las reglas sintácticas deben comenzar con una letra minúscula y siguen un formato análogo a la notación utilizada en YACC, aunque en este caso se permite la notación EBNF.

Definición de reglas para generar ASTs

En la definición de reglas que generan un AST, es necesario utilizar una sintaxis adicional y especial para especificar la estructura del árbol generado. No obstante, las especificaciones en los tres tipos de definiciones de reglas son similares.

Acciones

Son bloques de código fuente ejecutados durante el análisis, después de que los elementos de una producción hayan sido reconocidos y antes de reconocer el siguiente elemento situado a su derecha. Si la acción ocupa el primer elemento de una producción, se ejecuta antes de procesar la regla.

Ejemplo 6.6 *El siguiente ejemplo muestra los tres tipos de especificaciones gramaticales (léxicas, sintácticas y AST) necesarias para analizar expresiones aritméticas sencillas.*

```
class CalcParser extends Parser;
options {
buildAST = true; // usa CommonAST por defecto
}
expr
: mexpr (PLUS^ mexpr)* SEMI!
;
mexpr
: atom (STAR^ atom)*
;
atom:  INT
;

class CalcLexer extends Lexer;
WS :  (' '
   |  '\t'
   |  '\n'
   |  '\r')
      { _ttype = Token.SKIP; }
```

```
   ;
LPAREN:'('
   ;
RPAREN:')'
   ;
STAR:  '*'
   ;
PLUS:  '+'
   ;
SEMI:  ';'
   ;
protected
DIGIT
: '0'..'9'
;
INT  :  (DIGIT)+
;

class CalcTreeWalker extends TreeParser;
expr returns [float r]
{
float a,b;
r=0;
}
:  #(PLUS a=expr b=expr) {r = a+b;}
|  #(STAR a=expr b=expr) {r = a*b;}
|  i:INT       {r = (float)Integer.parseInt(i.getText());}
;
```

Las especificaciones en Antlr siguen un estilo similar a las de YACC. Incluyen la gramática del lenguaje que debe reconocer el analizador que se va a generar, y al lado de cada una de las producciones de la gramática, aparece el código de las acciones semánticas en el lenguaje de implementación utilizado C/C++ o Java. También se incluye la especificación del analizador léxico, definiéndose los tokens necesarios para ser utilizados por el analizador sintáctico.

El usuario deberá proporcionar, además del fichero de especificaciones, un objeto con un método `main()` que se encargue de inicializar una instancia del analizador léxico y del analizador sintáctico.

4.5.3 La salida de Antlr

Antlr genera tres tipos de ficheros `MY_PARSER`, `MY_LEXER`, y `MY_TREE_PARSER`. Son nombres genéricos ya que en realidad se asignan los nombres de las clases contenidas en el fichero de especificaciones.

El **fichero `MY_LEXER`** contiene la clase que describe las reglas del analizador léxico. Su función consiste en dividir la secuencia de caracteres de entrada en tokens para ser suministrados al analizador sintáctico.

El **fichero `MY_PARSER`** contiene la clase con los métodos de las reglas del analizador sintáctico. Esta clase generada contiene un método para cada una de las reglas de la gramática y es subclase de `LlkParser`, que es la clase padre de todos los analizadores sintácticos que genera Antlr. Como realiza un análisis

descendente, comienza el proceso siempre desde un símbolo inicial y va derivando hasta encontrar los tokens que le proporciona el analizador léxico.

El **fichero** `MY_TREE_PARSER` contiene la clase que representa el AST (*Abstract Syntax Tree*). Los ASTs permiten construir un árbol de derivación genérico que podrá ser aplicado a un programa concreto expresado en un lenguaje de programación determinado. Sobre este AST se podrá aplicar una o varias clases (*tree-walkers*) que se encarguen de recorrer el árbol.

4.5.4 Tratamiento de errores

El mecanismo de tratamiento y recuperación de errores en Antlr es bastante bueno. Se establece sobre la base del sistema de tratamiento de excepciones presente en Java y en otros lenguajes de programación como C++.

Soporta mecanismos de tratamiento y recuperación de errores a través de los bloques de código `try/catch` que especifica el usuario al lado de determinados elementos de interés de la gramática. Si en el proceso de derivación se produce un error, se lanza una excepción que es tratada por alguna de las acciones semánticas asociadas con la sección gramatical que invocó al método en donde se produjo el error.

Si no se incluye un tratamiento de errores explícito, las excepciones serán propagadas y tratadas de manera simple, con el código que proporciona por defecto la herramienta.

Los analizadores sintácticos generados pueden lanzar excepciones para señalar determinados problemas o errores encontrados. El mecanismo de tratamiento de errores se basa en la siguiente jerarquía de excepciones de la *figura 4.3*.

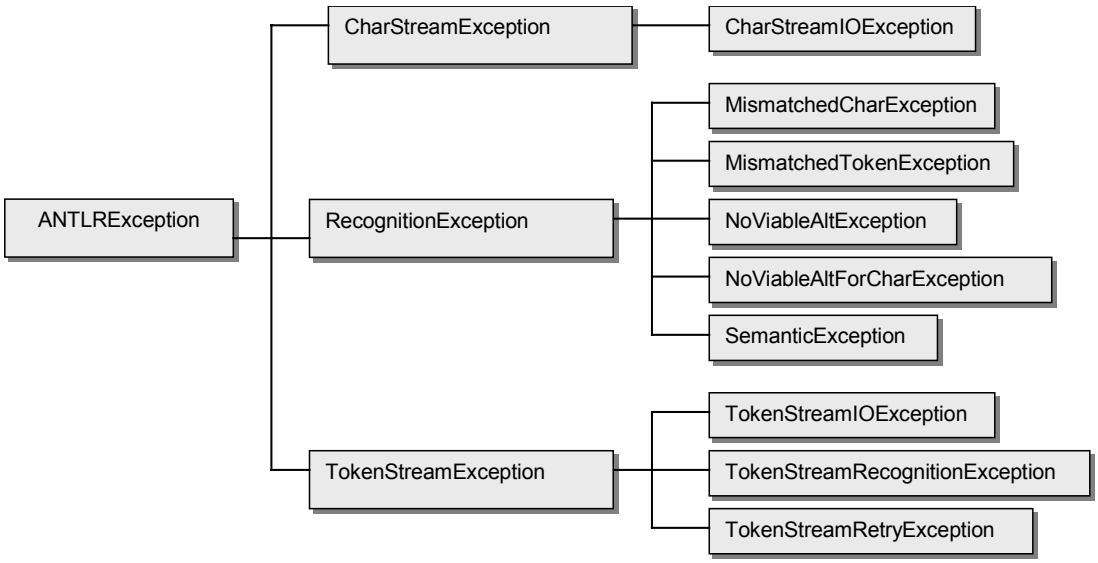


Figura 4.3: Jerarquía de excepciones en Antlr

- **ANTLRException**. Es la raíz de la jerarquía de excepciones.
- **CharStreamException**. Indica que hubo problemas leyendo del fichero que contiene la especificación de entrada.
- **RecognitionException**. Señala un error genérico cada vez que hay un problema reconociendo la entrada. Todas las reglas gramaticales lanzan esta excepción
- **TokenStreamException.** Indica que ocurre algún problema en la formación de los tokens.
- **CharStreamIOException**. Indica problemas de entrada/salida.
- **MismatchedCharException**. Señala que el carácter leído no es el esperado.
- **MismatchedTokenException**. Señala que el símbolo terminal no es el esperado.
- **NoViableAltException**. Indica que el analizador sintáctico no puede elegir entre las alternativas de una regla, debido a que el token leído, no se incluye en el comienzo de ninguna de ellas.
- **NoViableAltForCharException**. Indica que el analizador léxico no puede elegir entre las alternativas de una regla, debido a que el carácter leído, no se incluye en el comienzo de ninguna de ellas.
- **SemanticException**. Esta excepción se lanza automáticamente cuando se encuentran problemas con los predicados semánticos.
- **TokenStreamIOException**. Indica que se encontraron problemas durante la generación de una secuencia de tokens en la entrada/salida.
- **TokenStreamRecognitionException**. Señala un error indicando que ocurre algún problema en la formación de los tokens.
- **TokenStreamRetryException**. Señala que se abandona el reconocimiento del token actual y se intenta conseguir otro nuevamente.

4.5.5 Crítica

4.5.5.1 Eficiencia

Los predicados sintácticos resultan ser muy caros en tiempo de ejecución y las verificaciones semánticas deben realizarse durante todo el proceso de análisis para que puedan ser utilizados los predicados semánticos. Además, los dos tipos de predicados forman parte de la especificación de la gramática añadiéndole cierta complejidad a su descripción.

La flexibilidad de los predicados puede llegar a ser muy costosa en cuanto a rendimiento si no son usados de forma correcta por el programador.

4.5.5.2 Mezcla de código y especificación gramatical

Como ya se pudo observar anteriormente en la descripción de Lex/YACC, el hecho de que las acciones estén mezcladas con la especificación de la gramática supone un grave problema desde el punto de vista de la ingeniería del software.

Por otra parte, con la duplicación del código de acción en la especificación y en el programa generado, deja al programador la responsabilidad de mantener actualizados el fichero de especificaciones y el programa resultante. Esta tarea se podría realizar mejor en un entorno de desarrollo integrado.

4.5.5.3 Ciclo de desarrollo largo

El ciclo de desarrollo y depurado del código de las acciones semánticas sigue los mismos pasos vistos en la sección 4.1.5.4.

Se incrementa también la dificultad del trabajo de depuración puesto que los errores se cometen en el fichero de especificación pero son sólo visibles en el analizador generado. Acortar este ciclo resolviendo los errores únicamente en el programa generado implica tener un fichero de especificaciones no actualizado con los últimos cambios realizados.

4.5.5.4 Integridad de los ASTs generados

La integridad de la información en los ASTs se deja en manos del programador. Esto ocasiona graves inconvenientes ya que los errores que se puedan producir después de realizar una transformación sobre un AST, no originan ningún tipo de aviso y resulta muy difícil localizarlos posteriormente. Los problemas que pueden surgir son comparables a los que se producen en los lenguajes C/C++ cuando se accede fuera de los límites de una matriz.

4.5.6 Características interesantes

4.5.6.1 Integración de los analizadores léxico y sintáctico

Antlr proporciona una buena integración de los analizadores léxico y sintáctico, compartiendo la misma sintaxis a través de su especificación conjunta en el fichero de entrada.

4.5.6.2 Capacidades y modo de análisis

El rango de lenguajes que pueden ser procesados con analizadores descendentes LL(k) es similar a los procesados con analizadores LALR(1). Además el uso de predicados semánticos permite el análisis de gramáticas sensibles al contexto o gramáticas de tipo 1.

Las especificaciones gramaticales de Antlr admiten la notación EBNF y es posible generar ASTs, muy convenientes si se desea construir compiladores de más de una pasada. Por otra parte, debido a la naturaleza de los analizadores descendentes, el código generado por Antlr es más fácil de depurar y comprender que el generado por YACC basado en tablas de análisis LALR(1).

4.6 JavaCC

JavaCC [Jav01b] es un generador de analizadores sintácticos escritos en lenguaje Java. Soporta básicamente las mismas características que Antlr, aunque hay que tener en cuenta que JavaCC es un producto comercial diseñado para ser utilizado con aplicaciones Java, del que no están disponibles los ficheros fuentes, a diferencia de Antlr que es de dominio público.

4.6.1 Características generales

Como características generales de la herramienta se destacan las siguientes:

- Permite generar analizadores léxicos y analizadores sintácticos recursivos descendentes LL(k).

- Integra en la descripción completa del fichero de entrada, la especificación léxica y la especificación sintáctica.

- Incluye otra herramienta denominada JJTree que es un preprocesador de la entrada de JavaCC y cuya función consiste en insertar acciones en determinados lugares de la especificación del fichero para generar árboles sintácticos abstractos (*ASTs*). La salida de JJTree debe ser procesada por JavaCC para generar el analizador sintáctico. Los ASTs son muy utilizados fundamentalmente en analizadores que deben procesar entradas complejas.

4.6.2 La entrada de JavaCC

El formato de entrada en JavaCC, contiene una colección de reglas léxicas y sintácticas para describir el lenguaje que va a ser reconocido y un conjunto de acciones semánticas definidas por el usuario para realizar las tareas correspondientes siempre y cuando las instrucciones de entrada sean reconocidas. Formalmente, la descripción del fichero de especificaciones contiene las siguiente secciones:

```
Lista de opciones
Java Compilation Unit
Producciones de la gramática
```

Lista de opciones

La lista de opciones puede estar o no presente en el fichero de entrada. Permite incluir distintas informaciones mediante un conjunto de identificadores que son inicializados a unos valores concretos.

Java Compilation Unit

Esta sección incluye una JCU (Java compilation unit) delimitada por dos nombres que actúan como marcas, `PARSER_BEGIN` y `PARSER_END`. Contiene como mínimo la declaración de una clase cuyo nombre coincide con el del analizador sintáctico generado. JavaCC no realiza comprobaciones sobre esta unit, por tanto es posible generar ficheros que produzca errores cuando sean compilados.

El analizador generado contiene todo el código de esta unit y la declaración de un método público para cada uno de los *no terminales* de la gramática especificada. Estos métodos serán llamados posteriormente para realizar el proceso de análisis.

Producciones de la gramática

Existen cuatro clases de producciones: las de código Java y notación BNF, usadas para definir la gramática, las que describen las expresiones regulares y las que se usan para introducir declaraciones.

- **Producciones de código Java.** Es una forma de escribir código para algunas producciones que tienen que reconocer una secuencia de tokens difíciles de describir mediante reglas gramaticales.

- **Producciones EBNF.** Son las producciones estándar usadas para la descripción de la gramática. Como cada símbolo no terminal se traduce a un método en el analizador generado, el nombre debe ser el mismo que el del método, y los parámetros y valores de retorno declarados, se utilizan para pasar valores hacia arriba y hacia abajo en el árbol sintáctico. La parte derecha de una producción comienza con un conjunto de declaraciones y código, que es colocado al comienzo del método generado para el símbolo no terminal de la parte izquierda. Cada vez que se usa el símbolo no terminal en el proceso de análisis, se ejecuta el código pero JavaCC no procesa este código y las declaraciones incluidas en esta parte, es el compilador de Java el que detecta posibles errores.

- **Producciones de expresiones regulares.** Se usan para definir las entidades léxicas que serán procesadas por el analizador léxico generado. Cada producción proporciona el estado léxico para el cual se aplica, la clase de expresión regular y las especificaciones que describen de forma detallada las entidades léxicas de dicha expresión regular junto con las acciones correspondientes.

- **Producciones para incluir declaraciones.** Contienen bloques de código en Java con declaraciones y sentencias accesibles desde el código de las producciones léxicas.

Ejemplo 6.7 *El siguiente ejemplo muestra las especificaciones necesarias para analizar expresiones aritméticas sencillas, implementando una pequeña calculadora.*

```
PARSER_BEGIN(Calc0)  // define la clase parser
 public class Calc0 {
   public static void main (String args []) {
    Calc0 parser = new Calc0(System.in);
    for (;;)
      try {
       if (parser.expr() == -1)
         System.exit(0);
      } catch (Exception e) {
       e.printStackTrace(); System.exit(1);
      }
   }
 }
PARSER_END(Calc0)

SKIP:    // define la entrada que deber ser ignorada
 { " " | "\r" | "\t"
 }
TOKEN:    // define nombres de tokens
 { < EOL: "\n" >
 | < CONSTANT: ( <DIGIT> )+ >
 | < #DIGIT: ["0" - "9"] >
 }

int expr():   // expr: sum \n
 {}    // -1 para eof, 0 para eol
 { sum() <EOL> { return 1; }
 | <EOL> { return 0; }
 | <EOF> { return -1; }
 }
void sum():   // sum: product { +- product }
 {}
 { product() ( ( "+" | "-" ) product() )*
 }
void product():   // product: term { *%/ term }
 {}
 { term() ( ( "*" | "%" | "/" ) term() )*
 }
void term():   // term: +term | -term | (sum) | number
 {}
 { "+" term()
 | "-" term()
 | "(" sum() ")"
 | <CONSTANT>
 }
```

En este ejemplo, no se ha incluido una lista de opciones. Se comienza con la descripción del código que debe proveer el usuario para inicializar y terminar el proceso de análisis, es decir, la clase que contiene al analizador sintáctico y que generará JavaCC a partir de la especificación del fichero de entrada.

En la sección de producciones, se describen dos clases de expresiones regulares. `SKIP`, contiene las expresiones que se corresponden con las partes de la entrada que se pueden ignorar en el proceso de análisis, como por ejemplo, comentarios. `TOKEN`, recoge las expresiones asociadas a los tokens de la gramática. A partir de esta información se genera el analizador léxico.

El resto de la especificación corresponde a la definición de la gramática. Para cada símbolo no terminal de la gramática se colocan las distintas producciones y al lado de estas, el código de acción. Se puede observar también que los símbolos *no terminales* se definen como métodos de la clase que se va a generar con los parámetros y valores de retorno

Aunque en el ejemplo se ve como todos los componentes léxicos han sido definidos previamente en la sección correspondiente, también se pueden realizar definiciones de este tipo dentro del ámbito de una regla, aunque en este caso dicha definición solo tendrá validez dentro de esa regla.

4.6.3 La salida de JavaCC

La salida de JavaCC incluye las siguientes clases:

- **TokenMgrError**. Es la excepción que se lanza cuando se detectan problemas con el reconocimiento de los tokens. También se encarga de que los mensajes de error presenten suficiente información al usuario. Es una subclase de la clase *Error*.

- **ParseException**. Es la excepción que se lanza cuando el analizador sintáctico encuentra algún problema. Es una subclase de la clase *Exception*.

- **Token**. La clase que encapsula los símbolos *terminales* que se encarga de obtener el analizador léxico para suministrárselos al analizador sintáctico.

- **Parser**. La clase del analizador sintáctico principal.

- **ParserTokenManager**. Un gestor de símbolos *terminales* para el analizador contenido en el parser.

- **ParserConstants**. La clase que define las constantes utilizadas por el analizador sintáctico.

La palabra *parser* en estas clases representa un nombre simbólico, puesto que en realidad dicho nombre es elegido por el usuario de la herramienta.

4.6.4 Tratamiento de errores

El tratamiento de errores de JavaCC está basado en el mecanismo de excepciones del lenguaje Java. No existe una jerarquía de excepciones determinada, sino que cuando se encuentra un error durante el proceso de análisis, se lanza una excepción de tipo *ParserException*.

Esta excepción podrá ser tratada en el lugar donde se produjo o en cualquier otra producción que la pueda anteceder en el proceso de derivación, que en el caso de un analizador descendente comienza en un símbolo inicial y va derivando hasta que se encuentran en la entrada los símbolos *terminales*.

El usuario podrá utilizar otras excepciones además de *ParserException* siempre y cuando las haya definido previamente. El código asociado al tratamiento de errores se incluye en la propia especificación de JavaCC, junto al código de acción y al lado de la especificación sintáctica.

Gracias a que JavaCC genera la clase *TokenMgrError*, es posible personalizar los mensajes de error para que sean comprensibles para el usuario y le ayuden a resolver el error que ha tenido lugar.

4.6.5 Crítica

4.6.5.1 Mezcla de código y especificación gramatical

Como la descripción del fichero de entrada en JavaCC es muy similar a la de Antlr, sigue arrastrando el mismo problema desde el punto de vista de la ingeniería del software por el hecho de mezclar las acciones semánticas y la especificación gramatical. Estos ficheros de especificaciones pueden volverse realmente complejos cuando se trata de obtener procesadores de lenguajes con cierta envergadura, ya que contienen muchas más cosas que la especificación gramatical propiamente dicha.

Así mismo, como ya se mostró en la sección 4.5.5.2, la duplicación de las acciones semánticas en la especificación y en el programa generado, deja al programador la responsabilidad de mantener actualizados el fichero de especificaciones y el programa resultante. Esta tarea se podría realizar mejor en un entorno de desarrollo integrado.

4.6.5.2 Ciclo de desarrollo largo

Mantiene un ciclo de desarrollo largo y el depurado del código de las acciones semánticas sigue también los mismos pasos vistos en la sección 4.1.5.4.

Por otra parte, se incrementa la dificultad del trabajo de depuración puesto que los errores que se cometen en el fichero de especificación aparecen cuando se compila y ejecutan los analizadores generados. Acortar este ciclo subsanando los

errores únicamente en la salida del generador de analizadores, implica tener un fichero de especificaciones no actualizado con los últimos cambios realizados.

4.6.6 Características interesantes

4.6.6.1 Integración de los analizadores léxico y sintáctico

JavaCC integra en el propio fichero de entrada, las especificaciones léxicas y sintácticas necesarias para generar los analizadores, aunque en este caso, la especificación de cada parte no comparte la misma sintaxis.

4.6.6.2 Capacidad y modo de análisis

El analizador sintáctico contenido en la clase *Parser*, es recursivo descendente LL(k). Debido a la naturaleza de estos analizadores, el código generado por JavaCC es más fácil de depurar y comprender que el de Lex/YACC, basado en tablas de análisis LALR(1). Se considera más útil este tipo de análisis a la hora de depurar el procesador de lenguaje generado, cuando se produzcan errores en el código aportado por el usuario.

Las especificaciones gramaticales de JavaCC admiten también la notación EBNF y es posible generar ASTs, muy convenientes si se desea construir compiladores de más de una pasada.

4.7 ELI

4.7.1 Características generales

El sistema Eli[1] [GHL+92] combina métodos y técnicas de construcción de compiladores que son aplicables a un conjunto de problemas relacionados con el procesamiento de lenguajes. Utiliza para ello diversas herramientas estándar que implementan potentes estrategias en la construcción de compiladores, dentro de un entorno de programación de dominio especifico.

Se basa en la descomposición de los problemas de compilación en un número de subproblemas más pequeños asociados a cada una de las fases de su desarrollo: análisis léxico, análisis sintáctico, análisis semántico, transformación y generación de código.

Las soluciones son descritas en forma de especificaciones declarativas (en vez de código escrito), que pueden ser creadas de forma explícita o extraídas de una librería de especificaciones reusables. El uso de especificaciones permite la descripción de la naturaleza del problema, siendo las herramientas las que se encargan de traducir dichas especificaciones a código.

La construcción del procesador de lenguaje está dirigida por un sistema experto llamado *Odin* que oculta los detalles de uso de las herramientas de tres formas: eliminando redundancias de las especificaciones, ocultando todo el proceso que se necesita para preparar la salida de una herramienta como entrada de otra, y simplificando las peticiones a los usuarios durante el desarrollo del compilador.

Eli genera un compilador a partir de las especificaciones y las relaciones existentes entre las estructuras del texto del programa fuente, el árbol sintáctico abstracto del programa fuente, el árbol sintáctico abstracto del programa objeto y el conjunto de instrucciones del código máquina.

4.7.2 Características relativas a su funcionamiento

La descripción completa de un compilador consta de un conjunto de ficheros que deben ser suministrados por el usuario al sistema. Cada fichero contiene una especificación o parte de ella, y tiene una extensión que indica el tipo de ese fichero. Los tipos de los ficheros son utilizados por el sistema experto para determinar como van a ser procesados.

[1] El sistema Eli ha sido desarrollado en 1989 por la Universidad de Colorado.

Entrada al sistema

La entrada inicial al sistema Eli consiste en un fichero de texto con la extensión `.specs` y cuyo nombre es el que se dará al compilador que se va a generar. El sistema pasa este fichero a través del preprocesador de C y luego interpreta cada línea como el nombre de un fichero de especificación. El uso del preprocesador permite al usuario agrupar nombres de ficheros de especificaciones, controlar la selección mediante directivas e incluir especificaciones de las librerías.

Especificación de los subproblemas

Proporciona lenguajes especializados, que tienen un propósito determinado para realizar las especificaciones de los subproblemas. Por ejemplo, el lenguaje OIL (*Operator Identification Language*) es usado para resolver el problema de identificación de operadores y permite especificar indicaciones, operaciones, y coerciones permitidas por el lenguaje fuente.

El análisis semántico y la generación de código intermedio se realiza a través de una gramática atribuida escrita en LIDO, definiendo las computaciones asociadas a los símbolos *no terminales* de la gramática. Estas computaciones son invocaciones a las funciones definidas por los módulos generados a partir de las especificaciones, o por los módulos estándar de la librería del sistema

Interacciones entre las herramientas

La interfaz de cada herramienta consiste en un fichero shell script de UNIX. Este fichero es usado para ejecutar la herramienta y hacer que se adapte al entorno del sistema.

4.7.3 Critica

4.7.3.1 Conjunto amplio de especificaciones

El sistema utiliza un conjunto amplio de especificaciones, con el formato impuesto por los lenguajes especializados que soporta, para poder generar un compilador. Esto significa que el usuario, que debe proveer esas especificaciones, necesitaría tener un conocimiento exhaustivo de las opciones, y convenciones necesarias para utilizar bien el sistema. El tiempo de aprendizaje resulta ser demasiado grande, aunque solamente se necesiten desarrollar pequeños procesadores de lenguajes.

4.7.3.2 Falta de uniformidad en las especificaciones

Las tareas de mantenimiento de los procesadores generados pueden verse afectadas en gran medida por esta falta de uniformidad en las especificaciones.

4.7.3.3 El tamaño de las especificaciones puede ser elevado

El usuario necesita aportar muchas especificaciones a través de ficheros. Estas especificaciones pueden volverse demasiado grandes para ser manejadas cómodamente.

4.7.3.4 Complejidad de las estructuras de datos

Las implementaciones de las estructuras de datos utilizadas por el sistema son bastante complicadas, debido a los numerosos requerimientos de funcionalidad, espacio y velocidad de las operaciones realizadas con estas estructuras.

4.8 GENTLE

4.8.1 Características generales

Gentle[1] [Wil97] es un sistema que abarca todo el desarrollo de las fases de un compilador, liberando al usuario de la necesidad de tratar con los detalles de la implementación.

Se trata de un sistema de construcción de compiladores que parte de un conjunto de especificaciones de alto nivel. Soporta un lenguaje de descripción que proporciona una notación uniforme para todas las especificaciones.

4.8.2 Características del lenguaje

En Gentle una especificación consta de un lista de declaraciones. Una declaración puede introducir un tipo de dato, un predicado, una variable global o una tabla global.

El **tipo de dato** más importante con el que opera es término (*term*). Un término puede ser una constante simple o una estructura compuesta, y se puede ver como una notación lineal de una estructura representada en forma de árbol. Por ejemplo, `f(g(a,b), h(c,d))`.

Un **predicado** es usado para especificar una acción y se lleva a cabo cuando se invoca al predicado. Puede tener parámetros de entrada y de salida, y se describe mediante reglas que siguen una estructura análoga a la notación BNF.

Este lenguaje permite al usuario implementar tipos y predicados en el lenguaje C, pero los usa como si estuvieran especificados en el propio lenguaje de Gentle.

Las **variables globales** son usadas para representar un concepto que es global al concepto procesado por una regla particular.

En Gentle no está permitido modificar los campos de un término. Utiliza para ello las **tablas globales**, que representan colecciones ilimitadas de registros. Cada registro, identificado por una única clave, contiene una lista de uno o mas campos que sí pueden ser modificados.

---

[1] Gentle fue diseñado en 1989 en el centro de investigación nacional de Alemania (GMD). Este grupo también ha creado los compiladores de Ada y Modula-2, el generador de la fase de síntesis de un compilador denominado BEG y la herramienta Cocktail.

Gentle es descendiente del lenguaje de descripción de compiladores CDL de Kees Koster de 1969.

4.8.3 Arquitectura

El sistema permite que la especificación completa esté contenida un solo fichero o que pueda estar dividida en un conjunto de módulos. Cada módulo representa en este caso un fichero. Todos los módulos de especificaciones tienen la extensión `.g`.

Para generar un compilador es necesario invocar a Gentle con cada uno de los módulos de especificaciones. En este caso, Gentle traduce cada `module.g` en un `module.c`. A partir del módulo que contiene la especificación gramatical, se generan los siguientes ficheros (*figura 6.4*):

- `gen.lit`. Incluye las especificaciones léxicas de los símbolos *terminales*.
- `gen.tkn`. Contiene una lista de tokens introducidos en la especificación.
- `gen.h`. Introduce los tipos de datos para los atributos de los tokens y define los códigos asociados. Es un fichero cabecera de C.
- `gen.y`. Engloba las especificaciones necesarias para la herramienta YACC.

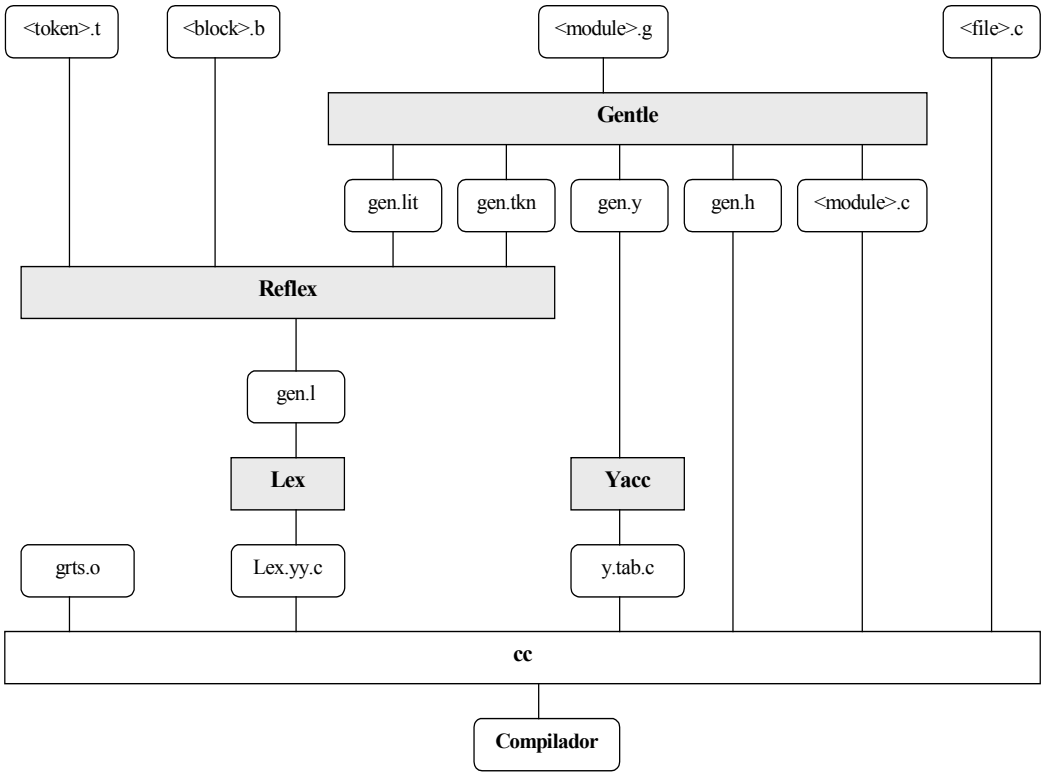


Figura 6.4: Arquitectura de Gentle

Para generar el analizador léxico, el sistema utiliza la herramienta *Reflex*. Su función consiste en juntar y combinar varios ficheros para crear el fichero de

especificaciones (`gen.l`) necesario para la herramienta *Lex*. De los ficheros que usa *Reflex*, dos son generados por el sistema y los otros dos son:

- `token.t`. Uno para cada token introducido en la especificación. El fichero debe especificar una expresión regular para el token y una acción que computa a los atributos del token.

- `block.b`. Permite definir los espacios en blanco y el formato de los comentarios.

El fichero `file.c` contiene código añadido, suministrado por el usuario y el fichero `grts.o` es el *run-time* del sistema que soporta un conjunto de módulos C para cubrir la funcionalidad básica del sistema.

4.8.4 Crítica

4.8.4.1 Añade un capa más de especificaciones

Con este sistema se añade una capa más de especificaciones en el desarrollo de un procesador de lenguaje. Parte de un conjunto de especificaciones uniformes de alto nivel y las divide en dos conjuntos diferentes para que puedan adaptarse a las especificaciones léxicas y sintácticas requeridas por las herramientas tradicionales Lex/YACC.

4.8.4.2 Mezcla de acciones y especificaciones

Las especificaciones pueden volverse realmente complejas cuando se trata de obtener procesadores de lenguajes a gran escala, ya que contienen muchas más cosas que la especificación gramatical propiamente dicha. Además, la posibilidad de que el sistema permita su división en varios ficheros incrementa la dificultad de mantenimiento.

4.8.4.3 Tratamiento básico de los errores

Se limita a llamar a la función *yyerror()* de YACC señalando la posición y un mensaje de error, para terminar posteriormente la ejecución del programa.

4.9 Spirit

4.9.1 Características generales

Spirit [Guz01] es un generador de analizadores sintácticos recursivos descendentes orientado a objetos. Su implementación se basa en el uso de una técnica conocida como *template meta-programming*[1] [Vel95] y es una de las muchas utilizadas en la programación generativa[2] [CE00].

Este sistema permite aproximar la sintaxis de una gramática expresada en notación EBNF a código escrito exclusivamente en C++, a diferencia de los generadores de analizadores convencionales, que necesitan realizar una traducción para pasar de un código fuente EBNF a código C/C++ u otro lenguaje específico. Para adaptarse a las reglas de sintaxis de este lenguaje C++, se deben realizar ciertas modificaciones en la notación original EBNF.

El analizador sintáctico generado es recursivo descendente no determinista. Se necesita realizar un análisis con retroceso (*backtraking*), comprobando todas las alternativas posibles al recorrer la jerarquía de clases para determinar si un objeto puede reconocer la secuencia de entrada.

El analizador léxico se encuentra integrado dentro del sistema y realiza un análisis tanto a nivel de caracteres como a nivel de frases. No se encarga de extraer tokens en un sentido tradicional.

El sistema permite crear analizadores de mayor complejidad a partir de la composición de unas pocas clases primitivas. Esta composición es totalmente estática (sucede en tiempo de compilación) y se hace posible a través de la flexibilidad que proporcionan las plantillas de C++.

4.9.2 Características relativas a su funcionamiento

La entidad más básica dentro del sistema es un objeto *parser*. Un parser acepta una entrada (secuencia de caracteres) y devuelve un objeto `Match` como resultado, que se evalúa a verdadero si puede reconocer esa parte de la entrada.

---

[1] Es una técnica mediante la cual el mecanismo de instanciación de plantillas del compilador de C++ es usado para evaluar parcialmente un programa en tiempo de compilación. Por tanto, hace posible escribir programas en un subconjunto de C++ para ser interpretados en tiempo de compilación.

[2] El objetivo de la Programación Generativa (GP) es modelar familias de sistemas software a través de módulos software. Dada una especificación de un requerimiento particular, es posible obtener componentes de esa familia, reusables y adaptables de forma automática.

Todos los *parsers* heredan de la clase base `Parser`, cuya función es ofrecer una interfaz común a todas las subclases, y éstas a su vez, deben tener una función miembro `Parse` compatible con dicha interfaz.

En Spirit, cada regla tiene asignado un parser identificado a través de un nombre. El tipo y comportamiento de éste se codifica mediante una clase abstracta denominada `AbstractParser`. El operador asignación y el constructor de copia de la regla, son los que se encargan de crear posteriormente una instancia concreta de esta clase abstracta. Mediante el nombre asociado a una regla se permite que su parser pueda ser referenciado en otra parte de otra regla.

Las acciones semánticas, se añaden al sistema como funciones C/C++ y serán llamadas dentro del contexto en el que se encuentren, si en el proceso de análisis se reconoce la entrada correspondiente. Estas funciones deben ser compatibles con la interfaz impuesta por el sistema.

4.9.3 Crítica

4.9.3.1 Realiza un análisis No Determinista

El enfoque dado a la herramienta, como un generador de analizadores sintácticos recursivos no determinista, tiene el inconveniente de la aplicación de un análisis con retroceso. Fundamentalmente para el desarrollo de procesadores de lenguajes a gran escala, el tiempo de compilación se verá incrementado de manera sustancial.

4.9.3.2 La herramienta se encuentra aún en fase inicial de desarrollo

Spirit se encuentra en fase de desarrollo y tiene aún numerosas carencias. Por ejemplo, no permite generar árboles sintácticos abstractos (ASTs) y no incluye un tratamiento y recuperación de errores.

4.10 Comparación, carencias y tendencias

El estado del arte de las técnicas revisadas, pone de manifiesto que el uso de herramientas para la construcción de procesadores de lenguajes resulta ventajoso. A partir de una o varias especificaciones, que como mínimo incluirán la gramática del lenguaje que se va a procesar, es posible generar analizadores léxicos y sintácticos eficientes e incluso todas las fases de un compilador, sin demasiado esfuerzo.

4.10.1 Comparación

En términos generales podríamos decir que existe una gama de sistemas (*Lemon, Gentle, Eli, Cocktail, Accent)* que ofrecen muy pocas variaciones con respecto a las técnicas usadas en las herramientas clásicas Lex/YACC[1], otro grupo de herramientas se integra mejor con los lenguajes orientados a objetos (*Antlr, JavaCC*) y otros incluyen técnicas que incorporan características orientadas a objetos (*Spirit*).

Un resumen de las características individuales más destacables de cada uno de los sistemas revisados se puede ver en la tabla de la sección 4.11.

4.10.2 Carencias

A excepción de Spirit, todos los sistemas revisados se caracterizan porque **mezclan especificaciones** sintácticas y semánticas (a veces también léxicas) y se apoyan en lo que podríamos denominar **técnicas basadas en procedimientos**. Estas características impiden que se puedan beneficiar dichos sistemas de las ventajas que ofrecen la aplicación de tecnologías orientadas a objetos.

De forma generalizada podríamos señalar las cinco carencias más importantes atribuibles a las técnicas de construcción de procesadores de lenguajes revisadas: reusabilidad, modularidad, mantenibilidad, extensibilidad y entornos visuales de desarrollo.

4.10.2.1 Reusabilidad

El hecho de que se mezclen las especificaciones sintácticas y semánticas (e incluso léxicas) atenta gravemente contra los principios de la ingeniería del software. Se incrementa la *complejidad* puesto que es más difícil comprender la estructura y el sentido de una especificación gramatical, y hace que sean muy *poco reutilizables*.

[1] Existe una lista bastante amplia de herramientas que contienen pequeñas variaciones de las clásicas Lex/YACC. En la presente tesis solamente se han incluido las más representativas.

4.10.2.2 Modularidad

Las acciones semánticas y los tipos de datos asociados a los símbolos de la gramática, pueden cambiar según las circunstancias. Esto significa que sería necesario realizar cambios en la especificación incluso cuando no se desea que cambie la gramática.

Siguiendo en la misma línea, cualquier cambio realizado en el código de las acciones semánticas significa volver a utilizar la herramienta que se haya elegido para procesar el fichero de especificaciones, compilar y ejecutar el programa generado. Este proceso es claramente ineficiente ya que complica y ralentiza el ciclo de desarrollo.

4.10.2.3 Mantenimiento

La duplicación del código de acción semántica en la especificación y en el programa generado, deja al programador la responsabilidad de mantener actualizados el fichero de especificaciones y el programa resultante

Por otra parte, este hecho hace que se incremente de manera importante el esfuerzo necesario para poder depurar los inevitables errores cometidos en la fase de desarrollo, puesto que el programador no está familiarizado con el código, probablemente complejo, generado por la herramienta.

4.10.2.4 Extensibilidad

La construcción de procesadores de lenguajes es una tarea que debe comenzar a partir de cero, aunque existan otros ya implementados con especificaciones análogas. La noción de reusabilidad no ha sido incorporada al proceso de construcción de los procesadores, y por tanto no existen mecanismos que permitan aprovechar parte de las implementaciones ya existentes para extenderlas con nuevas características.

4.10.2.5 Entornos visuales de desarrollo

Las herramientas actuales no se integran de manera sencilla en los diferentes entornos de desarrollo de aplicaciones. El uso de generadores para la construcción de procesadores de lenguajes implica escribir código en un lenguaje de programación sin que se puedan utilizar entornos de desarrollo específicamente diseñados para dicho lenguaje.

De igual forma, estos sistemas generadores no incorporan un entorno integrado de desarrollo que permita a los usuarios de las distintas disciplinas, no necesariamente expertos en el campo de la teoría de compiladores, desarrollar procesadores de lenguajes con el menor esfuerzo posible.

4.10.3 Tendencias

Cada vez está tomando más fuerza la utilización de procesadores de lenguajes, de manera extensiva, en numerosas disciplinas. Esto significa que pueden ser de utilidad para muchas personas que no necesariamente están especializadas en el campo de la informática. Diseñar herramientas para usuarios no expertos que puedan incorporar entornos visuales de desarrollo, debe ser uno de los objetivos a tener presente.

Por otra parte, es necesario que sean útiles para el tratamiento de los diferentes lenguajes y que se puedan aplicar de manera sencilla a las distintas disciplinas.

La aplicación de **tecnologías orientadas a objetos** para la **construcción de generadores** de procesadores de lenguajes resulta un buen punto de partida para afrontar los nuevos retos. De esta forma, se podrían beneficiar de las características importantes que mantienen.

4.11 Resumen de características de las técnicas revisadas

Nombre	Tipo de Análisis	Lenguaje Generado	Tratamiento de errores	Especifi-cación	Otras Características Destacables
Lex/Yacc	LALR(1)	C	Malo	BNF mezclado con C	- Falta de integración entre analizadores léxicos y sintácticos. - Ciclo de desarrollo largo. - No permite generar ASTs.
Lemon	LALR(1)	C	Malo	BNF mezclado con C	- Mantiene la misma estructura y modo de operar de Yacc. - Ciclo de desarrollo largo. - No permite generar ASTs.
Accent	AE/AP	C	Malo	EBNF mezclado con C	- Procesa cualquier gramática Libre de Contexto. - Mantiene la misma estructura que los generadores clásicos. - Ciclo de desarrollo largo. - No permite generar ASTs.
Cocktail	LALR(1) LL(1)	C Modula-2	Regular	BNF mezclado con C o Módula-2	- Mantiene características similares a los generadores clásicos. - Formado por un conjunto de herramientas . - Falta de integración entre analizadores léxicos y sintácticos. - Ciclo de desarrollo largo.

LL-ND = Recursivo Descendente No Determinista, AE/AP = Análisis Exhaustivo/Análisis Predictivo

Tabla 4.1 Resumen de características de las técnicas de construcción de Procesadores de Lenguajes

Nombre	Tipo de Análisis	Lenguaje Generado	Tratamiento de errores	Especificación	Otras Características Destacables
Antlr	LL(k)	C++/Java	Bueno	EBNF mezclado con C++/Java	- Flexibilidad de predicados costosa. - Ciclo de desarrollo largo. - Falta de integridad en los ASTs generados.
JavaCC	LL(k)	Java	Bueno	EBNF mezclado con Java	- Ciclo de desarrollo largo. - Dispone de una herramienta adicional para generar ASTs.
Eli	LALR(1)	C	Regular	Especificaciones Declarativas	- Usa un conjunto amplio de especificaciones declarativas para describir la naturaleza del problema. - Falta de uniformidad en las especificaciones. - El tamaño de las especificaciones puede ser elevado. - Necesita estructuras de datos complejas.
Gentle	LALR(1)	C	Malo	Especificaciones de alto nivel	- Añade una capa más de especificaciones. - Utiliza internamente Lex/Yacc
Spirit	LL-ND	C++	Malo	Código en C++	- Usa la técnica *template meta-programming* para generar los analizadores sintácticos. - Las acciones semánticas se añaden al sistema como funciones

LL-ND = Recursivo Descendente No Determinista, AE/AP = Análisis Exhaustivo/Análisis Predictivo

Tabla 4.1 Resumen de características de las técnicas de construcción de Procesadores de Lenguajes (continuación)

CAPÍTULO 5

TECNOLOGÍAS ORIENTADAS A OBJETOS

En el capítulo anterior se ha planteado la problemática existente con los sistemas actuales de construcción de procesadores de lenguajes. Como nota a destacar se señalaba la carencia de reusabilidad, mudularidad, extensibilidad, mantenimiento y entornos visuales de desarrollo.

Los problemas anteriormente mencionados pueden beneficiarse de la adopción, por parte de los sistemas generadores, del paradigma de orientación a objetos [Hol95]. En este capítulo se comentan las características que aporta la aplicación de tecnologías orientas a objetos basadas en marcos de aplicación y patrones.

5.1 Introducción

Tradicionalmente, la construcción de procesadores de lenguajes ha tomado una aproximación procedimental. En su forma más simple, el proceso está formado por un analizador léxico, un analizador sintáctico, un analizador semántico y un generador de código (ver sección 3.2).

Los analizadores léxico, sintáctico y semántico, operan como entidades monolíticas y aunque esto pueda resultar adecuado en entornos estáticos, sería necesario realizar una aproximación más modular en casos donde la sintaxis o las acciones semánticas estén sujetas a cambios.

Este desarrollo monolítico, resulta por tanto inadecuado debido fundamentalmente a diversas tendencias de las que se podrían destacar principalmente tres:

- La creciente aparición de aplicaciones de dominio específico.
- El uso de técnicas de compilación para obtener entradas estructuradas. Por ejemplo, en determinados servicios el usuario puede realizar una petición marcando los dígitos que están asociados a un determinado servicio, estos dígitos son procesados por un analizador sintáctico y a continuación el servicio es activado. Estos servicios están sujetos a cambios en la mayoría de los sistemas.

- Los modelos de lenguaje extensible. Un lenguaje puede ser extendido con nuevas construcciones y la semántica de las construcciones existentes puede ser cambiada.

5.2 Problemas derivados del procesamiento tradicional

Se pueden identificar diversos problemas derivados del procesamiento tradicional [Bos96, Bos97].

- *Complejidad*. El procesamiento monolítico trata de resolver la complejidad de las aplicaciones mediante su descomposición en distintas fases. Aunque así se logra disminuir su complejidad, sería necesario buscar niveles mayores de descomposición. Por ejemplo, el tratamiento de las gramáticas no provee mecanismos que permitan su descomposición modular, esto significa que cualquier modificación realizada en una parte de la gramática puede afectar a otras.

- *Modularidad*. Los analizadores léxico, sintáctico, semántico y el generador de código, actúan como filtros, es decir, procesan la entrada y la transforman en la salida correspondiente. Si se extiende el lenguaje para el que se construye el procesador, cada una de las fases se verán afectadas. Con un diseño modular, los procesadores se construirían básicamente a partir de elementos ya implementados. Solamente se implementarían los nuevos elementos del lenguaje específico.

- *Mantenimiento*. Aunque la construcción de procesadores se ha descompuesto en varias fases, cada una de ellas se puede ver como una entidad larga y compleja, con muchas interdependencias por lo que dificulta su mantenimiento.

- *Reusabilidad*. El dominio de los procesadores tiene una base teórica muy sólida, sin embargo, para construir un procesador casi siempre es necesario empezar de cero, incluso cuando existen aplicaciones similares disponibles. No existen mecanismos que permitan aprovechar procesadores ya implementados y extenderlos con nuevas características.

- *Extensibidad*. Es un concepto relacionado con la reusabilidad. Si se añaden nuevas especificaciones a un determinado lenguaje, y ya existe un procesador para dicho lenguaje, extenderlo requeriría realizar muchas adaptaciones. Sería muy útil poder reusar las partes existentes de la nueva especificación mediante el uso de técnicas de procesamiento que lo soporten.

5.3 Marcos de aplicación en procesadores de lenguajes

Muchos de los problemas presentes en los diferentes sistemas y herramientas de construcción de procesadores de lenguaje son debidos a que se fundamentan en técnicas procedimentales, estando este problema agravado por el hecho de que en

muchas ocasiones, los procesadores de lenguaje suponen proyectos de gran escala y complejidad.

Cambiar la forma en que los procesadores de lenguaje son construidos en general, no es una tarea sencilla. Trabajan sobre textos fuente en una serie de etapas, lo cual facilita pensar en técnicas procedimentales.

Los marcos de aplicación[1] (*frameworks*) suponen un concepto derivado del paradigma de la orientación a objetos ampliamente difundido y utilizado [Boo94, BRJ99, JBR99, RJB99].

Podemos usar la siguiente definición [JF88].

Definición 5.1 (Marco de aplicación) *Un marco de aplicación es un conjunto de clases que incorporan un diseño abstracto para dar solución a un conjunto de problemas relacionados*.

De una manera más informal, se podría decir que es un diseño reutilizable de una parte o del total de un sistema software, que es descrito por un conjunto de clases abstractas y por el modo en que las instancias de estas clases colaboran. Esto significa que es posible desarrollar un conjunto de clases diseñadas para trabajar de forma conjunta, así como con otras clases externas al marco de aplicación. Estas clases van a facilitar el desarrollo de determinados sistemas informáticos, agrupados bajo un ámbito concreto de actuación.

Hay muchos ejemplos que han supuesto una notable ventaja a la hora de desarrollar sistemas informáticos en ciertos ámbitos. Como ejemplos podemos destacar **AWT** (*Abstract Window Toolkit*) [Abs01] el estándar API para proporcionar interfaces gráficas de usuario (GUIs) en Java, **JFC** (*Java Foundation Classes*) [Jav01a, Fow01] que extiende el original AWT añadiendo un conjunto de librerías de clases para crear interfaces gráficas portables o **MFC** (*Microsoft Foundation Classes*) [Mic01] para crear programas en el sistema operativo Windows.

5.3.1 Tipos de marcos de aplicación

Los marcos de aplicación actualmente, se crean con una estructura fundamental basada en interfaces, y una o más implementaciones por defecto. De esta manera se puede reutilizar el código cuando sea aplicable, y el diseño siempre.

Pueden ser clasificados según las técnicas usadas para extenderlos en marcos de aplicación de caja blanca y caja negra [FS97].

Los marcos de aplicación **de caja blanca**[2] (*whitebox frameworks*), dependen de características de los lenguajes orientados a objetos, como la herencia y el enlace dinámico, para conseguir la extensibilidad. La funcionalidad existente es

[1] En los capítulos sucesivos se utilizará el término frameworks
[2] De ahora en adelante frameworks genéricos.

reutilizada y extendida en primer lugar, cuando el usuario modifica y particulariza el comportamiento, creando clases que heredan de las clases base, y en segundo lugar, escribiendo métodos usando patrones como por ejemplo el *Template Method* [GHJ+95]. Se necesita un conocimiento exhaustivo de su estructura interna y aunque son ampliamente usados, tienden a producir sistemas que dependen de los detalles de la especificación, como las jerarquías de herencia.

Los marcos de aplicación **de caja negra**[1] (*blackbox frameworks*), soportan la extensibilidad definiendo interfaces para componentes que puedan ser introducidos mediante la composición de objetos. La funcionalidad existente es reutilizada definiendo componentes que se amoldan a una interfaz particular, o integrando esos componentes en el marco de trabajo usando patrones como *Strategy* y *Functor* [GHJ+95]. Al hacer más uso de la composición y delegación de objetos, y menos de la herencia, son más estructurados, fáciles de usar y extender que los anteriores. Sin embargo, son más difíciles de desarrollar ya que es necesario definir interfaces que anticipen el uso potencial de los usuarios.

5.3.2 Aspectos que favorecen una arquitectura basada en Frameworks

- Las técnicas de implementación de lenguajes son genéricas y especializadas para cada lenguaje en particular.
- Las nociones abstractas del lenguaje son especializadas en estructuras sintácticas concretas.
- Los mismos conceptos semánticos abstractos pueden ser usados en muchos lenguajes y especializados en diferentes estructuras concretas del lenguaje.
- Las estructuras del lenguaje están organizadas jerárquicamente.

5.3.3 Ventajas

Los frameworks pueden ser utilizados para:

- Mejorar el proceso de desarrollo de procesadores de lenguajes, facilitando la introducción de técnicas orientadas a objeto.
- Reducir de forma notable el coste asociado a la creación de aplicaciones haciendo fácil la reutilización tanto del diseño como del código, en un ámbito bien definido como es el de los procesadores de lenguaje.
- Proporcionar a las personas con un conocimiento básico de la teoría de procesadores una plataforma adecuada para poder desarrollar sin muchos problemas sus propios procesadores de lenguaje.

[1] De ahora en adelante frameworks específicos.

5.3.4 Aplicación práctica

El campo de los procesadores de lenguaje tiene una base teórica muy consolidada, el número de etapas en las que se divide su desarrollo y el orden que estas siguen, es en la mayoría de los casos siempre igual. Este hecho debería hacer que la aplicación de los marcos de aplicación en este tipo de sistemas fuese una tarea sencilla.

Lamentablemente, esto no sucede así. Los procesadores de lenguajes trabajan con una serie de etapas comunes, pero dependen tanto del lenguaje sobre cuya gramática han sido construidos, que no es viable diseñar un marco de trabajo para todos los procesadores en general.

Se podría enfocar el diseño desde dos puntos de vista diferentes. Primero, haciendo un diseño tan general como para poder ser utilizado para resolver cualquier problema en la construcción de procesadores, de cualquier tipo de lenguaje que pueda ser diseñado y desarrollado. Segundo, proveer al usuario solo un núcleo compacto de funcionalidad para dar cabida a cualquier tipo de lenguaje. En el primero de los casos estamos ante una solución inviable debido al tamaño desmesurado mientras que el segundo no sería útil. Por tanto, ninguno de los enfoques presentados serviría para que fuera posible desarrollar procesadores de lenguaje de manera práctica

Se ha pensado en un enfoque intermedio o más bien mixto a la hora de establecer un marco de trabajo para el diseño e implementación del prototipo en la presente tesis [LLD+98].

Por un lado, parece evidente que un compacto grupo de clases, que provea la funcionalidad mínima de un procesador de lenguaje, es algo conveniente para poder alcanzar un buen grado de *reusabilidad*, tanto de código como de conceptos, entre diferentes procesadores de lenguaje. Por otra parte, es necesario un determinado grado de *flexibilidad,* puesto que se necesita un marco adaptado al lenguaje para el que se está desarrollando el procesador.

5.4 Beneficios derivados del uso de Frameworks

La incorporación de técnicas orientadas a objetos, mediante el uso de frameworks, proporciona a los desarrolladores de procesadores de lenguajes importantes beneficios [FS97].

- *Modularidad*. Se incrementa el nivel de modularidad encapsulando los detalles de la implementación, puesto que éstos pueden cambiar, de la parte que proporciona la interfaz. De esta forma se ayuda a mejorar la calidad de las aplicaciones, localizando el impacto del diseño y los cambios de la implementación. Esta localización reduce el esfuerzo requerido para entender y mantener las aplicaciones existentes.

- *Reusabilidad*. Las interfaces estables proporcionadas por los frameworks, permiten la reusabilidad definiendo componentes genéricos que puedan ser utilizados para crear nuevas aplicaciones. Se consigue evitar así la creación y prueba de nuevas aplicaciones, que son necesarias para encontrar soluciones comunes a otras ya existentes. La reutilización de componentes puede mejorar de manera sustancial la productividad de los programadores, además de mejorar la calidad, seguridad, interoperabilidad y rendimiento de las aplicaciones. Por tanto, reduce el tamaño del código escrito por el programador, puede acortar el tiempo de desarrollo y hace que el código sea más fácil de leer y mantener.

- *Extensibilidad*. Se permite la extensibilidad proporcionando métodos explícitos que permitan a las aplicaciones extender sus interfaces. Estos métodos sistemáticamente adaptan las interfaces y los comportamientos de un dominio de aplicación, a las variaciones requeridas por medio de instanciaciones de una aplicación en un contexto particular. La extensibilidad es fundamental para asegurar nuevas características y servicios a los usuarios de las aplicaciones.

- *Mantenimiento*. El mantenimiento de los frameworks se puede realizar de diferentes formas, añadiendo o quitando funcionalidad y generalización. Las aplicaciones que los usan también deben ser modificadas pero esta tarea es más sencilla con el uso de técnicas orientadas a objetos.

- *Inversión de control*. Permite al framework determinar que conjunto de métodos de aplicación es necesario invocar en respuesta a eventos externos. De esta forma, cuando el usuario tiene que implementar un conjunto de funciones o especializar determinadas clases, invoca un único método y el framework se encarga de realizar el resto del trabajo.

5.5 Relación entre frameworks y otras tecnologías orientadas a objetos

Los marcos de aplicación están muy relacionados con otras tecnologías orientadas a objetos.

- *Patrones*. Presentan soluciones recurrentes a los problemas de desarrollo del software dentro de un contexto particular. La primera diferencia es que los frameworks permiten reutilizar diseños concretos, algoritmos e implementaciones en un lenguaje de programación determinado mientras que los patrones permiten reutilizar diseños abstractos. Cuando se utilizan los patrones para estructurar y documentar los frameworks, cada clase juega un papel bien definido y colabora de manera efectiva con otras clases. Los patrones se pueden ver como los elementos que añaden la semántica a los frameworks de manera efectiva.

- *Librerías de clases*. Se pueden considerar tecnologías complementarias. Por ejemplo, los marcos de aplicación utilizan librerías de clases internamente para simplificar el desarrollo del framework como por ejemplo el C++ STL (Standard Template Library). Así mismo, los manejadores de eventos del framework, pueden utilizar librerías de clases para realizar tareas básicas como por ejemplo el procesamiento de cadenas, o el manejo de ficheros.

- *Componentes*. Un componente define un conjunto de operaciones que pueden ser reutilizadas conociendo la sintaxis y la semántica de sus interfaces. Se pueden usar frameworks para desarrollar componentes, en este caso las interfaces de los componentes proporcionan la estructura interna de las clases del framework, y componentes para desarrollar frameworks específicos.

5.6 Patrones y Frameworks

La mayoría de los patrones utilizados en el desarrollo del software han sido patrones de diseño. Su éxito de debe fundamentalmente a la gran popularidad y aceptación del libro *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJ+95] que describe patrones de diseño orientado a objetos, no obstante, existen otros tipos de patrones que abarcan todos los aspectos de la ingeniería del software.

Una definición de patrón aplicable en términos generales es la siguiente [RZ96].

Definición 5.2 (Patrón) *Un patrón es la abstracción de una solución concreta dada para un problema concreto que ocurre una y otra vez en un contexto determinado*.

El término *patrón de diseño* se utiliza de forma habitual para hacer referencia a los diferentes tipos de patrones que interviene en el análisis, diseño e implementación de aplicaciones software. No obstante, diversos autores [RZ96, BMR+96] prefieren hacer una importante distinción entre esos tres niveles conceptuales, definiendo tres tipos de patrones: patrones arquitectónicos, patrones de diseño y patrones de programación.

- *Patrones arquitectónicos*. Expresan una organización estructural fundamental para los sistemas software. Son estrategias de alto nivel que relacionan componentes y propiedades globales de un sistema. La forma de estos patrones se describe mediante términos y conceptos del dominio de la aplicación.

- *Patrones de diseño*. Proporcionan un esquema para perfeccionar los componentes de un sistema software o las relaciones entre ellos. Su forma se describe mediante objetos, clases, herencia, agregación e interacción.

- *Patrones de programación*. Describen como se deben implementar aspectos particulares de los componentes o las relaciones entre ellos, utilizando para ello, las características de un determinado lenguaje de programación.

Los patrones de diseño pueden ser utilizados para el diseño y la documentación de un framework. Sin embargo, aunque un framework pueda ser visto como la implementación de un sistema de patrones de diseño, son dos conceptos bien distintos. Un framework es *software ejecutable* mientras que los patrones de diseño representan *conocimiento* y *experiencia* sobre el software. En este sentido, los frameworks representan la implementación física de una o más soluciones a través de patrones software y los patrones constituyen las instrucciones que indican como implementar esas soluciones.

Las principales diferencias entre patrones de diseño y frameworks son las siguientes [GHJ+95].

- *Los patrones de diseño son más abstractos que los frameworks*. Los frameworks pueden estar codificados en un lenguaje de programación y ser estudiados, ejecutados y reutilizados directamente. En contraste, los patrones de diseño explican los propósitos, intercambios y consecuencias de un diseño pero deben ser implementados cada vez que se utilizan.

- *Los patrones de diseño son elementos arquitectónicos más pequeños que los frameworks*. Por regla general, un framework contiene diversos patrones de diseño pero lo contrario nunca es cierto.

- *Los patrones de diseño son menos especializados que los frameworks*. Los frameworks siempre tienen un dominio de aplicación particular mientras que los patrones de diseño, pueden ser usados prácticamente por cualquier tipo de aplicación.

CAPÍTULO 6

MÁQUINAS ABSTRACTAS ORIENTADAS A OBJETOS

En este capítulo se describen las ventajas que proporciona el uso de máquinas abstractas en general, para el diseño e implementación de los procesadores de lenguajes, señalando además las características que ofrece la plataforma Java para su desarrollo.

6.1 Introducción

El uso de máquinas abstractas no orientadas a objetos, ha sido, y aún sigue siendo, bastante habitual en el desarrollo de procesadores para los distintos lenguajes de programación, debido fundamentalmente a las grandes ventajas de portabilidad. Basta con desarrollar el emulador software para cada plataforma y el código binario de la máquina abstracta podrá ser ejecutado en diversos entornos sin dificultad.

Se podría señalar otra gran ventaja, la flexibilidad que se tiene con el software construido, ya que permite modificar fácilmente decisiones de implementación y añadir o eliminar características.

Pero no solamente se han diseñado máquinas abstractas pensando en la portabilidad del código. El proyecto UNCOL (Universal Computer Oriented Languages) [Ste60] tenía como objetivo la reducción del esfuerzo en la implementación de compiladores de diferentes lenguajes para diferentes arquitecturas. Los compiladores de cualquier lenguaje generarían código para el juego de instrucciones de la máquina abstracta y a partir de este lenguaje intermedio se usarían generadores de código para diferentes plataformas hardware.

6.2 Salto semántico

A pesar del éxito de las tecnologías orientadas a objetos, la adopción del paradigma OO no se ha realizado de manera integral, y el problema de desadaptación se produce cuando hay que realizar un cambio de paradigma al interactuar un elemento (por ejemplo una aplicación OO), con otro (por ejemplo el

sistema operativo). Este hecho se produce básicamente porque la mayor parte del hardware convencional sobre el que deben funcionar las tecnologías OO son versiones evolucionadas de la arquitectura de Von Newmann.

Los sistemas operativos ofrecen abstracciones basadas en este tipo de hardware siendo más adecuadas para el paradigma procedimental estructurado. Así los contextos de ejecución y espacios de direcciones de los procesadores de lenguajes convencionales tienen sus abstracciones correspondientes en el sistema operativo: procesos, memoria virtual y mecanismos de comunicación entre procesos (IPC, Inter-Process Communication).

Las aplicaciones OO se estructuran por regla general en un conjunto de objetos que colaboran entre sí mediante la invocación de métodos. La distancia semántica entre esta estructura y las abstracciones que ofrecen los sistemas operativos es muy grande. Por ejemplo, los objetos de los lenguajes en que se desarrollan las aplicaciones suelen ser de tamaño pequeño (grano fino). Sin embargo, el elemento más pequeño que manejan los sistemas operativos es el de un proceso asociado a un espacio de direcciones, de un tamaño mucho mayor, por lo que obliga al compilador del lenguaje a estructurar los objetos internamente dentro de un espacio de direcciones.

Para solucionar los problemas anteriores se acaba recurriendo a la interposición de capas adicionales de software que adapten la gran diferencia existente entre el paradigma OO y los sistemas actuales.

Se deben considerar varios problemas por el uso de las capas de adaptación sobre sistemas tradicionales

- Disminución del rendimiento del software
- Falta de uniformidad y trasparencia
- Pérdida de portabilidad y flexibilidad
- Aumento de la complejidad
- Soluciones parciales a los problemas que surgen
- Pérdida de productividad

6.3 Acercamiento de la OO al hardware

Se trata de reducir la gran distancia semántica existente entre el hardware tradicional y las tecnologías orientadas a objetos, haciendo que las aplicaciones trabajen con una máquina que utilice sus mismos términos OO.

Esto es una continuación de la tendencia existente desde los primeros momentos de la informática. La evolución de la programación, primero desde el código máquina al lenguaje ensamblador, y posteriormente la utilización progresiva de lenguajes cada vez de más alto nivel hasta llegar a los lenguajes orientados a objetos, son un ejemplo de esta tendencia.

Simplemente se trata de continuar esta tendencia con la arquitectura básica sobre la que funcionan las aplicaciones. El hardware ha evolucionado soportando conceptos utilizados en los lenguajes y entornos, aunque con la orientación al paradigma estructurado-procedimental y sistemas operativos para este paradigma. Ejemplos son el soporte para la programación estructurada, con llamadas a procedimientos que ya proporcionan los procesadores (instrucciones CALL). También el soporte para multiprocesamiento (cambio de contextos por hardware, etc) y la protección de procesos (memoria virtual y segmentación, etc.).

El auge de las tecnologías orientadas a objetos hace que ya sea el momento de proporcionar un soporte similar para estas tecnologías, en forma de máquina abstracta por razones de flexibilidad, portabilidad etc. Esto no excluye que también deba tenderse a incluir este soporte en procesadores reales.

6.4 Ventajas del uso de máquinas abstractas OO

Con la utilización de máquinas abstractas OO se consiguen una serie de ventajas [Alv98]:

6.4.1 Portabilidad y heterogeneidad

La utilización de una máquina abstracta dota de portabilidad al sistema. El juego de instrucciones de alto nivel puede ejecutarse en cualquier plataforma donde esté disponible la máquina abstracta. Por tanto, los programas escritos para máquinas abstractas son portables sin modificación a cualquier plataforma. Basta con desarrollar una versión (emulador o simulador) de la máquina para ejecutar cualquier código de la misma sin modificación: el código es portable incluso entre plataformas heterogéneas.

6.4.2 Facilidad de comprensión

La economía de conceptos, con un juego de instrucciones reducido únicamente en la OO se facilita la comprensión del sistema. Está al alcance de muchos usuarios comprender no sólo los elementos en el desarrollo, si no también la arquitectura subyacente. Esto se facilita aún más al usarse los mismos conceptos de la OO que en la metodología de desarrollo de la OO.

6.4.3 Facilidad de desarrollo

Al disponer de un nivel de abstracción elevado y un juego de instrucciones reducido, es más sencillo desarrollar procesadores de nuevos leguajes OO o de lenguajes ya existentes. La diferencia semántica que debe salvar un procesador de lenguajes entre los conceptos del lenguaje y los de la máquina es muy reducida, pues el alto nivel de abstracción OO de la máquina ya está muy cercano al de los lenguajes

6.4.4 Buena plataforma de investigación

La facilidad de comprensión y desarrollo, así como la portabilidad y heterogeneidad, permitirán que más personas puedan acceder al sistema sobre cualquier equipo y utilizarlo como base para la investigación en diferentes áreas de las tecnologías OO. Esto se aplica también a la propia máquina en sí cuya estructura permite una fácil experimentación con diferentes implementaciones de la misma.

6.5 Ventajas de la implementación de maquinas abstractas OO

En lo que concierne a la implementación de la propia máquina, también se obtienen una serie de ventajas.

6.5.1 Esfuerzo de desarrollo reducido

El juego de instrucciones reducido hace que el desarrollo de un simulador de la máquina abstracta sea muy sencillo. No hay que programar código para un número muy grande de instrucciones, con lo que el tiempo necesario y la probabilidad de errores disminuye.

6.5.2 Rapidez de desarrollo

Al ser la interfaz de la máquina independiente de estructuras internas, se puede elegir la manera interna de implementarla más conveniente. Para desarrollar rápidamente una implementación se pueden utilizar estructuras internas más sencillas aunque menos eficientes.

6.5.3 Facilidad de experimentación

Todo ello facilita la experimentación. Se puede desarrollar rápidamente una máquina para una nueva plataforma para hacer funcionar el sistema. Posteriormente, debido a la independencia de la interfaz, se pueden experimentar con mejoras internas a la máquina: optimizaciones, nuevas estructuras, etc. sin necesidad de modificaciones en las aplicaciones.

6.6 Minimización del problema del rendimiento de las máquinas abstractas OO

La propia naturaleza de una máquina abstracta necesita la utilización de un programa para simularla, con lo que la velocidad tiene que ser menor que si se usase el hardware directamente. Se manejan cifras que otorgan a los intérpretes una velocidad entre uno y dos órdenes de magnitud más lenta que el código

compilado [May87]. Sin embargo, existen una serie de razones que hacen que este problema de velocidad no sea tan grave, e incluso llegue a no tener importancia.

6.6.1 Compromiso entre velocidad y conveniencia

La simple velocidad no es el único parámetro que debe ser tenido en cuenta en un sistema. Lo verdaderamente importante es la percepción que tengan los usuarios de la utilidad del sistema, que es función del esfuerzo de programación, la funcionalidad de las aplicaciones, y el rendimiento conseguido. El éxito que ha alcanzado la plataforma Java [KJS96] está basado en la utilización de una máquina abstracta. Esto demuestra que el compromiso entre el rendimiento y la conveniencia de los beneficios derivados del uso de una máquina abstracta ya es aceptado por los usuarios con implementaciones sencillas de una máquina abstracta.

6.6.2 Mejoras en el rendimiento

Existen una serie de áreas con las que se puede mejorar el rendimiento de las máquinas abstractas, reduciendo aún más el inconveniente de su (aparente) pobre rendimiento.

6.6.2.1 Mejoras en el hardware

En la informática, se ha justificado históricamente la elevación del nivel de abstracción por la tendencia exponencial del aumento de rendimiento del hardware, junto a la disminución de su precio [Bol97]. Las máquinas abstractas son una continuación de esta tendencia, como lo fue el paso del ensamblador a los lenguajes de alto nivel. La potencia adicional se destina a elevar el nivel de abstracción (utilizar una máquina abstracta), que hace que los proyectos sean más baratos de desarrollar (hay una relación no lineal entre el nivel de abstracción y el coste de un proyecto). Los beneficios de un mayor nivel de abstracción compensan la pérdida de rendimiento.

Por otro lado, si con los procesadores convencionales actuales e implementaciones sencillas de máquinas abstractas se ha conseguido una gran aceptación, el aumento de potencia del hardware no hará más que minimizar aún más el problema aparente del rendimiento.

6.6.2.2 Optimizaciones en la implementación de las máquinas.

La implementación de una máquina abstracta pude optimizarse para que la pérdida de rendimiento sea la menor posible. Una técnica de optimización es la compilación dinámica o justo a tiempo (JIT, *Just In Time*). Se trata de optimizar la interpretación del juego de instrucciones de la máquina. En lugar de interpretarlas una a una, se realiza una compilación a instrucciones nativas (del procesador convencional) del código de los métodos en el momento de acceso inicial de los

mismos (justo a tiempo) [Adl96]. Los siguientes accesos a este método no son interpretados de manera lenta, si no que acceden directamente al código previamente compilado, sin pérdida de velocidad. Este código nativo compilado puede ir siendo optimizado aún más en cada llamada adicional (generación incremental de código) [HU94].

Ciertas implementaciones de máquinas abstractas que utilizan esta técnica han resultado sólo de 1.7 a 2.4 veces más lentas que un código C++ equivalente optimizado [Höl95].

Otro ejemplo de la posibilidad de optimización en la implementación de máquinas abstractas se comprueba en el producto Virtual PC de la compañía Connectix Corporation. *Virtual PC* es una aplicación Macintosh que emula un ordenador PC completo por software sobre una plataforma PowerPC-Mac. Se alcanzan relaciones de 5 a 9 instrucciones PowerPC por cada 3 instrucciones Pentium [Tro97].

6.6.2.3 Implementación en hardware

En aquellos casos en que no sea aceptable la pequeña pérdida de rendimiento de una máquina optimizada, se puede recurrir a la implementación de la máquina en hardware. Esta implementación en hardware especializado ofrecería un rendimiento superior al de cualquier implementación software [Way96].

6.7 La plataforma Java

En los últimos años, el lenguaje de programación Java™ [GJS96] ha ganado una enorme popularidad en Internet y en el desarrollo de numerosas aplicaciones. Aunque superficialmente tiene una sintaxis similar al lenguaje C++, Java proporciona otras características adicionales. Por ejemplo, tiene recolector de basura liberando al programador de la gestión de memoria, un mecanismo más claro de herencia con clases e interfaces, y una rica plataforma estándar con soporte para interfaces gráficas de usuario y programación en redes.

Una de las propiedades más interesantes de java es la portabilidad del código objeto. Los programas escritos en Java son compilados a una plataforma independiente como instrucciones *bytecode* y en tiempo de ejecución, esos *bytecodes* son interpretados por la máquina virtual de Java [LY97]. Es decir, un programa escrito en lenguaje Java y compilado a instrucciones *bytecode* puede ejecutarse sobre cualquier sistema operativo que incorpore esta plataforma. Esta portabilidad es posible porque el elemento principal de la plataforma Java es su máquina virtual. Además, por el beneficio que esto supone, hace que sea ideal para una red heterogénea como la Internet: el código para la máquina virtual puede moverse por la red, independientemente de cuál sea la plataforma destino.

En la presente tesis, se ha elegido tanto para la implementación del prototipo como para los procesadores de lenguajes que genera, el lenguaje Java por las ventajas señaladas anteriormente.

El sistema que se describe a continuación, permite construir procesadores de lenguajes completos, como es el caso de los interpretes o los traductores que procesan determinados lenguajes. Para el desarrollo de compiladores, el sistema no soporta generación y optimización de código para máquinas reales, se ha dado más importancia a la generación de código para máquinas abstractas ya que esto supone una serie de ventajas. El análisis de código y las optimizaciones, son más fáciles de realizar, ya que se utiliza un lenguaje de programación más cercano al que usa habitualmente un programador, y además se consigue una mayor flexibilidad y portabilidad.

CAPÍTULO 7

REQUISITOS DEL SISTEMA O2C2

En el capítulo 4 se revisaron las diferentes técnicas que ofrecían los sistemas generadores de procesadores de lenguajes actuales. De forma generalizada, estas técnicas presentaban problemas relacionados con la *modularidad*, *reusabilidad*, *extensibilidad, mantenimiento y entornos visuales de desarrollo*. Desde un punto de vista más concreto, la mayoría de los sistemas limitaban su funcionalidad puesto que incluían barreras adicionales que aumentaban su curva de aprendizaje, ofreciendo diferentes opciones y convenciones necesarias para manejar cada una de las herramientas, o dificultando la integración al ser herramientas que se desarrollan de forma independiente unas de otras.

Por otro lado, en el capítulo 3 se definió un generador de procesadores de lenguajes como un programa que transforma una especificación en un procesador para el lenguaje de programación descrito en la especificación. Entre los requisitos básicos que se le deben exigir al nuevo sistema, estaría principalmente el de que sirva para la misma finalidad para la que se han concebido los diferentes generadores de procesadores de lenguajes usados actualmente, y que a la vez resuelva los inconvenientes y carencias que estos presentan.

En este capítulo se identifican los requisitos que debe cumplir el sistema propuesto. Estos requisitos afectan básicamente al diseño y a la construcción del mismo.

El nombre dado al sistema, **O2C2**, proviene precisamente del propósito de su desarrollo, ofrecer un sistema que facilite y ayude al usuario en la ardua tarea de la **C**onstrucción de Pro**c**esadores de Lenguajes, aplicando Técnicas **O**rientadas a **O**bjetos.

7.1 Aplicación de técnicas orientadas a objetos a la construcción de procesadores de lenguajes

En la actualidad, ingeniería del software y metodologías orientadas a objetos son casi sinónimos. Los objetos están presentes en todas las fases del desarrollo de software: análisis, diseño e implementación. Existen gran cantidad de libros y

artículos que hablan de los beneficios de la utilización de las tecnologías de objetos en el desarrollo de software, de los que destacan [Boo94, Mey99].

Los requisitos enumerados en este apartado son aquellos que deben ser especialmente considerados en el diseño y la construcción del sistema.

7.1.1 Diseño del sistema como un marco orientado a objetos

Se trata en este caso de aplicar los principios de diseño e implantación del paradigma de orientación a objetos al sistema de forma que quede organizado como un marco orientado a objetos. El sistema ha de ser diseñado como una jerarquía de clases, cada una de las cuales ofrecerá una determinada funcionalidad.

Los beneficios de esta aproximación son dobles. En primer lugar, el sistema tendrá una estructura modular, en la cual determinados componentes podrán ser identificados y reutilizados. En segundo lugar, con el uso de mecanismos como la herencia, el sistema podrá ser configurado mediante la extensibilidad para unas necesidades específicas, asegurando nuevas características y servicios o cumpliendo determinadas restricciones.

7.1.2 Modelo de objetos uniforme y homogéneo

Los objetos se presentan como una herramienta muy útil para estructurar la funcionalidad del sistema por varias razones:

- Los objetos proporcionan modularidad y encapsulación, dado que separan claramente la interfaz de la implementación.

- Los sistemas estructurados en términos de objetos, son más fáciles de extender y mantener. Además, los subsistemas individuales podrán ser reutilizados con facilidad y adaptados a las diferentes condiciones de operación o entorno, vía al mecanismo de la herencia.

De esta forma, los diferentes módulos que serán identificados para dotar al sistema de los servicios básicos: *análisis léxico*, *análisis sintáctico, análisis semántico* y *tratamiento de errores*, podrán ser modelados por medio de conjuntos de objetos. La comunicación entre los diferentes grupos de objetos podrá realizarse a través de las interfaces de las principales clases que colaboran en el sistema y que son ofrecidas por los frameworks.

7.1.3 Patrones

En el capítulo 5 se mostró como los frameworks están muy relacionados con otras tecnologías orientadas a objetos. La incorporación de patrones al sistema proporcionaría un mecanismo mediante el cual se permitiría la comunicación de las clases que contienen el código de acción semántica, suministradas por el usuario, con las clases generadas por el sistema. De esta forma, las clases de

usuario que podrán estar asociadas a las diferentes secciones de la gramática, serán activadas cuando sea posible reconocer dichas secciones durante el proceso de análisis.

7.2 Especificaciones sencillas

El sistema deberá utilizar un metalenguaje sencillo que permita describir la sintaxis de los diferentes lenguajes de programación. Por otro lado, también deberá permitir que se obtengan los distintos tipos de componentes léxicos que tengan que ser procesados a partir de un texto fuente.

El uso de un metalenguaje conlleva al establecimiento de al menos tres criterios para su elección. En primer lugar, que sea fácil de aprender y de utilizar. En segundo lugar, que pueda estar integrado con las diferentes notaciones utilizadas en la representación de funciones y operadores. En tercer lugar, que no disponga de construcciones que permitan cometer errores fácilmente.

Es necesario tener presente que el uso de distintas especificaciones, o el uso de un conjunto amplio de lenguajes especializados en la descripción de especificaciones, actúan como capas intermedias incrementando el proceso de desarrollo del software, y esto supone una carga adicional para el usuario. En este sentido, no debería ser un punto de partida impuesto por el sistema, puesto que el tiempo de aprendizaje se vería incrementado notablemente.

7.3 Tratamiento y recuperación de errores

Un buen diseño del sistema debe estar dotado de mecanismos para el tratamiento y recuperación de los errores. La incorporación de un mecanismo robusto tiene que cumplir un conjunto de requisitos que garanticen como mínimo las siguientes características:

- Aplicar el modelo de objetos para conseguir un mecanismo completo y fácil de utilizar. Su aplicación podrá ser llevada a cabo de manera uniforme por lo que no sería necesario introducir conceptos nuevos en el sistema.

- Permitir que los usuarios del propio sistema puedan localizar y corregir los errores producidos durante su utilización.

- Permitir que el sistema pueda incorporar de una forma sencilla código especializado para el tratamiento de los errores que pudieran producirse, para el normal funcionamiento de los procesadores de lenguajes generados por el sistema.

- Permitir que el sistema se recupere y permanezca en un estado estable para que pueda seguir su proceso.

7.4 Generación de AST

El sistema deberá permitir que se generen árboles sintácticos abstractos (AST). Dependiendo de la complejidad del lenguaje de programación a tratar, a veces resulta necesario que los analizadores operen a través de varias pasadas para recoger diversa información del fichero fuente. La obtención de AST permitiría conseguir procesadores de lenguajes que realicen sus tareas en varias pasadas, facilitando además las comprobaciones semánticas que se deben realizar.

7.5 Entorno visual de desarrollo

Para que el usuario pueda disponer de toda la funcionalidad ofrecida por el sistema, se deberá proporcionar un entorno visual de desarrollo usable y completo, que facilite al usuario la manipulación del mismo. El entorno deberá ser fácil y cómodo de utilizar por usuarios no experimentados y con conocimientos limitados en la teoría de procesadores de lenguajes. Para ello, el entorno visual ha de incorporar un conjunto de herramientas que faciliten las tareas de desarrollo, teniendo en cuenta además que:

- Deberá ser diseñado para poder automatizar, en la medida de lo posible, todo el proceso de construcción de procesadores de lenguajes. Se conseguiría de este modo, facilitar el desarrollo rápido de las aplicaciones sin demasiado esfuerzo.

- Deberá permitir que las distintas herramientas que integran el entorno puedan interactuar de una forma libre, sin que el usuario tenga que establecer los enlaces necesarios con el código generado por las distintas herramientas.

A pesar de ser un requisito impuesto, el objetivo no es diseñar un sistema que esté condicionado por el entorno visual de desarrollo, el propio sistema debe ser capaz de proporcionar la misma funcionalidad básica si es utilizado a nivel de línea de comandos.

7.6 Usabilidad

La definición formal aplicada por la organización estándar internacional ISO (*International Standards Organization*) establece que la usabilidad es la "*eficacia, eficiencia y satisfacción con la que un conjunto de usuarios pueden afrontar un conjunto de tareas en un entorno particular*"

La definición anterior tiene un significado operacional pero también requiere una cierta medida de eficacia, eficiencia y satisfacción para su desempeño. El concepto de usabilidad es asociado muchas veces a las aplicaciones cuya interfaz de usuario juega un papel importante, sin embargo, es un concepto amplio que debe estar aplicado al desarrollo de cualquier producto software.

El sistema O2C2 deberá ser diseñado para que integre todos los aspectos relacionados con la usabilidad. Esta característica permitirá que el sistema sea eficiente, ofrezca un entorno visual agradable, sea fácil de aprender con un nivel de abstracción adecuado para la retención de ideas, sea fácil de usar, incorpore mecanismos para poder tratar y subsanar los errores rápidamente y ofrezca un buen grado de satisfacción por parte del usuario.

7.7 Plataforma de desarrollo

Desde un punto de vista teórico, un generador de procesadores de lenguajes no necesita una plataforma especial de desarrollo, el lenguaje elegido para su implementación y el de los procesadores generados hace que sea más o menos flexible y por consiguiente utilizable. Sin embargo, a la hora de establecer los requisitos del sistema es muy importante disponer de una buena plataforma de desarrollo para mejorar las prestaciones del sistema.

Se deberá proporcionar una plataforma de desarrollo que:

- Soporte la experimentación en el campo del diseño e implementación de lenguajes, automatizando las tareas de construcción desde el punto de vista educacional y comercial.

- Permita la portabilidad de las aplicaciones. Esta portabilidad queda garantiza si se utilizan técnicas de construcción de procesadores de lenguajes para máquinas abstractas en general, y además, el propio sistema se implementa utilizando un lenguaje de programación cuyo código objeto es portable.

- Sea flexible para conseguir que la funcionalidad del sistema esté proporcionada en forma de API para que las aplicaciones cliente puedan interactuar con el sistema desde diferentes entornos de desarrollo.

CAPÍTULO 8

DISEÑO DEL SISTEMA O2C2

Después de analizar los problemas y señalar las carencias que limitaban la funcionalidad de las diferentes técnicas de construcción de procesadores de lenguajes utilizadas actualmente, se especificaron en el capítulo anterior, los requisitos que debía cumplir el sistema O2C2 que se describe en este trabajo.

Con el fin de construir un sistema que cumpla los requisitos anteriormente mencionados, en este capítulo se describe la arquitectura del sistema O2C2 que forma el núcleo de la presente tesis doctoral. Una vez mostrada esta arquitectura se analizarán sus componentes, propiedades y características. Posteriormente, en los capítulos 9 y 10 se presentará el diseño e implementación del prototipo desarrollado para el sistema propuesto.

8.1 Objetivos de la arquitectura

De forma resumida, los objetivos que se pretenden alcanzar con la arquitectura del sistema son:

- ***Incorporar técnicas orientadas a objetos*** *al sistema para que le doten de características relativas a* ***modularidad, reusabilidad, extensibilidad y mantenimiento*****,** beneficiándose de esta forma los desarrolladores de procesadores de lenguajes, además de facilitar la construcción de software de calidad.

- ***Automatizar*** *todo lo posible* ***el proceso de construcción de procesadores de lenguajes***. Se logra así facilitar el desarrollo rápido de este tipo de aplicaciones permitiendo que usuarios no necesariamente expertos en el campo de la teoría de compiladores puedan utilizar el sistema sin demasiado esfuerzo.

- ***Permitir generar analizadores*** *léxicos y sintácticos* ***a partir de especificaciones sencillas***. Para abreviar su especificación, las gramáticas podrán estar expresadas en notación EBNF y se permitirá procesar la práctica totalidad de la gramáticas libres de contexto.

- ***Proporcionar un mecanismo de anclajes*** mediante el cual las acciones proporcionadas por el usuario para la aplicación del análisis semántico puedan estar asociadas a diferentes secciones de la gramática.

- ***Conseguir que las distintas herramientas*** *que integran el sistema* ***puedan interactuar*** *de una forma libre,* sin que el usuario tenga que establecer los enlaces necesarios con el código generado por las distintas herramientas.

- *Proporcionar* ***un entorno de desarrollo usable,*** fácil y cómodo de utilizar por usuarios no experimentados y con conocimientos limitados aunque no por ello debe ser menos potente que otras herramientas usadas para la misma finalidad.

- *Permitir generar* ***árboles sintácticos abstractos*** (*AST*). Con ello se conseguiría crear fácilmente compiladores de múltiples pasadas, además de facilitar las comprobaciones semánticas.

- ***Dotar*** *al sistema generador de* ***un mecanismo para el tratamiento y recuperación de errores robusto***. Se debe permitir de una forma cómoda, sencilla y fácil que el usuario pueda tratar errores encontrados y que el sistema se recupere y permanezca en un estado estable para que pueda seguir operando. El sistema debe facilitar también la realización del proceso de depuración.

- *Proporcionar* ***una plataforma que dé soporte a la experimentación*** en el campo del diseño e implementación de lenguajes de forma automática, desde un plano tanto educacional como comercial.

- *Conseguir que la* ***funcionalidad*** *del sistema sea* ***proporcionada en forma de API*** para que las aplicaciones cliente puedan interactuar con el sistema desde diferentes entornos de desarrollo.

8.2 Arquitectura del sistema

Para la consecución de los objetivos se parte de la implantación de nuevas líneas de desarrollo en la construcción de procesadores de lenguajes frente a las propuestas clásicas o convencionales [LLD+98, LLC+01]. Se aplica como método de diseño, el uso de técnicas de programación orientada a objetos basadas en *frameworks*, combinadas con *patrones*, mejorando de forma significativa la usabilidad de los sistemas generadores de procesadores de lenguajes actuales.

En la *figura 8.1* se muestra el esquema general del sistema. En esta estructura se identifican a muy alto nivel los elementos en los que se ha dividido el sistema para poder ser estudiados con más profundidad en la siguientes secciones. Consta de los siguientes bloques.

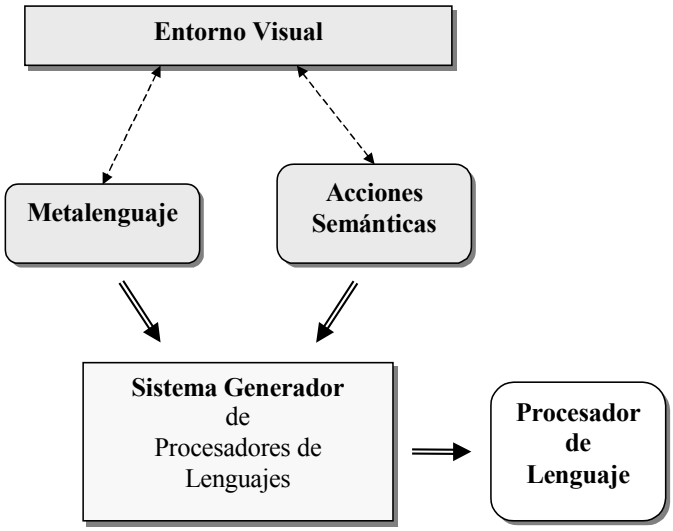


Figura 8.1: Esquema global del sistema

8.2.1 Entorno visual

Representa la capa de más alto nivel de la arquitectura y proporciona un entorno de trabajo que permite una interacción eficiente con el sistema generador. No obstante, como ya se señaló en el capítulo de requisitos del sistema O2C2, el objetivo no es diseñar un sistema generador que esté condicionado por el entorno visual, el propio sistema debe ser capaz de proporcionar la misma funcionalidad básica independientemente del entorno de desarrollo. Es decir, se podrá utilizar de manera análoga a nivel de línea de comandos y además, dicha funcionalidad podrá ser proporcionada en forma de API para que las aplicaciones cliente puedan interactuar con el sistema desde diferentes entornos de desarrollo.

Una descripción más completa de este entorno será expuesta en la sección 8.10.

8.2.2 Metalenguaje

Se utilizará para describir de forma sencilla las características del lenguaje. Se proporcionarán dos tipos: uno para reconocer los componentes léxicos y el otro para describir la sintaxis de los diferentes lenguajes.

Descripción de los componentes léxicos

Utilizando el entorno visual de desarrollo, podría realizarse la descripción de los componentes léxicos mediante la selección de los objetos encargados de reconocer los diferentes símbolos del lenguaje. El proceso de reconocimiento llevado a cabo por estos objetos queda oculto al usuario.

Por otro lado, también sería posible describirlos a nivel de línea de comandos. En este caso el usuario tendría que proporcionar el método `yylex`() encargado de reconocer los símbolos de la entrada, codificado de forma manual o generado mediante una herramienta.

Descripción de la sintaxis

Se limita a contener la gramática que describe la sintaxis del lenguaje. Para realizar esta descripción, se ha elegido la notación EBNF por aportar las principales propiedades que son atribuibles a un metalenguaje.

Las gramáticas especificadas por el usuario pueden ir acompañadas de enlaces. Estos enlaces son etiquetas (nombres entre comillas) que se colocan al lado de un fragmento gramatical, indicando que el usuario está interesado en captar los eventos sintácticos que se produzcan como consecuencia de la identificación o reconocimiento de dicha estructura gramatical. Es lo que se va a denominar *sistema de anclajes* y se describe en la sección 10.10 de este documento.

8.2.3 Acciones semánticas

Proporcionan todo el código de acción necesario para realizar un tratamiento semántico. Se utilizan para realizar diferentes tareas:

- Obtener un valor semántico a partir de los valores semánticos asociados a los distintos elementos que forman parte de la producción de un símbolo no terminal. Este valor podrá ser utilizado posteriormente en otros puntos de la especificación gramatical.

- Realizar comprobaciones de tipos para que las diferentes operaciones puedan ser llevadas a cabo con éxito.

- Permitir un tratamiento concreto de los errores que puedan originarse.

- Proporcionar el código necesario para la obtención de procesadores de lenguajes completos con sus diferentes aplicaciones.

Una descripción más completa de la funcionalidad de este bloque se puede ver en la sección 8.7, al tratar el análisis semántico.

8.2.4 Sistema generador

Representa el bloque base de la arquitectura. Debe proporcionar mecanismos capaces de transformar las especificaciones aportadas en forma de metalenguaje junto con las acciones semánticas, en un procesador de lenguaje. Las fases de construcción estarán integradas y encapsuladas facilitando el desarrollo de aplicaciones de forma sencilla y eficiente.

Una descripción más completa de este componente, que incluye los bloques funcionales básicos, será planteada en el apartado 8.4.

8.2.5 Procesador de lenguaje

Es el nombre genérico que se le asigna a la aplicación generada por el sistema. Engloba a todos los programas que realizan determinadas operaciones con los lenguajes, como ya se ha descrito en el capítulo 3.

Cuando se trate de obtener de forma automática un compilador para un determinado lenguaje, el sistema podrá generar una aplicación para traducir el programa fuente a código intermedio. En la descripción de la arquitectura del sistema no se contempla la posibilidad de optimizar este código intermedio y generar posteriormente código máquina. El capítulo 6 está íntegramente dedicado a justificar razonadamente cuáles son las causas que han llevado a tomar inicialmente esta decisión.

8.3 Aplicación de técnicas Orientadas a Objetos mediante Frameworks

En el capítulo 5 se mostraron los problemas derivados del desarrollo monolítico de los procesadores de lenguaje, señalando además que muchos de los problemas presentes en los diferentes sistemas y herramientas que utilizan los desarrolladores para su construcción, utilizan técnicas orientadas a funciones y procedimientos en vez de utilizar técnicas orientadas a objeto que fomentan características relacionadas con **modularidad**, **extensibilidad**, **mantenimiento** y **reutilización de código.**

Como es sabido, los frameworks suponen un concepto derivado del paradigma de la orientación a objetos. Integran una colección de clases para dar solución a un conjunto de problemas relacionados y pueden ser utilizados para mejorar el proceso de construcción de procesadores de lenguajes, facilitando la introducción de técnicas orientadas a objetos.

8.3.1 Creación de frameworks

No resulta factible diseñar un modelo único que pueda ser utilizado de forma general en el procesamiento de cualquier tipo de lenguaje, daría lugar a una decisión inviable debido al tamaño que pudiera llegar a tener dicho modelo. La solución pasa por otorgar al modelo un buen grado de *reusabilidad* y de *flexibilidad*.

El primer paso consiste por tanto en definir el framework genérico que se encargue de ser la base sobre la que se construirán posteriormente los frameworks específicos, adaptados a la descripción de la gramática de un lenguaje determinado.

Los frameworks que debe proporcionar el sistema para la generación de procesadores de lenguajes pueden dividirse en dos categorías:

- En primer lugar, existirá un framework encargado de proporcionar toda la funcionalidad necesaria y que es independiente del lenguaje hacia el que va dirigido. Sus clases podrán ser utilizadas a través de la herencia, haciendo que las clases derivadas añadan la funcionalidad necesaria para cada lenguaje concreto.

- En segundo lugar, existirá un framework generado específicamente para dar soporte a las características de un lenguaje concreto. Sus clases podrán ser utilizadas mediante la composición.

8.3.2 Comunicación con las clases del framework

La comunicación con las clases generadas se realiza a través de la implementación de las interfaces que debe proporcionar el framework. En el sistema propuesto intervienen cuatro tipos: la interfaz **Lexer** para tratar con las clases que realizan el análisis léxico, la interfaz **Parser** para manejar la clase principal de cualquier analizador sintáctico generado, la interfaz **Listener** para comunicarse con las clases que debe crear el usuario para incorporar el código de acción semántico a las distintas secciones gramaticales, y la interfaz **Error** que sirve para comunicarse con las clases proporcionadas por el usuario para el tratamiento de los errores producidos.

Esta comunicación se basa en el **modelo de delegación de eventos** [Cur01]. Los objetos que tienen algún interés en un determinado evento, se registran ante el objeto asociado a dicho evento y quedan esperando a que éste suceda. Estos objetos se denominan *Listeners* y deben implementar una interfaz específica, que define uno o más métodos, los cuales serán invocados por el objeto donde se originó el evento en respuesta a cada tipo de evento específico manejado por la interfaz.

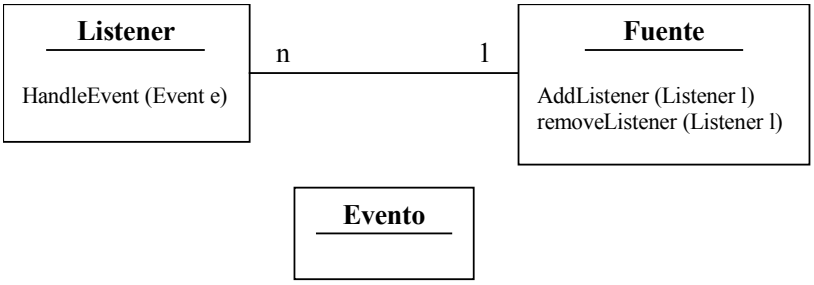


Figura 8.2 Diagrama de clases del patrón Listener

En la *figura 8.2* se muestra el diagrama de clases. Los objetos que generan eventos son llamados *objetos fuente*. Mantienen una lista de *objetos listeners* y se encargan de notificarles los eventos de interés que ocurren (*figura 8.3*). Cualquier número de listeners podrá ser añadido o eliminado de la lista cuando sea necesario. Para notificar la ocurrencia de un evento, el objeto fuente invoca un método con el objeto listener que corresponde al tipo especifico de evento que ha registrado.

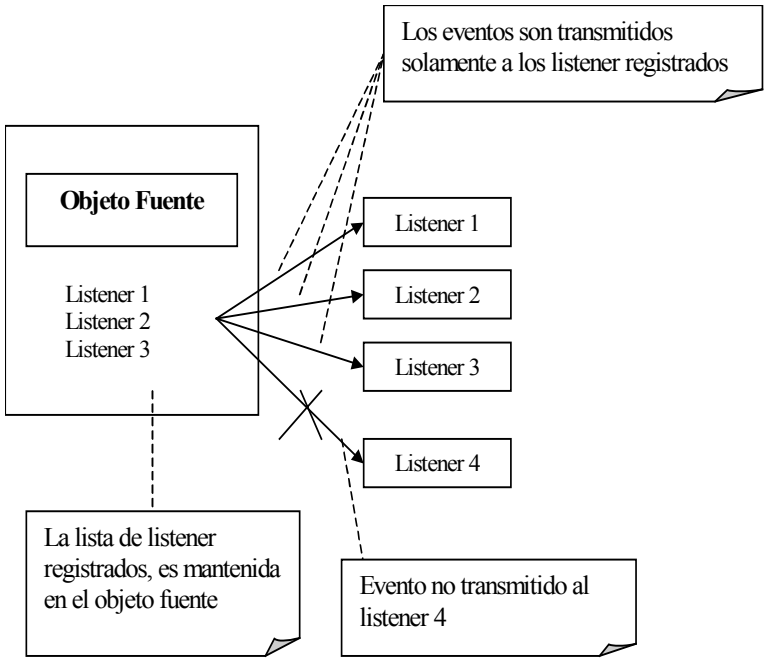


Figura 8.3 Notificación de eventos entre los objetos

8.4 Funcionalidad básica proporcionada por el sistema

El núcleo del sistema O2C2 lo constituye el sistema generador que será el encargado de proporcionar la funcionalidad básica del sistema. Y tal como se ha visto en el apartado anterior, la aplicación de técnicas orientadas a objetos basadas en frameworks proporcionan un conjunto de propiedades que van a servir de base para dotarle de dicha funcionalidad.

Los módulos básicos identificados para proporcionar los servicios del sistema son los siguientes:

- **Análisis Léxico**
- **Análisis Sintáctico**
- **Análisis Semántico**
- **Generación de código intermedio**
- **Detección, Tratamiento y recuperación de errores** (Gestor de errores)

Estos módulos representan en realidad un conjunto de objetos que están relacionados. Se les sigue agrupando en las clásicas unidades conceptuales para comprender más fácilmente su papel en esta arquitectura.

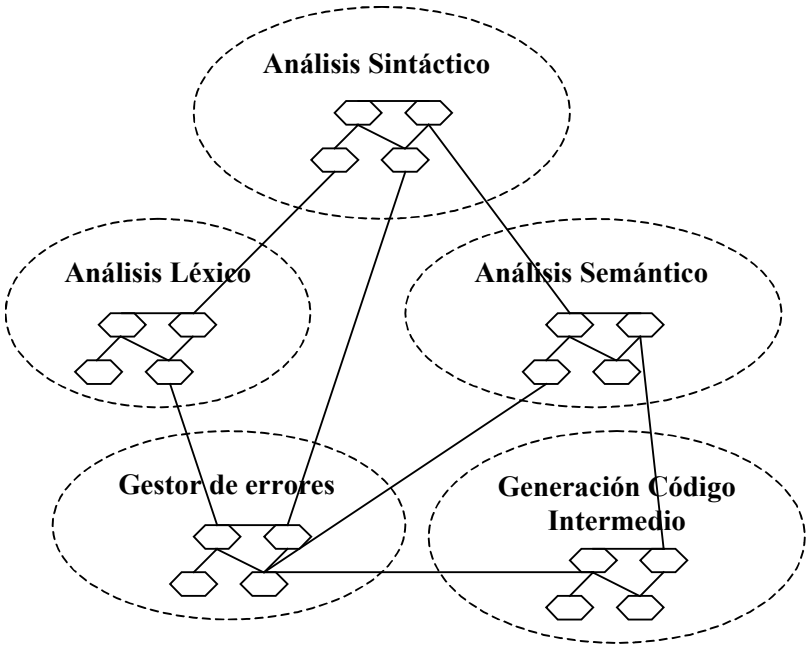


Figura 8.4: Módulos básicos del sistema

En el dibujo de la *figura 8.4* se representan los principales módulos que constituyen el sistema. No obstante, existen otros dos módulos que no han sido incluidos inicialmente porque no van a ser tratados explícitamente en este trabajo. Forman parte del back-end y dependen de la máquina objeto. Estos bloques son:

- **Optimización de código intermedio.** La finalidad de la optimización de código es producir un código objeto lo más eficiente posible.

- **Generación de código.** La fase de generación de código transforma la representación intermedia a código en lenguaje ensamblador o a código para una máquina real.

Como se pudo observar en el capítulo 3, los generadores se definían como herramientas que ayudaban en el proceso de construir procesadores de lenguajes generando alguna o todas las fases del procesador de forma automática.

En el plano teórico, podría parecer que un generador que construya todas las fases del procesador será más potente, sin embargo, en la práctica, los generadores parciales que construyen únicamente analizadores léxicos y/o sintácticos están integrados normalmente con un lenguaje de programación de propósito general. De esta forma, la implementación de estructuras de datos complejas, optimizaciones y análisis de código, son fáciles de realizar ya que se utiliza el lenguaje de programación nativo del programador

Se ha optado por una solución intermedia entre los generadores parciales y los generadores totales. Se va a utilizar un framework para crear procesadores de lenguaje de manera automática de tal forma que no solo se va a realizar un análisis léxico, sintáctico y semántico del lenguaje fuente sino que además se va a permitir añadir el código necesario para realizar cualquier tratamiento de la entrada, una

traducción, una interpretación o una generación de código para máquinas abstractas.

8.5 Análisis léxico

La herramientas convencionales utilizadas para generar analizadores léxicos se basan en patrones (*expresiones regulares*) y autómatas finitos deterministas. Por otra parte, el código de usuario se asocia a dichos patrones para ser ejecutado cuando es posible su reconocimiento. Este enfoque plantea varios inconvenientes:

- El uso de patrones oscurece el formato de los símbolos que son tratados, dificultando su entendimiento.
- Algunos problemas de reconocimiento resultan difíciles de resolver usando expresiones regulares.
- El ciclo de desarrollo – edición, generación y compilación – debe ser repetido para corregir, cambiar o extender el analizador léxico.

8.5.1 Modelo orientado a objetos

En este sentido, y siguiendo con la misma línea de actuación, resulta necesario establecer una nueva aproximación a través de las técnicas orientadas a objetos.

Un analizador léxico, será modelado como un conjunto de objetos. Cada uno de ellos podrá reconocer un determinado símbolo del lenguaje (identificadores, números, cadenas de caracteres, comentarios, espacios en blanco, etc.).

Desde el punto de vista del usuario final, existirá una librería de clases encargada de proporcionar diferentes símbolos que podrán ser configurables según las necesidades. Solamente sería necesario seleccionar los objetos para realizar un análisis específico y no tendría que preocuparse el usuario, del proceso de reconocimiento llevado a cabo por los objetos de forma individual.

8.5.2 Implementación del modelo

La colección de clases que permiten realizar el análisis léxico, estará proporcionada por un framework genérico que contendrá objetos que puedan reconocer los símbolos típicos de los lenguajes de programación. El usuario podrá usarlos, extenderlos o incorporar otros diferentes.

Las principales clases que lo integran son:

- *Clase scan*. Es la clase base abstracta para todos los objetos reconocedores de símbolos.
- *Clase input*. Es la clase encargada de proporcionar los caracteres de la entrada. Soporta el conjunto de caracteres del sistema Unicode o ASCII.

- *Clase scanner*. Es la clase encargada de manejar una tabla construida a partir de los objetos reconocedores (*objetos scan*) y contendrá métodos para poder añadir y eliminar objetos de la tabla.

Los caracteres de entrada son enviados a los objetos scan. Un objeto será marcado si no puede reconocer un carácter. El objeto seleccionado será el que consiga reconocer la secuencia de caracteres más larga. Si existen varios objetos no marcados, la ambigüedad puede resolverse a favor del primero que se haya localizado.

Una forma de optimizar la búsqueda del objeto ganador podría estar basada en los distintos tipos de símbolos que se deben reconocer en los diferentes lenguajes. El primer carácter de un símbolo tiende a ser significativo de su tipo (un número comienza por un dígito, un identificador por una letra, una cadena de caracteres por comilla, etc) y por tanto podría ser indexada la búsqueda por este carácter para seleccionar el objeto ganador.

La comunicación entre las clases del framework y las clases de usuario que contienen el código de acción semántica se realiza a través de una interfaz denominada `Lexer`.

Cuando un objeto scan es seleccionado como ganador, el objeto scanner debe invocar a la clase de usuario suscrita para realizar la acción correspondiente. Para que pueda ser llevado a cabo es necesario que:

- El método constructor de la clase de usuario realice la suscripción ante el objeto scan reconocedor del símbolo interesado.

- La clase de usuario implemente el método `yylex()` de la interfaz Lexer. Este método permite obtener los tokens de la entrada y será invocado cuando el objeto scan complete la secuencia de caracteres que puede reconocer.

8.5.3 Ventajas del modelo

El modelo descrito anteriormente ofrece numerosas ventajas respecto a las herramientas convencionales: permite la creación de clases que pueden ser usadas para especializar otras clases existentes sin necesidad de acceder al código fuente, las clases pueden ser reutilizadas por diferentes analizadores léxicos, los objetos encapsulan su estado simplificando de esta forma el reconocimiento individual y los objetos que representan las distintas acciones pueden ser reemplazados en un momento determinado por otros diferentes para atender otras necesidades.

8.6 Análisis sintáctico

El analizador sintáctico podrá ser generado a partir de la especificación exclusivamente sintáctica del lenguaje, utilizando para ello un metalenguaje

sencillo, la notación EBNF. Constituye la parte central que conforma y dirige todo el proceso de análisis llevado a cabo en un procesador de lenguaje.

8.6.1 Modelo orientado a objetos

El sistema deberá proporcionar un conjunto de clases para poder representar a cada uno de los elementos que puedan formar parte de la descripción de una gramática. Estas clases serán comunes a todas las gramáticas libres de contexto, con independencia de su estructura. Por otra parte será necesario también generar otras clases adaptadas al lenguaje específico para el cual se va a generar el procesador de lenguaje.

Desde el punto de vista del usuario final, solamente necesitaría proporcionar la especificación gramatical en notación EBNF y el sistema se encargaría de generar el framework que representa al analizador sintáctico para el lenguaje descrito por dicha gramática.

8.6.2 Implementación del modelo

El conjunto de clases que permiten llevar a cabo el proceso de análisis se organizan alrededor de un framework que se generará de forma automática por el sistema a partir de las especificaciones del lenguaje. Para llevar a cabo esta tarea se tendrán en cuenta dos grupos de clases generadas:

- En el primer grupo están las clases que conforman el núcleo del framework generado. Estas clases serán comunes a todos los frameworks generados por el sistema y proporcionarán un nivel de consistencia entre los procesadores de los diferentes lenguajes. La jerarquía de clases del núcleo estará diseñada para trabajar con los diferentes elementos que puedan aparecer en la descripción de una gramática expresada en notación EBNF.

- En el segundo grupo se encuentran las clases que complementan al núcleo y que son necesarias para que el sistema incorpore las características específicas del lenguaje. Se generará una clase por cada elemento no terminal, una clase por cada producción y una clase por cada uno de los elementos de tipo EBNF que forman parte de las producciones. Todas estas clases serán descendientes de las interfaces y las clases que componen el núcleo del framework.

Una descripción más detallada de la implementación llevada a cabo se puede ver en los capítulos 9 y 10 que describen la construcción del prototipo para el sistema O2C2.

8.6.3 Tipo de análisis

Existe una gran variedad de estrategias de análisis, cada una de ellas con diferentes capacidades de reconocimiento, que pueden ser utilizadas en la implementación de los analizadores sintácticos [GJ98].

En este sentido, una solución ideal pasaría por diseñar un sistema que pueda procesar gramáticas libres de contexto sin imponer restricciones (desde el punto de vista del usuario final) sobre la estrategia de análisis llevada a cabo. No obstante, como se pudo observar en el capítulo 3, algunos métodos son más eficientes para ser utilizados de forma generalizada.

El objetivo que aquí se persigue se centra en el desarrollo automático de analizadores deterministas ascendentes o descendentes. Para que en la práctica se pueda lograr este grado de flexibilidad, el sistema debería dar soporte a los dos tipos de análisis, implementando los correspondientes algoritmos de análisis y permitiendo adoptar en un momento dado, el que mejor se adapte a las necesidades reales del procesador del lenguaje.

Como es sabido, los analizadores deterministas descendentes LL(k) imponen ciertas limitaciones al diseño de las gramáticas. No pueden ser ambiguas, recursivas a izquierdas y cada símbolo no terminal debe satisfacer la condición LL(k). Su información de contexto está restringida a la secuencia de producciones reconocidas y debe tomar decisiones sin conocer otras alternativas. Los analizadores deterministas ascendentes LR(k) no imponen estas restricciones puesto que intentan corresponder varias producciones a la vez y posponen la toma de decisión hasta obtener suficiente información de entrada. Usan más información de contexto y resuelven muchas ambigüedades que se pueden presentar durante el análisis.

La cantidad de tokens disponibles para el analizador (*lookahead*) ha sido limitada por razones teóricas y prácticas exclusivamente a k=1. No obstante, existen lenguajes de interés práctico que pueden ser especificados más fácilmente con gramáticas LR/LL (k>1). El uso de k>1 requiere significativamente más espacio y tiempo en el generador y el procesador de lenguaje obtenido, pero se pueden solventar estos problemas implementado un heurístico efectivo, denominado *linear-approximate lookahead* [Par93] que elimina el coste exponencial del espacio necesario con un tratamiento convencional. En este sentido, la construcción de un analizador sintáctico que basa la toma de decisiones a un valor de k>1 debe ajustarse a los siguientes criterios.

1. El valor de k debe ser modulado de acuerdo a las necesidades de la toma de decisión de cada analizador sintáctico.

2. La aplicación del heurístico debe realizarse siempre que se pueda mejorar el coste de la implementación.

3. Es posible realizar una computación híbrida aplicando el heurístico y un tratamiento convencional en la toma de decisiones, para reducir los requerimientos de espacio y tiempo.

La implementación del heurístico mencionado, se aplicó en la construcción del generador de analizadores sintácticos descendente Antlr [PQ95].

8.7 Análisis semántico

El análisis semántico permite detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico y añadir el código necesario para realizar cualquier tratamiento de la entrada (compilación, interpretación o traducción) . La implementación podrá ser llevada a cabo a partir de la incorporación al sistema del código de acción semántica. Este código estará asociado a determinados elementos gramaticales y se ejecutará cada vez que se haya completado el correspondiente análisis sintáctico.

8.7.1 Modelo orientado a objetos

El modelo orientado a objetos capaz de comunicar el código de acción semántica proporcionado por el usuario con las clases del framework generadas por el sistema, se basa en aplicar un mecanismo, *el modelo de delegación de eventos*, que permita añadir este código de forma cómoda y sencilla beneficiándose a su vez de las propiedades derivadas del modelo orientado a objetos para solventar los inconvenientes presentes en otros sistemas.

De esta forma, es el usuario el encargado de crear las distintas clases con el código semántico necesario para completar el procesador de lenguaje generado. Estas clases estarán separadas de las clases generadas por el sistema aunque podrán comunicarse a través de las interfaces que éste proporcione.

8.7.2 Implementación del modelo

Una forma rápida y sencilla de incluir código para realizar cualquier tratamiento semántico sería escribirlo en las propias clases que representan las diferentes secciones de la gramática y que son generadas por el sistema. Esta solución presenta sin embargo numerosos inconvenientes:

- No facilita la modularidad en el sistema.
- No permite una separación clara entre el código generado por el sistema y el código proporcionado por el usuario.
- Un cambio en la especificación sintáctica obligaría a regenerar las clases pudiéndose perder la práctica totalidad del trabajo realizado por el usuario.

- El código añadido a una clase asociada a un elemento de la gramática no podría utilizarse en otras clases diferentes.

Una solución más viable pasaría por que la implementación de las acciones semánticas quedara reflejada en la propia estructura de las clases diseñadas por el usuario.

La comunicación con las clases sintácticas (generadas por el sistema) se realizaría a través de una interfaz denominada *Listener*. Esto significa que sería necesario realizar una suscripción. Las clases de usuario que actúan como listeners, se deben suscribir a las clases sintácticas para que sean notificadas de los eventos producidos al reconocer diferentes secciones gramaticales. Cada vez que se produzca un evento, se realizaría una invocación a las clases listeners para que puedan realizar el tratamiento correspondiente. Una clase sintáctica puede tener varias clases de usuario suscritas y una misma clase que actúa como listener puede estar suscrita a más de un evento, el modelo no impone restricciones en este sentido.

Una descripción más detallada de la implementación llevada a cabo se puede ver en el capítulo 10 que describe las ampliaciones y mejoras llevadas a cabo en el desarrollo del prototipo par el sistema O2C2.

8.7.3 Ventajas del modelo

Este modelo presenta dos ventajas fundamentales:

- En primer lugar, evita que las acciones semánticas estén mezcladas con los diferentes elementos que describen la gramática del lenguaje, existiendo una separación clara entre las clases generadas por el sistema y las clases de usuario. Se facilita de este modo la modularidad del sistema, y permite además mejorar el nivel de reconocimiento de analizadores deterministas que aplican el tipo de análisis LR, ya que está demostrado que las gramáticas que contienen acciones mezcladas entre los distintos símbolos de las producciones, fuerzan al analizador a tomar una decisión antes de tiempo y por tanto, con una menor información de contexto [PQ96].

- En segundo lugar, permite la reutilización del código semántico puesto que una misma clase semántica puede estar asociada a varios eventos sintácticos y varias clases semánticas pueden ser utilizadas en diferentes analizadores sintácticos. Por otra parte, cualquier cambio realizado en la especificación sintáctica no alteraría el contenido semántico de las clases.

8.8 Detección, recuperación y tratamiento de errores

Dentro de la funcionalidad mínima requerida a todo procesador de lenguaje se debe incluir la **detección** de los errores producidos durante la fase de análisis. Sin embargo, existen pocas circunstancias en las cuales el comportamiento de señalar

únicamente que se ha producido un error es aceptable[1]. Por norma general existe una gran diferencia en la percepción de un error entre el procesador y el usuario.

El estado interno del procesador debe ser adaptado para que pueda continuar procesando el resto de la entrada. Esta adaptación del estado interno es lo que se denomina **recuperación** de errores. Su propósito consiste en:

- Intentar detectar todos los posibles errores de sintaxis.
- Evitar la generación de mensajes de errores falsos. Estos mensajes son el resultado de una continuación del análisis después de producirse un error sin haber adaptado aún el estado interno del procesador de lenguaje.

En presencia de errores, la adaptación del estado interno puede dar lugar a acciones semánticas asociadas a las reglas gramaticales para ser ejecutadas en un orden que resulta imposible para entradas sintácticamente correctas, produciendo de esta forma resultados inesperados. Una solución a este problema pasa por ignorar estas acciones cuando se detecte un error sintáctico pero no resulta óptima y no debe ser adoptada. Otra solución que mejora la anterior consiste en usar un método para el tratamiento de los errores.

El método de **tratamiento** de errores puede transformar la entrada en otra sintácticamente correcta, borrando, insertando o cambiando símbolos. Su principal ventaja radica en que el analizador puede generar un árbol sintáctico y las acciones semánticas asociadas a las reglas pueden ser ejecutadas siempre en el orden correcto.

8.8.1 Modelo orientado a objetos

Los objetivos fundamentales que debe cumplir el modelo podrían resumirse utilizando dos términos: *completo* y *fácil*. Por una parte, ha de ser completo en el sentido de que el sistema debe detectar todos los posibles errores producidos durante el proceso de análisis, debe proporcionar mecanismos para la recuperación del sistema y adoptar un tratamiento adecuado, permitiendo al usuario del sistema que decida los tipos de errores que desea tratar y cómo hacerlo. Por otra parte, ha de ser fácil de utilizar por los usuarios, se deben utilizar conceptos ampliamente extendidos que no incrementen la curva de aprendizaje del nuevo sistema.

Para continuar con la aplicación de los mismos conceptos utilizados en el desarrollo del sistema, el modelo orientado a objetos propuesto se basará en el mecanismo de *delegación de eventos* y en las *excepciones*. Este último concepto es muy utilizado por los diferentes lenguajes de programación orientados a objetos.

[1] Para algunas aplicaciones especialmente interactivas sí podría ser satisfactorio.

8.8.2 Implementación del modelo

En la implementación del modelo, se utilizará una jerarquía de clases para dar soporte al mecanismo de detección, recuperación y tratamiento de errores. Cada clase representará algunos de los diferentes errores que puedan producirse durante la fase de análisis: *error léxico, error sintácticos, error semántico y error interno*.

El usuario puede tratar los errores producidos en las distintas secciones de la gramática, creando sus propias clases y realizando suscripciones ante las clases que delimitan las secciones para que puedan ser avisadas. Todas las clases del framework genérico deben incluir un *gestor de excepciones,* por lo que cada vez que se produzca una excepción, si el usuario ha demostrado interés en tratar el error, se pasaría el control a las clases suscritas para ese tipo de evento. Si no existen clases registradas, la excepción sería tratada por defecto en otro punto, antes de proseguir el análisis. De igual forma, también se permite al usuario que pueda tratar errores léxicos y semánticos.

La comunicación de las clases creadas por el usuario con las clases específicas del sistema se realiza a través de una interfaz denominada *Error*.

Una descripción más detallada de la implementación llevada a cabo se puede ver en el capítulo10 que describe las ampliaciones y mejoras llevadas a cabo en el desarrollo del prototipo para el sistema O2C2.

8.9 Generación de código intermedio

La generación de código intermedio podrá ser proporcionada por el usuario a través de las clases semánticas. Este código puede estar expresado en un lenguaje de alto nivel o de bajo nivel, aunque como ya se comentó anteriormente, el sistema no incorpora las fases del back-end que dependen de una máquina real específica.

8.10 Entorno visual de desarrollo

Como ya se mencionó en el capítulo 7, el entorno visual de desarrollo constituye uno de los requisitos que debe cumplir el sistema O2C2 y ha de permitir de forma sencilla e intuitiva, la realización de cada una de las funcionalidades proporcionadas por dicho sistema. Para que el usuario pueda acceder a toda esta funcionalidad, se incorporan un conjunto de herramientas que facilitan las tareas de desarrollo. Por tanto, es necesario dotar al sistema de una interfaz de usuario usable y completa que facilite al usuario la manipulación del mismo.

En este sentido, conviene destacar que la incorporación de entornos visuales constituye un aspecto que no ha sido considerado por los sistemas generadores y ninguno de los presentados en el capítulo 4 ofrece esta posibilidad de trabajo.

8.10.1 Características deseables

Las características que podríamos considerar esenciales para que el entorno visual del sistema permita establecer un buen grado de desarrollo de las aplicaciones son:

- **Transparencia**. La interfaz del sistema ha de ser transparente para permitir al usuario conocer en todo momento lo que sucede en el sistema.

- **Concisión y calidad en la representación**. El entorno debe ser conciso en la representación de las diferentes opciones y en la información al usuario.

- **Adaptabilidad y tutorialidad**. El entorno ha de poder adaptarse a las preferencias del usuario, y éste tiene que poder preguntar sobre el contexto en que se encuentra.

- **Completitud funcional**. El entorno debe proporcionar toda la funcionalidad disponible en el sistema O2C2.

- **Soporte de diferentes niveles de abstracción**. La interfaz tiene que permitir que el usuario elija el nivel de detalle deseado en la visualización de la información.

- **Independencia en las acciones**. El conjunto de funciones disponibles en cada contexto y la semántica de las acciones debe ser coherente en cada situación.

- **Integración**. El entorno debe permitir integrar en un solo entorno los procesos de análisis léxico, sintáctico y semántico.

- **Soporte para el desarrollo**. El entorno debe permitir flexibilizar la estructura de los proyectos de desarrollo y ofrecer la posibilidad de almacenar su información en un formato no plano.

8.10.2 Funcionalidad básica

Para poder ofrecer al usuario toda la funcionalidad del sistema O2C2, el entorno visual de desarrollo ha de incorporar al menos las siguientes herramientas:

- **Editor**. El entorno ha de permitir definir el conjunto de reglas que componen la especificación de la gramática y comprobar su adecuación en base al tipo de análisis realizado, el conjunto de tokens que conforman el léxico de un lenguaje y las acciones semánticas asociadas a las distintas secciones gramaticales. Para ello, se considera adecuado un editor que permita realizar las definiciones de una forma visual para facilitar la realización de las mismas y minimizar además el número de errores. También sería importante incorporar un módulo de definición de clases

totalmente visual, alternativo al editor de texto, y con el que se pudiese conmutar, de forma que una clase descrita visualmente tuviese su descripción textual en el lenguaje y viceversa.

- **Depurador**. El entorno debe estar provisto de un depurador que permita detectar y corregir fácilmente los errores producidos durante la fase de pruebas.

- **Visualizador**. Es deseable que el entorno permita la visualización de los datos y las estructuras internas así como los objetos que representan el AST. De esta forma, se podría ir conociendo paso a paso cómo se va realizando el proceso de análisis.

8.11 Resumen de las características ofrecidas por esta arquitectura

La arquitectura aquí presentada proporciona una serie de características que son enumeradas a continuación a modo de resumen.

- **Modularidad**. La separación clara entre las especificaciones y el código de acción semántica hace posible encapsular los detalles de la implementación ayudando de esta manera a entender y mantener mejor las aplicaciones generadas. Cualquier cambio realizado en la especificación sintáctica no alteraría el contenido semántico de las clases y a la inversa.

- **Extensibilidad**. La organización del código del sistema como un conjunto de clases hace posible que se puedan extender para adaptar su comportamiento a las nuevas condiciones requeridas por una aplicación en un contexto particular.

- **Reusabilidad**. Se consigue a través de la extensibilidad, reutilizando el propio código de las clases del sistema y también mediante la reutilización del código semántico, puesto que una misma clase semántica puede ser asociada a varios eventos sintácticos y varias clases semánticas pueden ser utilizadas en diferentes analizadores sintácticos.

- **Mantenimiento**. El mantenimiento del sistema generador se puede realizar añadiendo o quitando funcionalidad y generalización. Esta tarea es más sencilla con el uso de técnicas orientadas a objetos. En cuanto al mantenimiento de las aplicaciones generadas, resulta mucho más fácil su aplicación al estar separadas las clases del sistema y las clases de usuario.

- **Flexibilidad**. Se consigue a través de la incorporación del modelo de delegación de eventos como mecanismo de integración de las clases que aportan el código de acción semántica con el sistema.

- **Disminución de la complejidad**. La separación clara entre las especificaciones y el código de acción semántica permite comprender mejor la estructura y sentido de estas especificaciones así como el código

de las aplicaciones generadas. Por otro lado, esta separación también permite acortar el ciclo de desarrollo del software como se demuestra en el capítulo 11.

- **Usabilidad**. La idea de diseñar el sistema en base a criterios relacionados con la usabilidad permitirá que el sistema sea eficiente, fácil de usar y de aprender, ofrezca un entorno visual agradable y un buen grado de satisfacción por parte del usuario.

- **Entorno visual**. Facilita el trabajo del usuario permitiendo realizar de forma sencilla e intuitiva, cada una de las funcionalidades proporcionadas por el sistema.

- **Portabilidad de las aplicaciones**. Se garantiza su portabilidad si los procesadores de lenguajes son generados para máquinas abstractas y además, el propio sistema se implementa utilizando un lenguaje de programación cuyo código objeto es portable.

- **Integración flexible**. Se consigue una integración flexible si la funcionalidad del sistema es proporcionada en forma de API. De esta forma, las aplicaciones cliente podrán interactuar con el sistema desde diferentes entornos de desarrollo.

- **Soporte para el diseño e implementación de lenguajes**. Ofrece una buena plataforma y mecanismos que facilitan un desarrollo fácil desde el punto de vista educacional y comercial.

La aplicación de técnicas orientadas a objeto mediante la utilización de frameworks, proporciona numerosas ventajas respecto al desarrollo monolítico llevado a cabo en el procesamiento tradicional. Para probar estas ventajas, se ha diseñado e implementado el prototipo descrito en los capítulos 9 y 10.

CAPÍTULO 9

DISEÑO DE UN PROTOTIPO PARA EL SISTEMA O2C2

En el capítulo 8 se ha descrito una arquitectura para el sistema generador de procesadores de lenguajes y se han especificado las propiedades básicas del núcleo del sistema. En dicho núcleo se han identificado un conjunto de bloques encargados de proporcionar las distintas funcionalidades del sistema, finalizando con un resumen de las características ofrecidas por la arquitectura.

En este capítulo se describe el diseño y la implementación, en términos generales, de un prototipo para el sistema O2C2 propuesto [BLI+00a, LCL+01e].

9.1 Introducción

El prototipo aquí presentado fue desarrollado como Proyecto Fin de Carrera de la Escuela Superior de Ingenieros Industriales e Ingenieros Informáticos de la Universidad de Oviedo. Su implementación se basa en el modelo descrito por la arquitectura presentada en el capítulo 8. Sus propiedades y características son aplicadas de forma coherente para demostrar que su implementación real es factible. La descripción completa del mismo se puede consultar en [Bas00].

El principal objetivo del sistema O2C2 es el de ayudar, en la mayor medida de lo posible, a los desarrolladores de procesadores de lenguajes, automatizando las tareas que forman parte del proceso de construcción, aplicando para ello técnicas orientadas a objetos mediante el uso de framewoks.

Para conseguir esta finalidad se parte de la generación de un conjunto de clases por parte del prototipo, a partir de las especificaciones léxicas y sintácticas del lenguaje. Las clases generadas por el prototipo se organizan alrededor de un framework que representa un AST (Abstract Syntax Tree o Árbol sintáctico abstracto). Estos ASTs contendrán nodos en los que se va ha realizar un control estricto de tipos. Esto significa que la información que contienen esos nodos será siempre coherente ayudando a reducir los esfuerzos necesarios para el proceso de depurado cuando se produzcan errores.

9.2 Plataforma de desarrollo

Para la implementación del prototipo se ha elegido la plataforma Java. El elemento fundamental del éxito de esta plataforma es la asociación existente entre el lenguaje y la máquina virtual creada para darle soporte.

La utilización de un lenguaje de programación cercano al que usa habitualmente el usuario junto con la flexibilidad y portabilidad proporcionada por la máquina abstracta, establecen un punto de partida adecuado para la implementación de las propiedades del sistema O2C2.

9.3 Descripción general del prototipo

Se ha diseñado un prototipo, formado por un conjunto de subsistemas agrupados en paquetes, que permiten generar tanto sencillos analizadores sintácticos como completos procesadores de lenguaje.

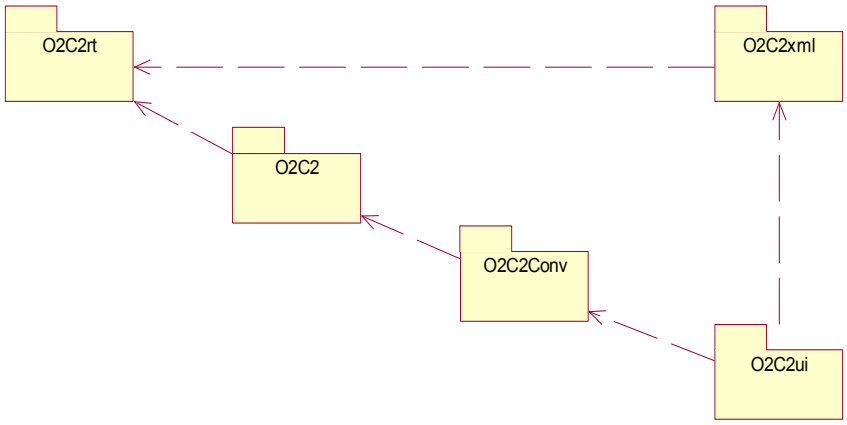


Figura 9.1 Relacción de dependencia entre los paquetes del prototipo.

Con el objetivo de separar de forma tanto conceptual como física los distintos subsistemas que conforman el prototipo se han establecido una serie de paquetes que agruparan a las distintas clases (*Figura 9.1*). En Java los paquetes permiten agrupar clases orientadas a un mismo propósito y será este tipo de paquetes los que serán tenidos en cuenta y utilizados por el prototipo.

9.3.1 Descripción de los Frameworks

Los frameworks que proporciona para procesar un determinado lenguaje se dividen en dos categorías como se apuntó en la sección 8.3.1.

- Por un lado existirá un framework encargado de proporcionar toda la funcionalidad necesaria que es independiente del lenguaje hacia el que va dirigido. Este framework será de tipo *genérico* y podrá ser utilizado heredando de las clases que lo componen y haciendo que las clases derivadas añadan el trabajo necesario para que la solución abstracta que

proporcione, se convierta en una solución válida para un problema concreto.

- En la segunda categoría se incluyen los de tipo *específico* y serán generados específicamente para un determinado lenguaje. Las clases que lo integran se utilizan por composición, haciendo referencia a estas clases como piezas utilizables en la solución de un problema.

De los paquetes representados en el diagrama de la *figura 9.1*, **O2C2rt** y **O2C2xml** pertenecen a la primera categoría, son los únicos que no están diseñados para ser utilizados directamente por el desarrollador. Su función es proporcionar apoyo al resto de los paquetes que sí van a ser utilizados por el usuario final.

Los paquetes restantes **O2C2**, **O2C2Conv** y **O2C2ui** se engloban en la segunda categoría. Contienen herramientas capaces de generar procesadores de lenguajes completando previamente el framework genérico proporcionado por O2C2rt para que se adapte perfectamente a las especificaciones del lenguaje.

A continuación se resumen las características más importantes de cada uno de los paquetes que integran y dan soporte a la funcionalidad básica del prototipo.

9.3.2 Paquete O2C2rt

En el paquete O2C2rt (*O2C2 RunTime*) se integra el conjunto de todas las clases que serán de utilidad para el resto de los subsistemas que puedan o no participar directamente en el proceso de análisis, así como la jerarquía principal de cualquier framework que genere el prototipo O2C2.

9.3.3 Paquete O2C2Conv

El paquete O2C2Conv contiene lo que podríamos denominar la herramienta principal encargada de generar los procesadores de lenguajes. Este paquete contiene las clases que permiten usar el prototipo a través de la aplicación de un **sistema de anclajes** (este sistema se describe en la sección 10.3).

Funcionalidad

A partir de las especificaciones léxicas y sintácticas proporcionadas por el usuario, se completa el framework genérico proporcionado por el paquete O2C2rt. Se generan también las clases necesarias para obtener un analizador sintáctico que se integre bien con el analizador léxico y con las clases que pueda proporcionar el usuario para tratar el análisis semántico y la generación de código intermedio.

Metalenguaje

La especificación que O2C2Conv espera a la entrada es particularmente sencilla puesto que se limita a incluir la especificación gramatical del lenguaje que se ha de reconocer en notación EBNF. La integración con el analizador léxico es

también sencilla, se limita a la necesidad de implementar una interfaz con un *método* conocido como **yylex()**.

Mecanismo de integración

El mecanismo de integración del analizador generado por el sistema con las clases que proporciona el usuario se basa en el modelo de **delegación de eventos.**

En O2C2Conv, las gramáticas especificadas por el usuario vienen acompañadas de **enlaces** (*anclas*). Los enlaces son etiquetas que se colocan a la derecha de un fragmento gramatical, indicando que el usuario está interesado en captar eventos sintácticos que se produzcan como consecuencia de la identificación de dicha estructura gramatical a la entrada del analizador.

Junto con las clases que se generarían a partir de la gramática, se generarán otras correspondientes a los enlaces. Será a estas clases a las que el usuario tendrá que registrar las que él mismo construya.

El modelo de **delegación de eventos** es flexible, potente y fácil de usar. Mediante este mecanismo, el usuario crea clases que serán invocadas por el analizador en el momento oportuno, siempre y cuando estas clases se hayan registrado ante el analizador sintáctico.

La **flexibilidad** de este enfoque supone diversas ventajas para el usuario.

- Un objeto de usuario puede registrarse o desregistrarse en tiempo de ejecución en función de las circunstancias del proceso de análisis.

- Una gramática puede ser utilizada con conjuntos de clases de usuario diferentes.

- Los compiladores que trabajan con varias pasadas pueden ser desarrollados con gran facilidad simplemente registrando y desregistrando clases.

- Un grupo de clases del usuario que trabajan con una estructura sintáctica determinada puede ser reutilizada casi sin esfuerzo. Esto significa que si una o más clases creadas por el usuario se encargan de la clásica estructura sintáctica "*if-then-else*", entonces esas clases pueden ser reutilizadas completamente en cualquier lenguaje que incluya dicha estructura sintáctica.

9.3.4 Paquete O2C2

El paquete O2C2 es otra herramienta integrante que se utilizó para la construcción de la versión inicial del prototipo. De ahí que lleve el mismo nombre dado al prototipo.

Funcionalidad

Ofrece la misma funcionalidad que el paquete O2C2Conv (descrito en la sección anterior) con respecto a la facilidad de integración con el análisis léxico, el tipo de analizador que genera o la potencia del mecanismo de delegación de eventos. La diferencia radica en la forma en que se subscriben las clases creadas por el usuario, para encargarse del análisis semántico ante el analizador sintáctico y para que éste las avise en el momento exacto en el que sean necesarias.

Metalenguaje

La especificación que O2C2 necesita, se limita también a incluir la especificación gramatical del lenguaje que se ha de reconocer en notación EBNF y la especificación léxica proporcionada con la implementación de la interfaz correspondiente a través del método **yylex()**.

Mecanismo de integración

El mecanismo de integración del analizador generado por el sistema con las clases que proporciona el usuario, también se basa en el modelo de **delegación de eventos**.

La integración se realiza mediante la selección por parte del usuario de la correspondiente sección de la gramática para la que esté diseñada la clase que ha creado. Esta sección gramatical dará lugar a una clase sintáctica dentro del framework que generará el sistema. A esta clase se le asigna un nombre de forma automática a partir de la composición de varias palabras. El nombre asignado estará en función del tipo de elementos sintácticos que componen dicha sección, un no terminal, una producción o una secuencia de elementos. A esta clase sintáctica es a la que ha de subscribirse la clase semántica del usuario

Este paquete aún se mantiene disponible en el sistema, aunque es necesario señalar que es más difícil de utilizar por un usuario que no conozca bien los aspectos técnicos del sistema. Obtener el nombre de la clase sintáctica a la que ha de suscribirse la clase semántica del usuario es una tarea laboriosa, que puede dar lugar a cometer numerosos errores y dificulta la realización de posibles modificaciones en la gramática inicial.

9.3.5 Paquete O2C2ui

Este paquete contiene todas las clases que conforman la interfaz gráfica de usuario y permite integrar desde una sola herramienta los análisis sintáctico, léxico, semántico y la generación de código intermedio.

Funcionalidad

La funcionalidad que presenta esta herramienta es triple:

- **Facilitar la labor del usuario** no experimentado, al ser éste asistido por una herramienta visual intuitiva.

- **Proporcionar un medio adecuado** para obtener de forma rápida y sencilla las especificaciones léxicas y sintácticas de los lenguajes.

- **Integrar todos los módulos básicos** del sistema, que son necesarios para poder crear procesadores de lenguaje de una forma sencilla y fácil de usar.

9.3.6 Paquete O2C2xml

Este paquete contiene las clases que integran el procesador de lenguaje capaz de reconocer los ficheros de especificaciones de los proyecto usados en O2C2ui y que están expresados en el lenguaje de marcas XML.

9.4 El framework principal

El desarrollo de este prototipo se ha centrado en la construcción de un sistema capaz de *generar frameworks* para proporcionar al usuario la funcionalidad básica que necesita en el desarrollo de cualquier procesador de lenguaje. .

El conjunto de todas las clases necesarias para la construcción de estos procesadores de lenguaje, participen o no directamente en el proceso de análisis, se incluyen en el paquete conocido como **O2C2RT** (O2C2 RunTime). Se encargará de ser la base sobre la que se construyan posteriormente frameworks específicos para una gramática determinada.

Cada vez que el prototipo crea un nuevo framework no necesita partir de cero puesto que el framework principal (O2C2RT) es independiente del lenguaje que se esté considerando, por tanto, se parte siempre del mismo. Esto nos permite establecer un mayor nivel de consistencia entre los procesadores generados de los diferentes lenguajes.

9.5 El núcleo del prototipo

El núcleo del prototipo está formado por el conjunto de clases de O2C2RT que sí toman parte activa en el proceso de análisis.

Es necesario considerar también algunas clases que aparecerán siempre que se quiera construir un procesador de lenguaje, y que sin embargo, no van a toman parte activa en el proceso de análisis. Estas clases, como por ejemplo las encargadas del tratamiento y recuperación de errores, serán también proporcionadas por el sistema.

Partiendo de la base de que las gramáticas de los distintos lenguajes, por extrañas y diferentes que sean, han de estar compuestas por los mismos bloques básicos: elementos gramaticales (*terminales* o *no terminales*) y composiciones de los mismos, como producciones o repeticiones, será posible abstraer dichas similitudes en el núcleo del sistema.

9.5.1 Estructura de las gramáticas

La propia estructura de las gramáticas que se utilizan de manera habitual para describir los diferentes lenguajes, se puede representar utilizando la notación EBNF. Esta forma de representación, puede ser vista como una **metagramática** que describe la estructura de todas las gramáticas que va a tratar el prototipo. Su definición es la siguiente:

```
<Gramatica> ::= <Regla> {<Regla>};
<Regla> ::= NOTERM [TERM] EQ <Producciones> PUNTOYCOMA;
<Producciones> ::= LITERAL <Seq> <Otras_prods>;
<Producciones> ::= <Vacio>;
<Otras_prods> ::= OPCION <Producciones>;
<Otras_prods> ::= <Vacio>;
<Seq>::=  {<Expr>} ;
<Expr>::=  LLAVEI <Seq> LLAVED;
<Expr> ::= CORCHETEI <Seq> CORCHETED;
<Expr> ::= PARI <Seq> PARD;
<Expr> ::= <Factor>;
<Factor>::= NOTERM;
<Factor> ::= TERM;
<Vacio> ::= ;
```

Esta descripción *metagramatical* ha sido extraída del analizador sintáctico incluido en el sistema. Cualquier gramática, por diferente que sea el lenguaje que describa, va a seguir esta estructura.

De forma general, observando la descripción anterior, se puede apreciar que una gramática está compuesta por elementos básicos de dos tipos: elementos *terminales* y elementos *no terminales*. Los elementos *terminales* son los denominados *tokens*, y son proporcionados por el analizador léxico. Los elementos *no terminales* pueden tener varias definiciones o producciones. Cada una de estas producciones es una secuencia de elementos *terminales* o *no terminales*. En ocasiones, la producción de un *no terminal* incluye subsecuencias de elementos *terminales* y/o *no terminales* con propiedades diferentes. Si la subsecuencia esta rodeada de corchetes, eso significa que la aparición de los elementos que describe esa subsecuencia es opcional. Por otra parte, si estuviera entre llaves, eso significaría que los elementos que representa pueden aparecer más de una vez en un programa escrito utilizando el lenguaje cuya gramática se está considerando.

9.5.2 Estructura del núcleo

A partir de la idea comentada anteriormente, se diseñó un framework genérico, que va a ser la base de cualquier framework generado por el prototipo al portar los elementos comunes a cualquier procesador de lenguaje construido a partir de la descripción de una gramática.

La jerarquía de clases que lo representa en notación UML es el que se muestra en la *figura 9.2*.

A excepción de la clase **Node**, todas las clases que participan en esta jerarquía han sido pensadas para trabajar con los diferentes elementos que pudieran aparecer en una gramática que haya sido descrita utilizando la notación EBNF. Por tanto, estas clases se encargan de representar una sección gramatical. **NodeChoice** representa los elementos *no terminales*, **NodeSeq** representa las producciones de los *no terminales*, **NodeToken** representa los elementos *terminales*, **NodeSeqOptional** representa las repeticiones múltiples de elementos gramaticales y **NodeOptional** representa los elementos de aparición opcional. A partir de estos elementos es posible construir cualquier analizador.

9.5.2.1 Descripción detallada de la jerarquía de clases

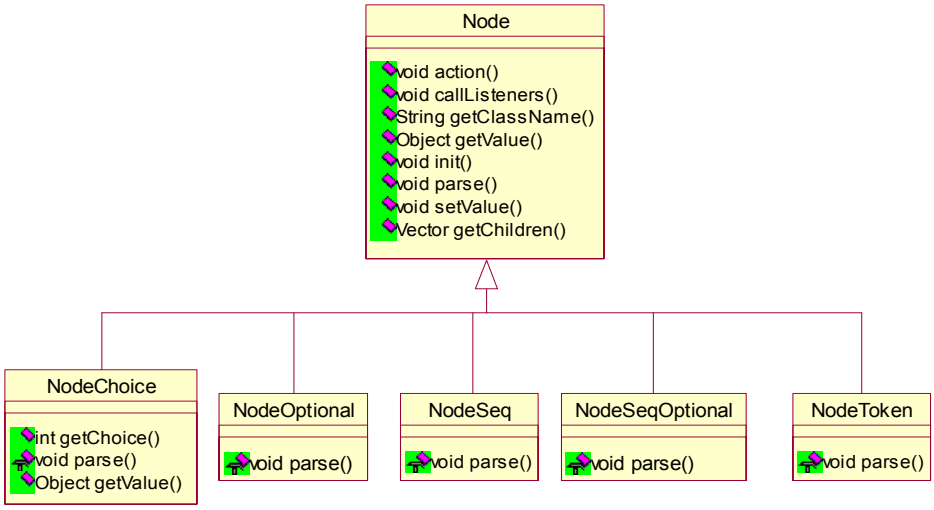


Figura 9.2 Diagrama de clases del núcleo del sistema

Clase Node

Node es la clase que representa la raíz de cualquier framework generado por el sistema O2C2. Cuando se ejecuta un procesador de lenguaje generado por el sistema con un texto fuente de entrada, se crea un árbol de derivación AST. Todos los elementos de la gramática derivan de la clase Node y se convierten en nodos del árbol (los símbolos *terminales* en nodos hojas y los símbolos *no terminales* en nodos internos). Esta clase Node también posee métodos que son comunes a cualquier nodo del AST generado.

Clase NodeChoice

Una subclase de **NodeChoice** es creada para tratar cada una de las producciones que pueda tener un elemento no terminal en la gramática. Su nombre proviene del hecho de que cuando un analizador está tratando un símbolo no terminal tiene que elegir entre las diferentes producciones que tiene asociadas.

Clase NodeOptional

La clase **NodeOptional** representa los elementos de aparición opcional. Es la superclase de todas las clases que tratan elementos de la gramática cuya aparición en los programas que usen el lenguaje descrito por dicha gramática no es obligatoria. Esto significa que algunos elementos gramaticales pueden ser utilizados por algunos programas y pueden no aparecer en otros y aun así, ambos tipos de programas pueden ser descritos por la misma gramática.

Clase NodeSeq

Las subclases que descienden de **NodeSeq** representan secuencias de elementos de una gramática. Son las producciones de los diferentes símbolos *no terminales* y pueden ser elementos *terminales*, *no terminales* o elementos EBNF.

Clase NodeSeqOptional

La clase **NodeSeqOptional** representa las repeticiones múltiples de elementos gramaticales. Parte de un concepto muy similar a la clase NodeSeq, pero en esta ocasión, la existencia de la secuencia de elementos es únicamente opcional.

Clase NodeToken

La clase **NodeToken** es la única clase de esta jerarquía que no tiene subclases. Su misión es obtener de la entrada o del analizador léxico los tokens que constituyen las hojas del árbol de derivación.

Como se puede observar en el diagrama de clases de la *figura 9.2*, el método que tienen todas las clases, y que va a ser diferente en cada una de ellas, es el método **parse()**. Este método describe cómo se analizará la sección gramatical de la que se encarga un nodo determinado, así como de continuar el análisis a partir del punto en el que se había dejado.

9.6 Las clases auxiliares

El paquete O2C2rt, que contiene la funcionalidad básica y no adaptada a un lenguaje determinado, incluye también una serie de clases que permiten hacer de interfaz entre este núcleo y el resto de las clases del prototipo, que sí se adaptan a un lenguaje determinado.

Para cada una de las clases que interactúan con el núcleo se define una interfaz, lo cual significa que las clases que se creen posteriormente, tendrán que implementar dichas interfaces.

Descripción de las interfaces

- La interfaz **Parser** es la clase que utiliza el núcleo del sistema para manejar la clase principal de cualquier analizador sintáctico. Esta clase se encarga de inicializar objetos antes de comenzar el análisis sintáctico, incluyendo también un objeto de la clase del analizador léxico. En lo que al núcleo del sistema se refiere, la interfaz parser tendrá que ser implementada por todos los procesadores de lenguajes que sean generados por el sistema. Obliga a que todos los *parsers* tengan una clase `lexer` que se encargue de obtener los elementos léxicos y a implementar el método `parse()` para inicializar el proceso de análisis sintáctico.

- La interfaz **Lexer** representa para el núcleo del sistema O2C2 al analizador léxico. Cuando el sistema haya construido un analizador funcional, es necesario disponer de un analizador léxico que se encargue de enviar a este analizador sintáctico los componentes léxicos. Obliga a implementar el **método** `yylex()` que se encargua de obtener los tokens de la entrada en forma de elementos de la clase **Symbol**.

  La clase **Symbol** permite representar la estructura de los tokens que se van encontrando durante la fase de análisis. Esta clase representa el valor semántico, el tipo de token y su posición dentro del texto de entrada.

- La interfaz **Listener** representa a las clases que ha tenido que crear el usuario para implementar el código de acción semántico que está asociado a una sección de la gramática en concreto. Mediante esta interfaz, las clases del núcleo de O2C2 saben a dónde llamar cuando se ha reconocido parte de la entrada y se reduce una expresión gramatical del analizador sintáctico.

  Es la interfaz que tienen que implementar todas las clases que pretenden ser alertadas cuando se haya producido un evento sintáctico durante el análisis. Solo obliga a implementar el **método** `performAction(Node)` que será invocado cada vez que se produzca el evento sintáctico para el que se ha subscrito la clase. Para su mejor comprensión re se recomienda ver la sección 10.10 en la que describe el sistema de anclajes.

- La interfaz **ErrorListener**, es muy parecida a la interfaz Listener comentada anteriormente, con la excepción de que es invocada, no cuando se reduce una expresión gramatical, sino cuando se produce un error en la entrada.

  Es la interfaz que deben implementar las clases que pretendan ser alertadas cuando se produce un error durante el proceso de análisis. El **método** que se debe implementar es `treatError(MismatchedTokenException)`.

CAPÍTULO 10

IMPLEMENTACIÓN DEL PROTOTIPO

Para desarrollar el prototipo completo, se ha partido de una versión inicial elaborada a partir de herramientas externas, con el objetivo de ser una versión temporal del prototipo. Este desarrollo inicial fue utilizado posteriormente para construir el prototipo final aplicando la técnica conocida como *Bootstraping*.

En este capítulo se describirán los pasos que se han seguido para su construcción y las ampliaciones y mejoras llevadas a cabo para dotar al prototipo de la funcionalidad del sistema O2C2 descrita en el capítulo 8 [BLI+00b].

10.1 Introducción

Parte de las necesidades básicas que tiene que cubrir la versión inicial del sistema incluyen la de dar soporte a un analizador léxico y sintáctico para el tratamiento de las especificaciones gramaticales en notación EBNF. Estos analizadores fueron generados haciendo uso de herramientas externas.

10.2 El analizador léxico

Para la construcción del analizador léxico se ha utilizado la herramienta JFLEX (Java Fast LEXical analyzer) [Jfl01] siendo su funcionamiento similar al de otras muchas otras herramientas utilizadas para el mismo propósito. Necesita un fichero de especificaciones léxicas divididas en tres secciones diferentes.

En la **primera sección** se incluyen una serie de opciones que permiten parametrizar el analizador generado y cuyo contenido es el siguiente:

```
%class lexer
%extends o2c2.symb
%implements o2c2rt.Lexer
%unicode
%line
%column
%type o2c2rt.Symbol
```

Esta especificación indica a Jflex que la clase que va a generar se llama **lexer**. Es una subclase de la clase **symb** (contiene un conjunto de constantes numéricas que son utilizadas en la comunicación entre el analizador léxico y el analizador sintáctico) e implementa la interfaz **Lexer**. Esto implica que el analizador sintáctico puede conocer a la clase `lexer` y saber que métodos puede usar.

La especificación también indica que el analizador deberá entender caracteres que utilicen el **sistema Unicode**, que el valor que se pasa al analizador sintáctico va a ser una instancia de `Symbol`, y que se registrarán la línea y la columna de cada token que se encuentre.

La **segunda sección** se encarga de realizar algunas definiciones léxicas que se refieren a asociaciones entre variables definidas por el usuario con expresiones regulares.

```
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
COMENTARIO = {TraditionalComment} | {EndOfLineComment} |
             {DocumentationComment}
TraditionalComment = "/*" [^*] {CommentContent} \*+ "/"
EndOfLineComment = "//" {InputCharacter}* {LineTerminator}
DocumentationComment = "/**" {CommentContent} \*+ "/"
CommentContent = ( [^*] | \*+[^*/] )*
ALPHA = [A-Za-z+*\-/.]
DIGIT = [0-9]
WHITE_SPACE_CHAR = [\n\r\ \t\b\012]
ALPHANUM = {ALPHA}({ALPHA}|{DIGIT}|_)*
```

La **tercera sección** establece la asociación entre los elementos que el analizador léxico va a encontrarse en la entrada, y las acciones correspondientes cuando se encuentren dichos elementos. En el caso que nos ocupa, lo que interesa no es hacer algo en concreto con los *tokens* que se encuentren en la entrada, sino pasárselos al analizador sintáctico para que este haga lo que considere oportuno.

```
"\""{ALPHANUM}"\""      {return symbol
          (symb.LITERAL,yytext().substring(1,yytext().length()-1));}
"<"{ALPHANUM}">"        {return symbol
          (symb.NOTERM,yytext().substring(1,yytext().length()-1));}
"::="                   {return symbol (symb.EQ);}
"|"                     {return symbol (symb.OPCION);}
({ALPHANUM}|{DIGIT})    {return symbol (symb.TERM,yytext());}
"("                     {return symbol (symb.PARI);}
")"                     {return symbol (symb.PARD);}
"["                     {return symbol (symb.CORCHETEI);}
"]"                     {return symbol (symb.CORCHETED);}
"{"                     {return symbol (symb.LLAVEI);}
"}"                     {return symbol (symb.LLAVED);}
";"                     {return symbol (symb.PUNTOYCOMA);}
{WHITE_SPACE_CHAR}*     { /* ignorado */ }
```

```
{COMENTARIO}    { /* ignorado */ }
.|\n            { throw new RuntimeException ("Caracter
                  ilegal \""+yytext()+"\" en la linea "+yyline+",
                  columna "+yycolumn); }
```

10.3 El analizador sintáctico

Partiendo de la base de que el objetivo del prototipo es desarrollar procesadores de lenguajes, se necesita internamente un analizador sintáctico que realice parte del trabajo necesario.

Para dotar al prototipo de funcionalidad sintáctica se utilizó un generador de analizadores sintácticos ya existente, la herramienta CUP (Constructor of Useful Parsers) [Hud97]. En este caso, la entrada que se debe proporcionar a CUP será la especificación (similar a la notación BNF) de *la gramática del analizador sintáctico* que se quiere generar. Esta entrada está formada por un complejo fichero dividido en tres secciones.

En la **primera sección** se incluyen definiciones previas de funciones, clases y variables que se utilizarán durante el proceso de análisis. Su contenido es el siguiente:

```
package gplold;

import java_cup.runtime.*;
import java.util.Hashtable;
import java.util.Vector;
import gplold.Simbolo;

parser code {:
    lexer l;
    Hashtable tablaSimb;
    Hashtable dataType;
    Vector terminales, noterminales;
    String primerNT = "";
                    // El primer no terminal de la gramatica
     public parser (lexer scanner, Hashtable tabla,
                    Hashtable dT, Vector terms, Vector noterms) {
       this.l = scanner;
       tablaSimb = tabla;
       dataType = dT;
       terminales = terms;
       noterminales = noterms;
   }
:};
scan with {: return l.yylex(); :};
```

Como se puede apreciar, se especifica qué estructura de datos se va a utilizar como tabla de símbolos, cual va a ser el constructor del analizador y qué código se va a emplear para obtener los tokens del analizador léxico.

La **segunda sección** se corresponde con la definición de la gramática propiamente dicha. Primero se incluye una lista de los elementos *terminales* y *no terminales*. De forma opcional, se especifica el tipo de datos que se asocia al valor semántico de estos elementos.

```
  /* Los elementos terminales (tokens devueltos por el léxico) */
terminal   EQ, OPCION, TERM, NO_TERM, PARI, PARD, LLAVEI, LLAVED,
CORCHETEI, CORCHETED, PUNTOYCOMA, LITERAL;

  /* Los elementos no terminales */
non terminal      gramatica, regla;
non terminal Vector  seq, producciones;
non terminal Simbolo expr, factor;
```

La definición de la *gramática* que describe a la **metagramática** es la siguiente:

```
gramatica ::= regla
 | gramatica regla
 ;
regla ::= NO_TERM:e1 EQ producciones:e2 PUNTOYCOMA
 | NO_TERM:e1 TERM:e3 EQ producciones:e2 PUNTOYCOMA
 | NO_TERM:e1 EQ PUNTOYCOMA
 ;
producciones ::= LITERAL:e1 seq:e2
 | producciones:e1 OPCION LITERAL:e2 seq:e3
 ;
seq ::= expr:e1
 | seq:e1 expr:e2
 ;
expr ::= LLAVEI seq:e1 LLAVED
 | CORCHETEI seq:e1 CORCHETED
 | PARI expr:e1 PARD
 | factor:e1
 ;
factor ::= NO_TERM:e1
 | TERM:e1
 ;
```

Con esta definición y el código de acción semántica, junto a las distintas reglas de la gramática, CUP genera un **analizador sintáctico en una sola clase**. Este analizador sintáctico es capaz de reconocer gramáticas expresadas en notación EBNF. Se ha omitido aquí el código de acción para resaltar la estructura de la gramática y ofrecer así una mayor claridad.

10.4 Aplicación de la técnica Bootstrapping

Como ya se ha comentado anteriormente, el prototipo utiliza internamente un analizador sintáctico generado por CUP. Puesto que el objetivo es construir procesadores de lenguajes está justificado que debe ser el propio prototipo el que desarrolle el generador de analizadores sintácticos interno.

La aplicación de la técnica Bootstrapping se basa en utilizar la facilidades que ofrece la versión inicial del prototipo para poder tratarse a sí mismo y regenerarse.

En la *figura 10.1* se muestran los pasos seguidos para obtener un generador de analizadores sintácticos construido por el prototipo inicial.

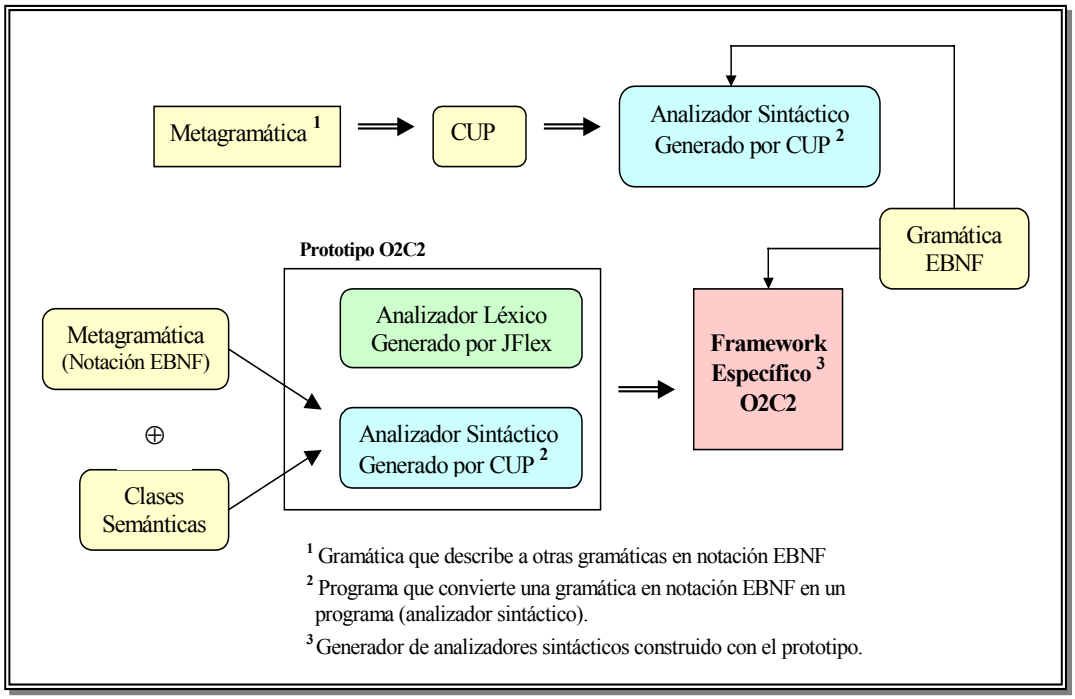


Figura 10.1 Construcción del generador de analizadores sintácticos O2C2.

En esta versión inicial, se construyó un generador de analizadores sintácticos descendentes LL(1) denominado O2C2, capaz de reconocer gramáticas expresadas en notación EBNF y extraer de ellas toda la información necesaria para construir los analizadores sintácticos de los lenguajes que describan. Este generador es funcionalmente equivalente y puede sustituir a la herramienta CUP.

Para desarrollar el generador de analizadores sintácticos O2C2 se utilizó la definición de una gramática en notación EBNF (para describir a la metagramática) algo diferente a la utilizada con la herramienta CUP. Esta gramática mantiene la misma estructura que descrita en la sección 9.5.1.

```
<Gramatica>     ::= <Regla> {<Regla>};
<Regla>         ::= NOTERM [TERM] EQ <Producciones> PUNTOYCOMA;
```

```
<Producciones> ::= LITERAL <Seq> <Otras_prods>;
<Producciones> ::= <Vacio>;
<Otras_prods>  ::= OPCION <Producciones>;
<Otras_prods>  ::= <Vacio>;
<Seq>          ::= {<Expr>} ;
<Expr>         ::= LLAVEI <Seq> LLAVED;
<Expr>         ::= CORCHETEI <Seq> CORCHETED;
<Expr>         ::= PARI <Seq> PARD;
<Expr>         ::= <Factor>;
<Factor>       ::= NOTERM;
<Factor>       ::= TERM;
<Vacio>        ::= ;
```

El fichero de entrada sólo contiene esta especificación gramatical y no es necesario detallar cuáles son los símbolos *terminales*, los símbolos *no terminales* o la precedencia de operadores.

Con esta gramática y unas clases creadas expresamente para dar un contenido semántico, se obtuvo el generador de analizadores sintácticos deseado. Este generador es un framework específico y representa al paquete O2C2 descrito en la sección 9.3.4. El siguiente diagrama de clases describe la arquitectura y diseño del subsistema O2C2.

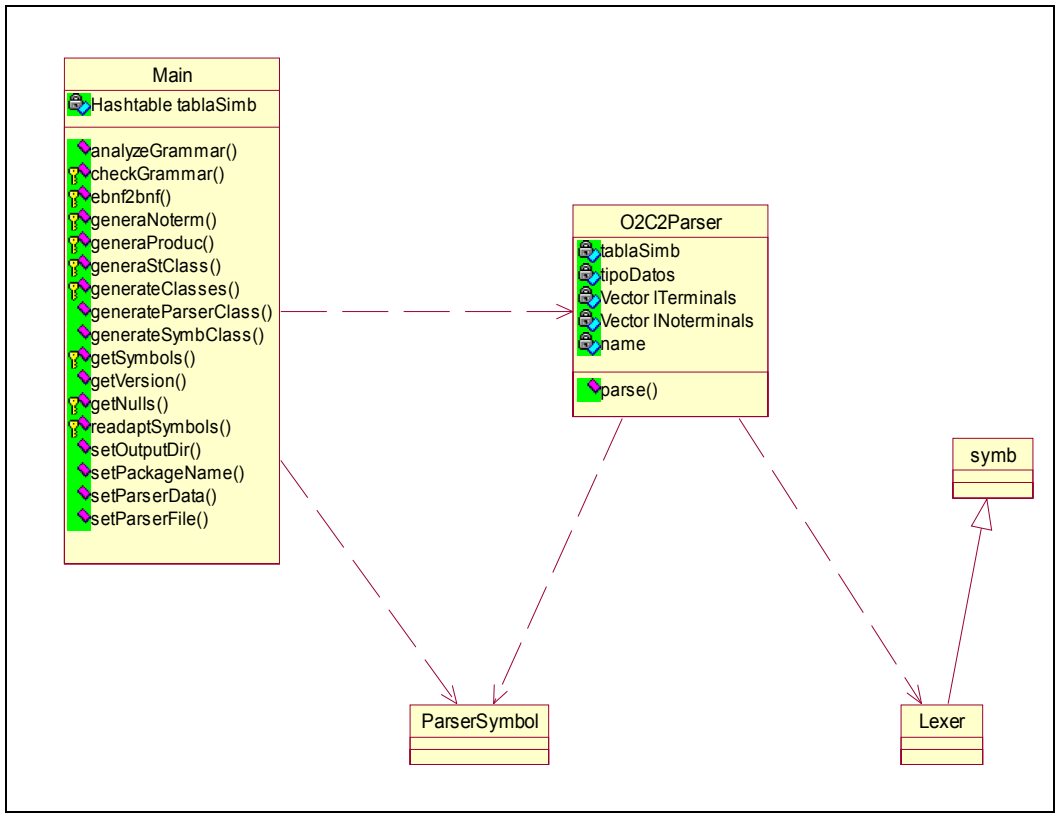


Figura 10.2 Diagrama de clases del subsistema O2C2

10.5 Construcción de un procesador de lenguaje

La versión inicial del prototipo ya es operativa, y se puede ver como un sistema generador de procesadores de lenguajes. En este apartado se tratará de mostrar cómo opera internamente el sistema para la *construcción del analizador sintáctico* únicamente.

Para almacenar la información mínima necesaria sobre la gramática que el sistema está procesando en un momento dado, se necesita una estructura a la que se ha denominado tabla de símbolos. Se define como una estructura de datos en la que se almacena información relevante obtenida durante el proceso de análisis. En el prototipo desarrollado, durante la fase de análisis se recoge información sobre la gramática (en notación EBNF) que se le pasa a la entrada, información que será posteriormente utilizada para generar el procesador de lenguaje.

Se ha elegido como estructura de datos para almacenar la información de la gramática una *tabla hash* abierta. Las claves de esta tabla hash son los elementos *no terminales*, y los valores de estas claves son listas de elementos. Cada una de las listas representa el contenido de una producción. y se estructura como una lista de símbolos almacenando dos valores: el nombre y el tipo del símbolo. Hay cuatro tipos de símbolos: *terminales*, *no terminales*, secuencias opcionales y secuencias repetidas. En los dos primeros casos el valor de un símbolo es sólo una cadena de texto, en los dos últimos casos es una lista de elementos como los que acabamos de describir o como los que forman el valor de una producción.

Ejemplo 10.1 *En el siguiente ejemplo se muestra una pequeña gramática y el correspondiente contenido en la tabla hash.*

```
<S> ::= ID | <num> OP <num>;
<num> ::= DIGITO {DIGITO};
```

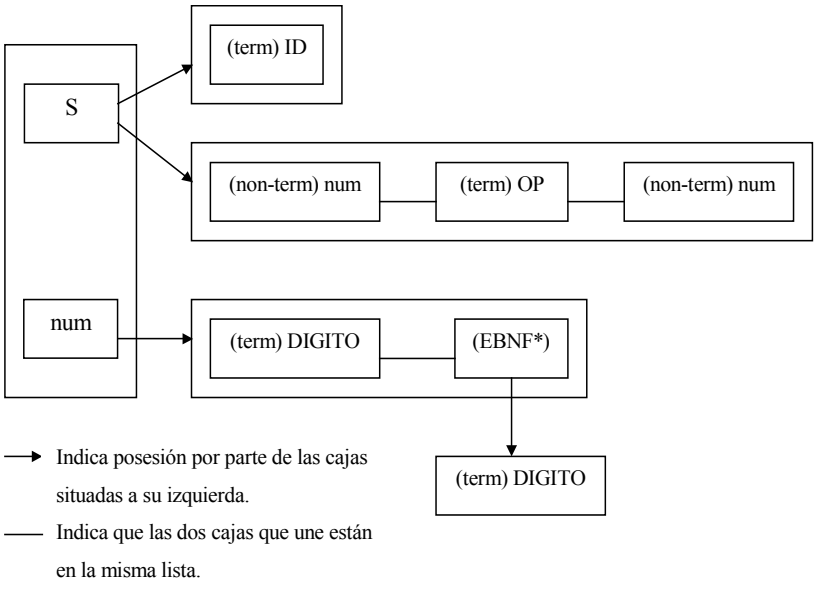


Figura 10.3 Representación gráfica del contenido de la tabla de símbolos

Con la información de la gramática obtenida por la herramienta O2C2 y almacenada en la estructura de datos es posible calcular los símbolos *nulos*, *iniciales* y *seguidores*, a partir de los cuales se obtendrán los símbolos directores. Como base para los algoritmos necesarios se han utilizado los descritos en [App98]. No obstante estos algoritmos se aplican a gramáticas expresadas en notación BNF por lo que fue necesario realizar una pequeña transformación para que pudieran ser utilizados. Se aplicó la siguiente transformación:

1. Para cada elemento con un operador opcional (**?**) se elimina el operador y se crea una nueva producción igual, pero sin el elemento.

2. Para cada elemento con un operador de multiplicidad ({}) se sustituye la secuencia enmarcada en la multiplicidad, que llamaremos x, por un nuevo no terminal que llamaremos t. Ese no terminal quedará definido como $t = x\ t \mid \lambda$.

10.5.1 Generación de frameworks

Una vez calculados los símbolos directores, con la información almacenada en la estructura de datos, si la gramática cumple la condición LL(1) se procede a generar los ficheros que van a componer el analizador sintáctico.

En primer lugar se generan los ficheros del framework específico. Para ello se recorre la estructura de la tabla de símbolos buscando los elementos almacenados.

- Cada uno de los elementos *no terminales* da lugar a una clase que desciende de **`NodeChoice`**.

- Cada producción da lugar a una clase que desciende de **`NodeSeq`**.

- Para cada uno de los elementos de las producciones de los símbolos *no terminales*.

    - Si es un elemento EBNF*, (lo cual significa que contiene una secuencia de elementos que pueden aparecer repetidamente en un punto concreto del código fuente), se genera una clase que desciende de **`NodeSeqOptional`**.

    - Si el elemento es de tipo EBNF? (es una secuencia de aparición opcional) se genera una clase que desciende de **`NodeOptional`**.

    - Si el elemento es un no terminal, entonces no hace falta crear una clase para él, ya se creará cuando se encuentre como elemento principal en la tabla de símbolos.

    - Si el elemento es un terminal, tampoco es necesario tratarlo, puesto que no se va a crear una clase específica para él, sino que se creará una instancia de **`NodeToken`** preparada para leer sólo ese terminal.

Existe un método único para crear estas clases. Este método se invoca pasándole la clase de la que desciende, la clase que se ha de generar, el nombre que va a tener, el conjunto de símbolos directores que va a necesitar (si es necesario) y por último un conjunto de clases que tendrá que instanciar durante el proceso de derivación del analizador, una vez esté operativo.

En segundo lugar se generan las denominadas clases auxiliares necesarias para dar funcionalidad al analizador sintáctico generado.

- La clase **`symb`** sirve de comunicación entre el analizador léxico y el analizador sintáctico. La comunicación entre los dos analizadores se realiza a través del intercambio de los tokens. Se encarga de asignar una constante numérica a cada uno de los símbolos *terminales* de la gramática. También provee un método a partir del cual se puede conocer la cadena de texto asociada a una constante numérica. Se utiliza para proporcionar mensajes de error significativos y explicativos cuando se produce algún fallo.

- La clase **`Parser`** es la clase principal de analizador. Se encarga de la inicialización de todas las clases necesarias para poner en marcha el proceso de análisis. Esta clase puede ser modificada por el usuario para incluir otras clases que implementen la *interfaz listener*.

10.6 Pruebas de validación

Con el propósito de comprobar el buen funcionamiento del generador de procesadores de lenguajes construido, se aplicaron las pruebas de validación sobre un conjunto de gramáticas que recogían diferentes aspectos. De forma adicional a las técnicas de prueba tradicionales, se ha utilizado un proceso de validación consistente en las siguientes etapas:

- **Pruebas de tipo A**. Se realizaron con gramáticas correctas comprobando que no producían errores a la hora de ser tratadas por el sistema O2C2. También se comprobó que se generaban analizadores sintácticos correctos capaces de procesar cualquier entrada descrita por dichas gramáticas.

- **Pruebas de tipo B**. Se realizaron con gramáticas erróneas con el propósito de comprobar que el sistema O2C2 era capaz de darse cuenta de la existencia de ese tipo de errores y señalárselos al usuario.

- **Pruebas de tipo L**. Se realizaron con gramáticas erróneas que no tenían efecto al ser procesadas por el prototipo sino a la hora de ejecutar el procesador de lenguaje generado.

- **Pruebas de tipo C**. Se realizaron con las mismas entradas que se utilizaron en los tests de tipo A. Se prueban una vez más con el propósito de verificar de manera efectiva que los analizadores generados realmente son capaces de reconocer entradas expresadas en la gramática que fue la entrada a su vez del prototipo.

- **Pruebas de tipo D**. Se realizaron con gramáticas correctas para comprobar las capacidades máximas de la implementación. Estas capacidades máximas se refieren, a símbolos *no terminales* con muchas producciones, a secuencias de elementos gramaticales con muchas repeticiones múltiples, y a gramáticas con un elevado número de símbolos *no terminales* y *terminales*.

- **Pruebas de tipo E**. Se realizaron con gramáticas especiales para buscar ambigüedades.

10.7 Conclusiones derivadas de la implementación inicial

A partir de la implementación inicial del prototipo se señalaron diversos puntos estratégicos con el objeto de ampliar y mejorar el desarrollo del sistema final.

1. La creación de clases para albergar el **código de usuario** necesario para llevar a cabo el proceso de análisis semántico es externo al sistema y debe incluir por tanto un mecanismo que facilite al usuario su integración. Se propone el **mecanismo de listeners** combinado con el **sistema de anclajes** descrito en las secciones 10.9 y 10.10 de este capítulo.

2. El prototipo **carece de un mecanismo adecuado de tratamiento y recuperación de errores,** por lo que resulta difícil para el usuario tratar los errores producidos por el propio prototipo, y por el procesador de lenguaje generado.

3. El prototipo **funciona exclusivamente desde la línea de comandos**. Es necesario incorporar un entorno integrado de desarrollo adecuado y conseguir que la funcionalidad del sistema sea proporcionada en forma de API.

4. El entorno integrado de desarrollo debe proporcionar **mecanismos para la obtención del analizador léxico**, generando la clase que implementa la interfaz **Lexer** de forma automática, a partir de la selección por parte del usuario de los tokens necesarios.

En este punto, el prototipo implementa la funcionalidad mínima necesaria para poder convertirse en la base sobre la que se implementará el resto de los componentes del sistema O2C2. Las ampliaciones y mejoras realizadas servirán para solventar los problemas detectados anteriormente y para construir el prototipo completo.

10.8 Creación de clases de usuario

El prototipo permite generar un analizador sintáctico capaz de integrarse fácilmente con un analizador léxico. Con la combinación de ambos analizadores

se obtiene un procesador de lenguaje capaz de reconocer las entradas expresadas en un determinado lenguaje descrito por la gramática proporcionada en la entrada. El usuario es el encargado de aportar el código de acción semántica necesario para completar el procesador de lenguaje generado.

El prototipo genera una clase para cada elemento relevante de la gramática. Una clase para cada no terminal, otra para cada producción y otra para cada elemento EBNF. Todas las clases del framawork específico se crean dentro de un espacio de nombres (*package*) cuyo nombre es asignado por el usuario y que por defecto se llama *parser*. Cada una de estas clases desciende de otras clases que forman parte del framework genérico.

El código de acción semántica asociado a un determinado elemento gramatical se puede incluir en las propias clases generadas por el sistema. Según este criterio, se añadiría en el método `action()` de la clase a la que quisiéramos añadir funcionalidad. Este método tiene tres signaturas diferentes en función de la clase del run-time de la que descienda el elemento gramatical descrito por la clase: `void action ()`, `void action (int)` y `void action (string)`. El run-time del sistema se encarga de invocar la versión correspondiente al tipo de nodo que se este modificando.

Este mecanismo permite añadir código rápidamente y de forma cómoda, pero presenta los inconvenientes mostrados en la sección 8.7.2.

Como se pudo observar en el capítulo 8, el mecanismo de **listeners** es el que mejor se adapta para conseguir los objetivos que persigue el sistema propuesto.

En el siguiente apartado se va a describir por un lado, cómo van a ser las clases que el usuario va a desarrollar para poder aportar al procesador de lenguaje generado este contenido semántico, y por otro lado, establecer cómo va a ser la comunicación entre las clases del framework y las clases que contienen el código de acción semántica.

10.9 Incorporación del mecanismo de listeners

El **mecanismo de listeners** es el medio por el cual un procesador de lenguaje generado por el sistema se pone en contacto con las clases creadas por el usuario. Este mecanismo tiene numerosos paralelismos con el ***modelo de delegación de eventos*** presente en la librería JFC (*Java Foundation Classes*) [Jav01a]. Este modelo de delegación de eventos consiste en separar los componentes visuales (*widgets*) del código encargado de manejar los eventos que se generen en dicho componente. Por ejemplo, una ventana puede ser fuente de muchos eventos, como el evento cambio de tamaño, el evento cerrar ventana, el evento minimizar ventana, etc. Cuando una clase quiere tratar uno de estos eventos, se suscribe ante el objeto ventana como objeto interesado en un evento concreto, por ejemplo cerrar ventana. Cada vez que se pulse el botón de cerrar ventana asociado al objeto, se invocará un método conocido del objeto suscrito para que trate dicho evento.

De igual forma, el sistema O2C2 separa la finalidad del propio analizador del tratamiento de los eventos que tienen lugar cada vez que se identifica una parte de la entrada con una sección de la gramática.

Los eventos son generados por las clases que representan las distintas secciones de la gramática. Cuando se consigue asociar una serie de tokens obtenidos del análisis léxico con una sección de la gramática y dicha sección está representada por una clase del analizador sintáctico, se dice que se ha reducido una expresión. Este hecho es el que genera un evento.

10.9.1 Las clases estáticas

Para comprender mejor como funciona el sistema de suscripciones a eventos gramaticales, se explicará de forma breve y concisa el concepto de clases estáticas.

Como es sabido, las clases son estructuras que unen al tradicional concepto de estructuras de datos, el código necesario para manejar los datos que contiene dicha estructura. Sin embargo, las clases, como cualquier otra estructura genérica de un lenguaje de programación, no están pensadas para ser usadas directamente sino por medio de instancias de las clases que se llaman objetos. Cuando se crea una instancia de una clase, el contenido de todos los datos es exclusivo de esa instancia y por lo tanto cuando se realiza un cambio sobre estos, el resto de las instancias de esa clase no son alteradas.

Sin embargo, en ocasiones es necesario o al menos conveniente, pensar que algunas de las variables de una clase sean comunes no sólo en forma sino en contenido en todas las instancias de una clase. Se entiende entonces que esas variables dejan de ser variables de las instancias y se convierten en variables de la clase, para lo cual se declaran como variables estáticas[1].

Cuando en una clase que contiene variables definidas como estáticas, existe un método que pretende tratar esas variables, dicho método también ha de ser declarado como estático y por lo tanto se convierte en un método de clase. Los métodos de clase pueden ser utilizados sin tener que crear una instancia de la clase para ello.

10.9.2 La suscripción de los listeners

En vez de unir de forma indisoluble el comportamiento a una clase, lo que se realiza en este caso es una suscripción.

[1] La clase System de Java contiene variables y métodos estáticos por lo que no es necesario crear instancias para poder tener acceso a los distintos servicios que ofrece. Esta clase se utiliza por ejemplo para poder visualizar cadenas de texto formateadas por la pantalla. Para ello cuenta con una variable estática conocida como `out`, la cual se utiliza directamente por el programador sin tener que crear una instancia de la clase System para ello.

Cada vez que una clase reduce una expresión, se genera un evento que será escuchado por todas las clases creadas por el usuario interesadas en dicho evento. Estas clases se dice que se suscriben a una determinada clase del analizador y ésta entiende que aquellas son *listeners*. Una misma clase del analizador puede tener registrados varios suscriptores y un *listener* puede estar subscrito a eventos de varias clases generadas por el sistema.

Se consigue de esta forma que el código que añade el usuario esté incluido en las *clases listeners* en vez de incluirlo directamente en las clases generadas por el sistema evitando así los inconvenientes citados anteriormente.

Si una clase creada por el usuario quiere ser avisada cuando se produzcan eventos gramaticales, entonces se subscribe para el tipo de evento que tenga interés y ante la sección gramatical que quiere escuchar.

El sistema O2C2 no obliga al usuario a realizar un tratamiento de la información resultante de cada una de las reducciones gramaticales que se produzcan. Si hay un listener subscrito entonces es la clase del usuario la que se encarga de realizar el tratamiento correspondiente, si no lo hay, la información será almacena convenientemente para que pueda ser utilizada en otros puntos de la gramática.

Para poder **realizar la suscripción,** el usuario debe conocer en primer lugar, ante qué clase tiene que subscribirse. En la *versión inicial* del sistema se necesitaba conocer el nombre de la clase (asignado de forma automática por el sistema) que se encargaba de una determinada sección gramatical, para realizar posteriormente la suscripción, solicitando a esa clase, que avisase cada vez que se reducía una expresión. En la *versión actual* del sistema, con la incorporación del *sistema de anclajes* (descrito en la sección 10.10), resulta fácil conocer ante qué clases se tienen que registrar las clases creadas por el usuario para tratar eventos generados durante el proceso de análisis.

Pero no solamente es necesario conocer la clase, sino que también se necesita conocer en todo momento la instancia de la clase que se encarga de la sección gramatical en la que el usuario está interesado. Esto resulta bastante difícil de conseguir si se considera que cuando el usuario diseña e implementa sus clases, no puede conocer de ninguna manera, como va a funcionar el analizador, qué objetos se instancian y cuándo lo hacen.

La forma de solventar esta dificultad es haciendo que la mayor parte del trabajo de las clases de los *frameworks* generados por el prototipo, no sea trabajo de las instancias de estas clases sino trabajo de las propias clases, basando su funcionamiento en clases estáticas, donde las operaciones que se realizan en una instancia de la clase tienen repercusión en cualquier instancia de la clase que pueda estar activa en un momento determinado.

De esta forma, para poder suscribirse, no es necesario que el usuario conozca una instancia de la clase en concreto, se suscribe ante la propia clase.

10.9.3 La interfaz listener

Para que las clases creadas por el usuario puedan ser invocadas cada vez que se produzca un evento al que el usuario se ha suscrito, son necesarias dos cosas:

1. Que exista una instancia de la clase listener. En el constructor de esta clase se encuentra la llamada al método `addListener()` de la clase a la que se quiere realizar la suscripción.

2. Que el analizador sepa qué hacer con esos objetos, para que reaccionen adecuadamente y puedan realizar tareas semánticas.

El lenguaje Java incluye el concepto de interfaces, que son clases abstractas puramente virtuales. Gracias a las interfaces es posible que una clase se pueda identificar ante cualquier otra como implementadora de una determinada interfaz, debido al sistema de identificación de tipos en tiempo de ejecución incluido en la máquina virtual de Java. Esto les obliga a incluir los métodos definidos en la interfaz.

En el sistema O2C2, es obligatorio que las clases con el código de acción semántica implementen la *interfaz Listener* (del paquete o2c2rt). Esta interfaz obliga a estas clases a incluir el método `performAction (Node n)`. Así, cuando se produce un evento gramatical, el analizador sabe qué método ha de invocar en los objetos que están suscritos.

El parámetro del método (`Node n`) hace referencia al nodo objeto de la clase semántica que invocó dicho método. Esto es importante porque una clase que representa un ancla en la gramática puede tener varios subscriptores (clases semánticas que actúan como listeners) y un mismo listener puede estar subscrito a eventos de varias clases que representan anclas. Con este parámetro se permite distinguirlas.

10.9.4 Interacción entre las clases

En general, los listener proporcionados por el usuario se limitan a implementar dos métodos exclusivamente:

1. El método **constructor**. Necesario para inicializar las estructuras de datos que se van a utilizar, y para realizar la suscripción de la clase, ante la clase del analizador generado por el sistema O2C2 que representa la sección gramatical en la que se está interesado.

2. El método **performAction()**. Necesario para implementar la interfaz listener.

Dentro del método `performAction()` se puede hacer referencia a tres secciones diferentes.

- La primera sección, obtiene una instancia de la clase del analizador a la que se suscribe. No se necesita conocer cuál es la instancia de esa clase, porque los valores que se van a obtener y los métodos que se van a invocar son estáticos y pertenecen a la clase, nunca a la instancia.

- La segunda sección, realiza las diferentes tareas semánticas con la información almacenada en la clase del analizador, que es resultado del proceso de análisis. Para poder hacer uso de esta información, se utilizan variables de la clase del analizador nombradas de la forma `fN`, donde `N` es un número entero que representa el orden del elemento en la sección gramatical.

- La tercera sección realiza la asignación de un valor a la clase como resultado final del proceso.

El diagrama de secuencia de la *figura 10.4* muestra la actuación de un *listener* en colaboración con la clase que suscribe.

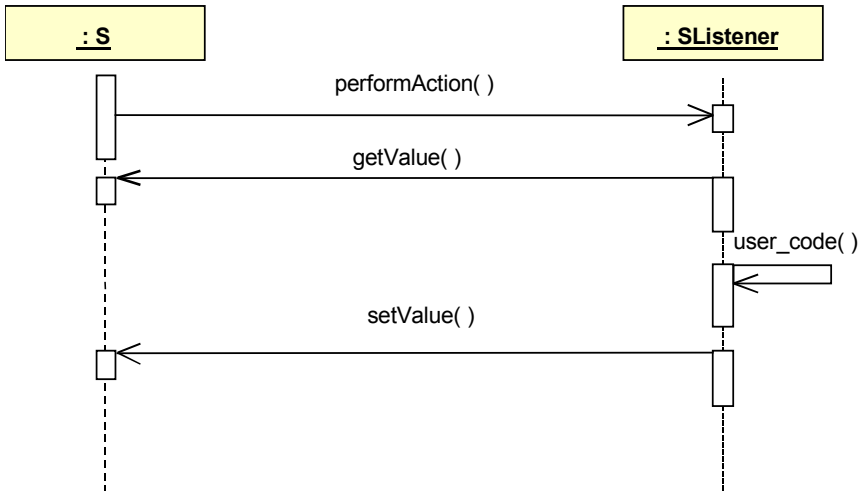


Figura 10.4 Diagrama de colaboración entre un listener y la clase que suscribe

Cuando se produce un evento gramatical generado por la clase `S`, esta avisa a los listeners de que un evento se ha generado usando `performAction()`. El listener pregunta a la clase por valores semánticos obtenidos anteriormente durante el proceso de análisis con `getValue()`. A partir de esa información, el listener efectúa las operaciones que considere oportunas con el código aportado por el usuario `user_code()`. Finalmente, se genera un valor semántico que será pasado a la clase generada por el prototipo con el método `setValue()` para que pueda ser usado en etapas posteriores del análisis.

10.10 Sistema de anclajes

El sistema de anclajes permite insertar dentro de la especificación de la gramática, nombres entrecomillados para denotar secciones de la gramática que,

una vez alcanzadas y reducidas, puedan dar lugar a eventos que el usuario estaría interesado en escuchar.

La incorporación del modelo de anclajes permite al usuario señalar únicamente las partes de la gramática que le interesan y que van a recibir tratamiento semántico. No necesita conocer los nombres que asigna el prototipo a los diferentes elementos gramaticales, puesto que es el usuario el responsable de dar nombre a las secciones que le interesen. Todas las clases que representan a las *anclas* derivan de la clase `NodeChoice`.

10.10.1 Tratamiento de valores semánticos

El sistema de obtención de los valores semánticos asociados a los descendientes en el árbol de derivación (AST) del símbolo no terminal asociado a un ancla determinado, consiste en utilizar un método exclusivo de la clase `NodeChoice` denominado `getValue(int)`. El parámetro indica la posición que ocupa dentro de la especificación gramatical.

Para cambiar los valores semánticos de los nodos del árbol se utiliza el método `setValue()`. Hay que puntualizar en este caso, que el valor que se debe cambiar no es el de la clase que representa el elemento no terminal *ancla* al que está suscrito el usuario, sino el valor de la clase del elemento no terminal que contiene la producción asociada al ancla.

El prototipo conserva los valores semánticos de los distintos nodos, y permite recorrer el árbol sintáctico abstracto hacia arriba con el método `getParent()` y hacia abajo, con el método `getChildren()`, una vez que se vayan reduciendo las diferentes expresiones gramaticales.

10.10.2 Ejemplo sencillo

El objetivo de este apartado es proporcionar un ejemplo sencillo para mostrar la incorporación de los *listeners* al sistema O2C2.

Las especificaciones mostradas a continuación corresponden a una simple calculadora de números enteros.

```
<S>  ::= <num> <op> <num> "operation" ;
<op> ::= + "addition" | - "subtraction" |
         * "multiplication" | / "division" ;<num> ::= DIGIT
{DIGIT} "number" ;
```

La gramática contiene *anclas* asociadas a las distintas producciones indicando de esta forma que existe interés en crear listeners para ellas.

La jerarquía de clases generada por el sistema O2C2 que completaría el núcleo del framework, es la que se describe en el diagrama de la *figura 10.5*. Cada elemento no terminal de la gramática da lugar a una clase que desciende de la

clase `NodeChoice`, para cada producción se genera una clase que desciende de `NodeSeq`, y para los elementos *terminales* que pueden aparecer repetidamente se genera una clase que desciende de `NodeSeqOptional`. Para simplificar el gráfico, no se han incluido las flechas de las clases "*ancla*" derivando de la clase `NodeChoice`.

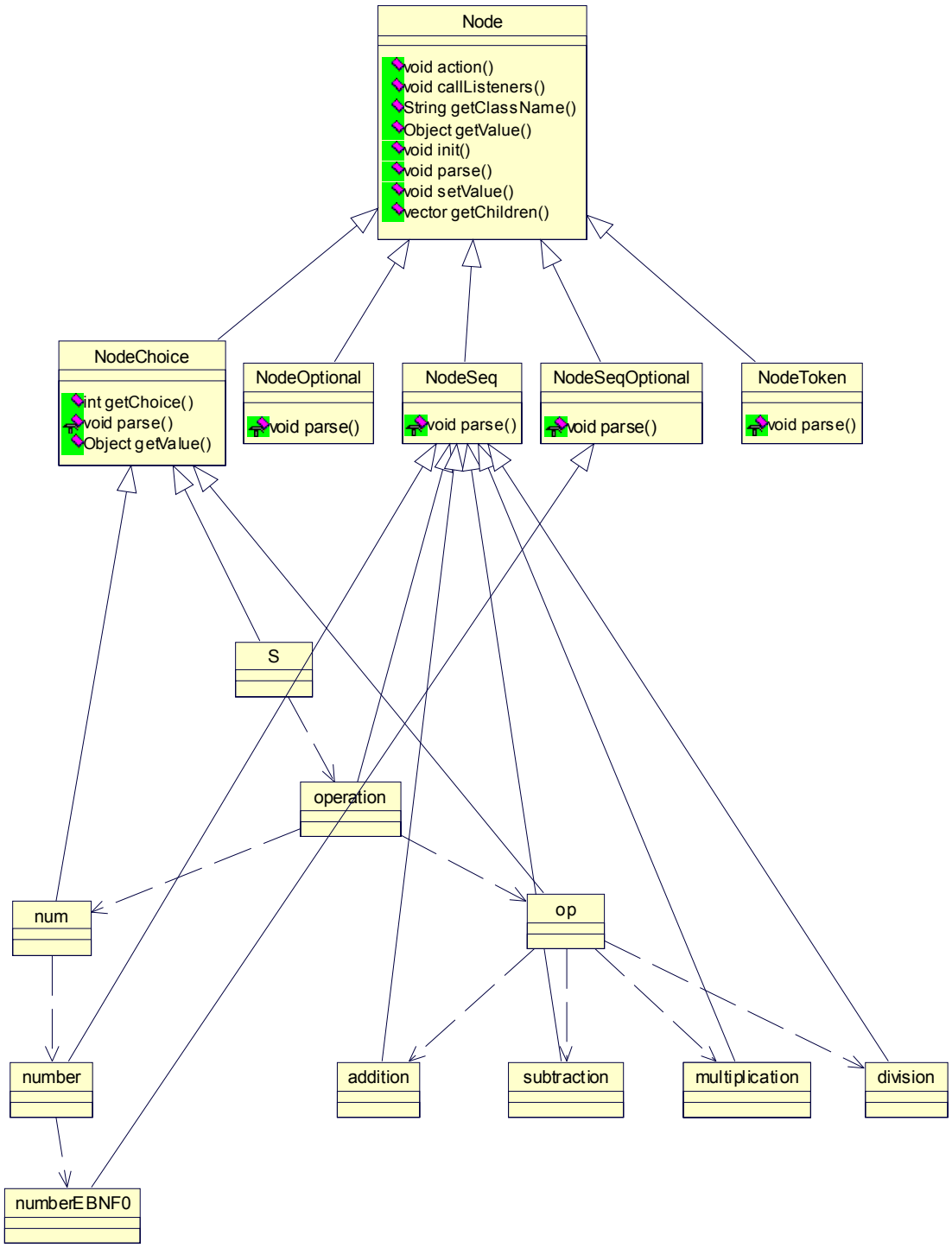


Figura 10.5 Conjunto de clases generadas por el sistema.

En este diagrama se puede observar también que existen determinadas conexiones entre clases por medio de flechas con trazo discontinuo. Representan las dependencias de llamadas entre las distintas clases generadas.

El proceso de análisis del fichero fuente que sigue la estructura definida por la gramática comienza con la llamada al método `parse()` de la primera clase del diagrama (clase S). A partir de esta clase el analizador continúa llamando al método `parse()` de la clase `operation`, continuando el proceso hasta llegar a una clase que llama a `NodeToken`. Por tanto, este proceso de análisis no finaliza hasta que las clases que representan las hojas del AST han terminado su análisis.

Para simplificar el ejemplo, se presentará a continuación el proceso de construcción de listeners contemplando solamente dos producciones.

En la primera producción `<num> <op> <num> "operation" ;` la clase a la que se debe suscribir el listener se llama `operation`. El siguiente código muestra como implementar la interfaz *Listener*.

```
import o2c2rt.*;
import parser.*;

public class AOperation implements Listener {
    public AOperation ()
    {
        operation.addListener (this);
    }
    public void performAction (Node o)
    {
   int result= 0;
   int num1, num2;
   String op;
        try {
              operation n = (operation) o;
              num1 = ((Integer)n.getValue(0)).intValue();
              num2 = ((Integer)n.getValue(2)).intValue();
              op= (String) n.getValue(1);
              if (op.compareTo("-")==0) resul=num1-num2;
              else if (op.compareTo("+")==0) resul=num1+num2;
              else if (op.compareTo("*")==0) resul=num1*num2;
              else if (op.compareTo("/")==0) {
                if (num2==0) {
                  throw new SemanticException ("Error, division by zero");
                }
                else resul = num1/num2;
             }
            System.out.println ("\nResult: "+result);
       } catch (ParserException pe) {
              pe.printStackTrace();
       }
};
```

El constructor añade la clase al conjunto de listeners que han solicitado suscripción a alguna instancia de `operation`.

El listener transforma `getValue(0)` y `getValue(2)`, los dos números de la alternativa, a dos enteros y `getValue(1)` que corresponde a `<op>` es transformado a cadena. Dependiendo del contexto, el listener realizará una determinada operación y presentará el resultado al usuario.

En la segunda producción `<num> ::= DIGIT {DIGIT} "number" ;` la clase a la que se debe suscribir el listener es `number`. Su código es el siguiente:

```
import parser.*;
import o2c2rt.*;
import java.util.Vector;

public class Anumber implements Listener {
    public Anumber ()
    {
        number.addListener (this);
    }
    public void performAction (Node o)
    {
       try {
        number num = (number) o;
        String s1 = (String) num.getValue(0);
        String s2 = "";
          Vector v1 = (Vector)num.getValue(1);
                if (v1!=null) {
                        for (int contador=0;contador<v1.size();contador++)
{
                            String aux = (String) v1.elementAt(contador);
                            s2 = s2+aux;
                        }
                }
                num.setValue (new Integer(s1+s2));
  } catch (ParserException e) {
     e.printStackTrace();
  }
}
};
```

El constructor añade la clase al conjunto de listeners que han solicitado su suscripción a alguna instancia de `number`.

En el método `performAction()`, la variable `getValue(0)` representa un dígito mientras que `getValue(1)` es la variable que corresponde a `{DIGIT}`. Como no existen listeners suscritos a esta clase el valor semántico de la clase EBNF es el valor por defecto, en este caso, una lista (`java.util.Vector`) de los tokens que han sido leídos por la clase. Los elementos son dígitos.

No es necesario crear un listener para cada elemento de la gramática que tiene su correspondiente clase en el framework generado por el sistema. Los valores obtenidos en este caso se mantienen para una utilización posterior en el proceso de análisis.

10.11 Detección, tratamiento y recuperación de errores

La incorporación del tratamiento y recuperación de errores al generador de procesadores de lenguajes es otro de los puntos estratégicos al que se ha dado especial importancia. Dada la complejidad de las especificaciones que tienen que analizar estos procesadores, sería un trabajo frustrante para los usuarios si cada vez que se comete un error, éste no fuese tratado de una manera clara y oportuna.

En general, ante la ocurrencia de un error existen dos formas de proceder, se puede tratar el error y solventarlo de forma adecuada para seguir el análisis a partir del punto donde se había dejado, o bien se puede decidir que es un error lo bastante importante como para no continuar con el análisis y presentar al usuario información relevante de dicho error.

El prototipo contempla las dos posibilidades, permitiendo al usuario bastante flexibilidad con respecto al grado de importancia que quiere otorgar al tratamiento de errores.

10.11.1 Requisitos del tratamiento y recuperación de errores

Como se pudo observar en el capítulo 8, el tratamiento y recuperación de errores que se ha de implementar en el sistema O2C2 tiene que cumplir con una serie de requisitos:

- Proporcionar el control necesario para que el usuario del sistema decida qué tipos de errores quiere tratar y cómo tratar esos errores.

- Permitir que incluso si el usuario no está dispuesto a contemplar los errores que puedan suceder, el analizador sepa actuar correctamente cuando se encuentre con ellos.

- Ser completo y fácil de utilizar por el usuario. No debería requerir la introducción de nuevos conceptos con el objeto de no complicar el aprendizaje.

- Permitir que el sistema se recupere de los errores producidos para poder continuar su proceso.

10.11.2 Descripción del sistema de excepciones

Con el objeto de satisfacer los requisitos mencionados anteriormente, se ha diseñado e implementado en el sistema O2C2, un mecanismo de tratamiento y recuperación de errores que pretende aprovechar conceptos que el usuario conoce: **excepciones** y **listeners**.

Básicamente, en muchos lenguajes que soportan el paradigma de la programación orientada a objetos, el mecanismo de errores está basado en las excepciones. El mecanismo de excepciones permite al desarrollador concentrarse en el flujo de ejecución normal del programa y separar el tratamiento de errores de

este. Cuando en una sección de un programa se producen errores importantes, se lanza una excepción que es una instancia de una clase especial que contiene información sobre el error en cuestión. Esta excepción va pasando desde el punto en el que se originó, hacia el método o función que invocó a dicha sección de código y en el caso de que no trate la excepción, ésta es lanzada otra vez pero en esta ocasión hacia el método o función que invocó al método o función que ha desechado la posibilidad de tratar la excepción. Este proceso sólo termina cuando un método, función o bloque de código trata la excepción, o bien, si finalmente nadie la ha tratado, se presenta un mensaje de error más o menos descriptivo al usuario.

10.11.3 Jerarquía de excepciones

La jerarquía de excepciones que se ha diseñado para dar soporte al mecanismo de detección, tratamiento y recuperación de errores en el sistema O2C2 está incluida en el paquete o2c2rt (O2C2 RunTime). Se puede observar en la *figura 10.6*.

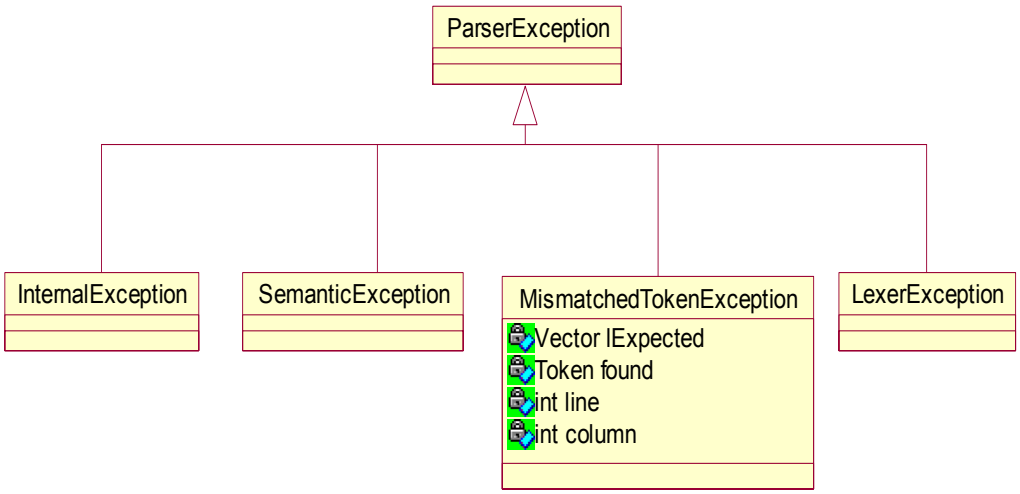


Figura 10.6 Jerarquía de excepciones presente en O2C2rt

Esta jerarquía dará tratamiento a los posibles errores que puedan originarse durante el proceso de análisis léxico, sintáctico y semántico.

ParserException

Esta clase es la raíz de la jerarquía de excepciones. Se utiliza en los casos en los que el usuario quiere detener de manera definitiva el proceso de análisis.

InternalException

Las excepciones de este tipo son lanzadas por el analizador cuando se producen errores en el propio sistema O2C2.

SemanticException

Cuando el código de acción de usuario se encuentra con errores a la hora de tratar valores semánticos generados durante el proceso de análisis, lanza este tipo de excepción.

MismatchedTokenException

Las instancias de esta clase son lanzadas cada vez que el analizador encuentra en la entrada un componente léxico que no se corresponde con lo esperado para seguir de forma correcta el análisis. Cuando se produce uno de estos errores, se recoge toda la información relevante para que el punto concreto que decida solucionarlo sepa que pasó y dónde sucedió dicho error.

LexerException

Es utilizada para señalar errores producidos durante el análisis léxico.

10.11.4 Descripción del mecanismo de tratamiento y recuperación de errores

El mecanismo de tratamiento y recuperación de errores consiste por tanto, en establecer una **jerarquía de excepciones** adecuada, que contemple la existencia de errores.

Si el usuario está interesado en tratar los errores que puedan producirse en una parte de la gramática, entonces se suscribe al evento error del *ancla* que delimita la sección gramatical correspondiente.

Cuando se produce una excepción, si el usuario ha demostrado interés en tratar los errores que pudieran aparecer, se cede el control de la excepción a la clase del usuario encargada de los errores en esa sección de la gramática. Si no hay clases registradas para tratar los errores, entonces la excepción generada se envía hacia arriba, a la producción de un no terminal que haya requerido llamar a la sección de la gramática donde se produjo el error. Si finalmente la excepción no es tratada en ningún lado se presenta el error al usuario del analizador.

Siempre que el usuario haya creado una clase para tratar los distintos errores, dicha clase tendrá que implementar la interfaz **ErrorListener**. Esta interfaz obliga a implementar el método **boolean treatError ()**, dentro del cual el usuario tendrá que decir en cuál de estas tres categorías entra el error :

- El error es **soluble**. En ese caso, el usuario debe asegurarse de que devuelve como valor de retorno de la función **true**. De esta manera el analizador entenderá que puede continuar el análisis a partir del siguiente token que le proporcione el analizador léxico.

- El error **no** es **soluble**, pero se puede aspirar a resolverlo en otra parte. Entonces el valor de retorno es false y el analizador se encarga de reenviar la excepción producida por el error para ser tratada en otro punto.

- El error no sólo es insoluble actualmente sino que es un **error fatal**. En ese caso el usuario debe lanzar una instancia de la excepción **ParserError**. Este tipo de excepción hace que pare inmediatamente el proceso de análisis y que se presente un mensaje de error informativo al usuario.

10.11.5 Tratamiento de errores en los nodos del framework genérico

De la jerarquía de clases del framework o2c2rt, son dos las que pueden encontrar en primera instancia los errores referidos a entradas que no se corresponden con la especificación de la gramática. `NodeToken`, que se encarga de leer directamente los tokens del analizador léxico, por lo que es la primera en darse cuenta de que un token leído no es válido, y `NodeChoice`, que es una clase que debe elegir entre las distintas producciones de un no terminal para realizar el análisis descendente.

Estas dos clases son las que consideran inicialmente la posibilidad de encontrarse con un error que pueda ser recogido en un `MismatchedTokenException`. Cuando esto sucede, construyen una instancia de esta clase de excepción, y recogen la información necesaria. La línea y columna en la que se produjo el error, el token que se leyó y una lista de todos los tokens que podrían haber servido.

Todas las clases del framework genérico incluyen un **gestor de excepciones** de este tipo al lado de la llamada al método `parse()`. Si la clase ha registrado que un objeto creado por el usuario del sistema está pendiente de la existencia de un error como ese, entonces le pasa a esa clase el error. Si el objeto del usuario lo resuelve, el análisis continua un token más allá de donde se había detenido, si no consigue solventar el error, se vuelve a lanzar el error por si un objeto diferente en la pila de llamadas consigue solventarlo. Por último, si finalmente nadie ha sabido tratar el error adecuadamente, se detiene el proceso de análisis y se presenta un mensaje advirtiendo al usuario del error.

10.11.6 La interfaz ErrorListener

Como se ha visto anteriormente, el usuario del sistema O2C2 puede crear clases que serán invocadas cuando se produzcan errores. Para ello se utiliza un mecanismo basado en *listeners*.

En este caso, la interfaz que tienen que implementar las clases de usuario que van a tratar errores, es la interfaz `Errorlistener`. Esta interfaz obliga al usuario a implementar el método cuyo prototipo es:

```
Boolean treatError (MismatchedTokenException e)
```

Este método es invocado por la clase que recibe en un determinado momento un `MistmatchedTokenException`, y conoce que tiene una clase de usuario suscrita a ese tipo de evento.

El usuario del sistema puede realizar en este método el tratamiento del error que considere oportuno. En todo caso, se espera que devuelva un valor lógico. Si el resultado del método es true, se entiende que se ha producido una recuperación del error y el análisis continuará a partir del token ya tratado. Si el resultado del método es false, se entiende que no se ha podido tratar el error en ese momento y por lo tanto se volverá a lanzar la excepción para ver si se puede tratar de forma adecuada en otro punto.

Si el usuario entiende que el error es lo bastante grave como para detener inmediatamente el proceso de análisis, lo puede realizar lanzando una excepción de tipo `ParserException`. El diagrama de secuencia de la *figura 10.7* muestra la actuación de un *listener* en colaboración con la clase que suscribe y su interfaz para el tratamiento de errores.

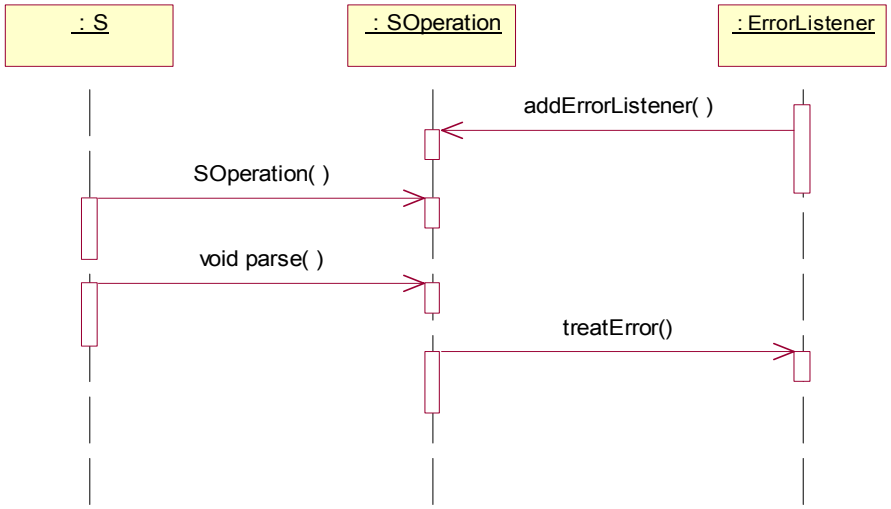


Figura 10.7 Diagrama de colaboración entre un listener, la clase que suscribe y su interfaz

10.12 Entorno integrado de desarrollo

El subsistema del prototipo O2C2 conocido como O2C2ui, es la interfaz gráfica de usuario que se puede utilizar para la construcción de procesadores de lenguajes de forma integral. Es necesario puntualizar aquí que este subsistema no resulta imprescindible para obtener un prototipo funcionalmente completo. La misma funcionalidad puede ser conseguida a nivel de línea de comandos.

Como ya se comentó en el capítulo 8, el entorno de desarrollo proporciona una forma sencilla e intuitiva de realizar las diferentes operaciones encaminadas a la construcción de un procesador de lenguaje. Con la incorporación de este subsistema al prototipo, se intenta recoger la mayor parte de las características y requisitos asociados al entorno visual de desarrollo del sistema, no obstante, para el desarrollo del prototipo se ha construido un entorno menos ambicioso pero diseñado con la idea de que pueda ser completado en futuras versiones.

Para una información detallada de su uso se debe consultar el Manual de Usuario del apéndice A.

10.12.1 Objetivos de O2C2ui

Los objetivos que se persiguen con la interfaz gráfica de usuario son:

- Facilitar la labor al usuario integrando en un solo entorno los procesos de análisis léxico, sintáctico y semántico necesarios para la generación de procesadores de lenguajes.

- Proporcionar un entorno que permita suavizar la curva de aprendizaje de un usuario no necesariamente experto en la teoría de construcción de procesadores de lenguajes.

- Realizar de manera visualmente agradable, la introducción de los diversos elementos que forman parte de la especificación de la gramática.

- Obtener los componentes léxicos que permitan la generación del analizador léxico correspondiente.

- Permitir al usuario obtener los esqueletos de las clases que deberán encargarse de la semántica del procesador de lenguaje generado sin que este tenga que crear las clases listener explícitamente.

- Flexibilizar la estructura de los ficheros generados por el entorno de desarrollo, permitiendo no solamente la obtención de ficheros de texto y ficheros binarios sino que también ofrezca la posibilidad de almacenar los datos del proyecto que aún está siendo desarrollado, en un único fichero utilizando para ello una aplicación XML que permita almacenar la información con un formato no plano.

10.12.2 Visión general del sistema integrado

En la *figura 10.8* se representan los principales bloques encargados de proporcionar la funcionalidad del subsistema O2C2ui.

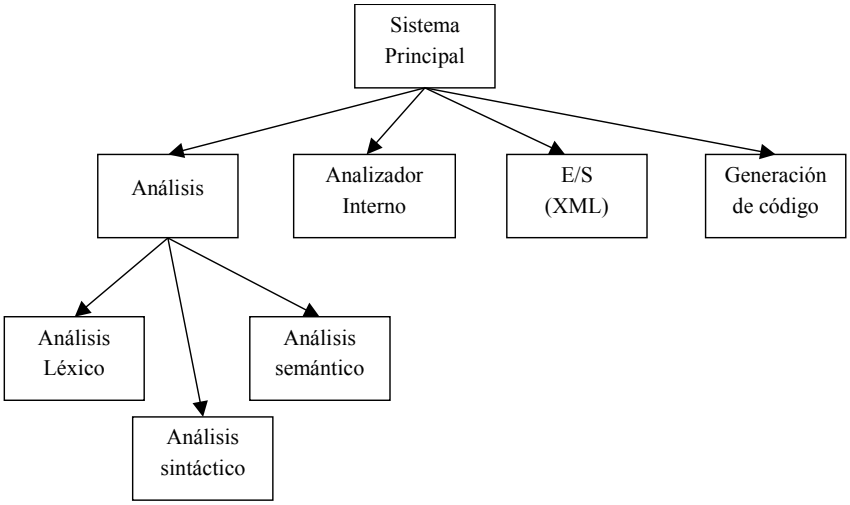


Figura 10.8 Representación por bloques de la funcionalidad proporcionada por O2C2ui

10.12.3 Ventana principal de O2C2ui

Cuando se inicia el sistema se accede a la ventana principal (*figura 10.9*) que es la ventana central de la interfaz desde donde se puede acceder a todas las opciones del sistema. Se distinguen 4 zonas:

1. Barra de menús.
2. Barra de herramientas.
3. Área de activación del tipo de análisis.
4. Barra de estado.

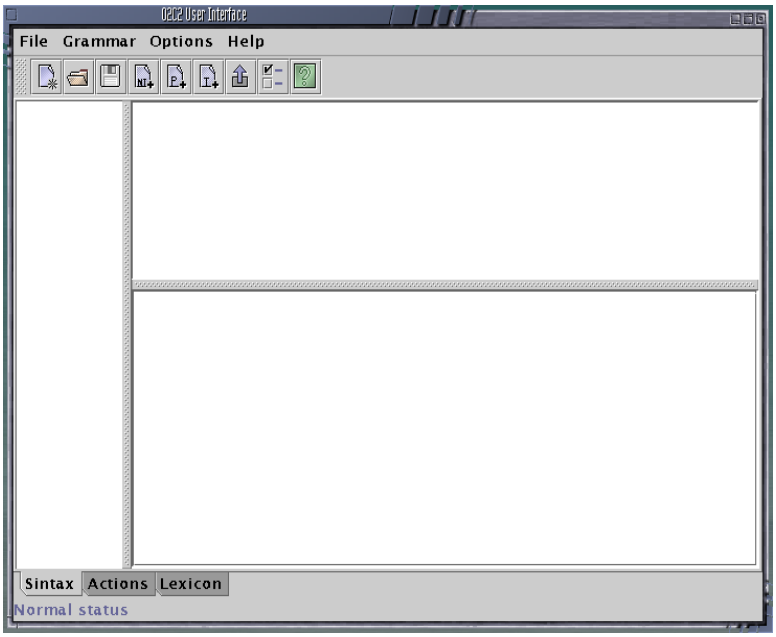


Figura 10.9 Ventana principal de O2C2ui

La barra de menús

En ella se encuentra un enlace a todas las acciones a las que se puede acceder desde la interfaz gráfica. Se encuentran agrupadas en: Fichero, Gramática, Opciones y Ayuda. Cada menú presenta a su vez sus respectivas opciones a través de submenús.

La barra de herramientas

Permite realizar las principales tareas en los diferentes análisis. Cuando el usuario mantiene el puntero del ratón sobre un botón, éste se resaltará e indicará mediante una etiqueta la acción que se ejecutará si se pulsa dicho botón.

El área de activación del tipo de análisis

La ventana contiene tres pestañas que se corresponden con los tres tipos de análisis que se realizan dentro del entorno. Cuando se selecciona una pestaña, se da paso a una ventana diferente para tratar ese tipo de análisis.

La barra de estado

Se encuentra situada en la parte inferior de la ventana y muestra información relacionada con las distintas operaciones que se van realizando en el entorno.

10.12.4 El trabajo con O2C2ui

La interfaz gráfica de usuario permite integrar todas las etapas del proceso de construcción de un procesador de lenguaje de forma fácil e intuitiva. Actualmente, se están llevando a cabo modificaciones para permitir que el usuario pueda:

1. Introducir la descripción gramatical sin tener que conocer la notación propia del sistema O2C2 sino mediante diagramas gráficos.

2. Realizar modificaciones tanto a escala visual como en el ámbito de la especificación textual sin que existan problemas de sincronización.

3. Reutilizar estructuras gramaticales y componentes léxicos mediante la encapsulación de los mismos a través de componentes.

10.12.4.1 Análisis sintáctico

Todo el proceso se centra alrededor del análisis sintáctico. El contenido de la ventana se muestra en la *figura 10.10*.

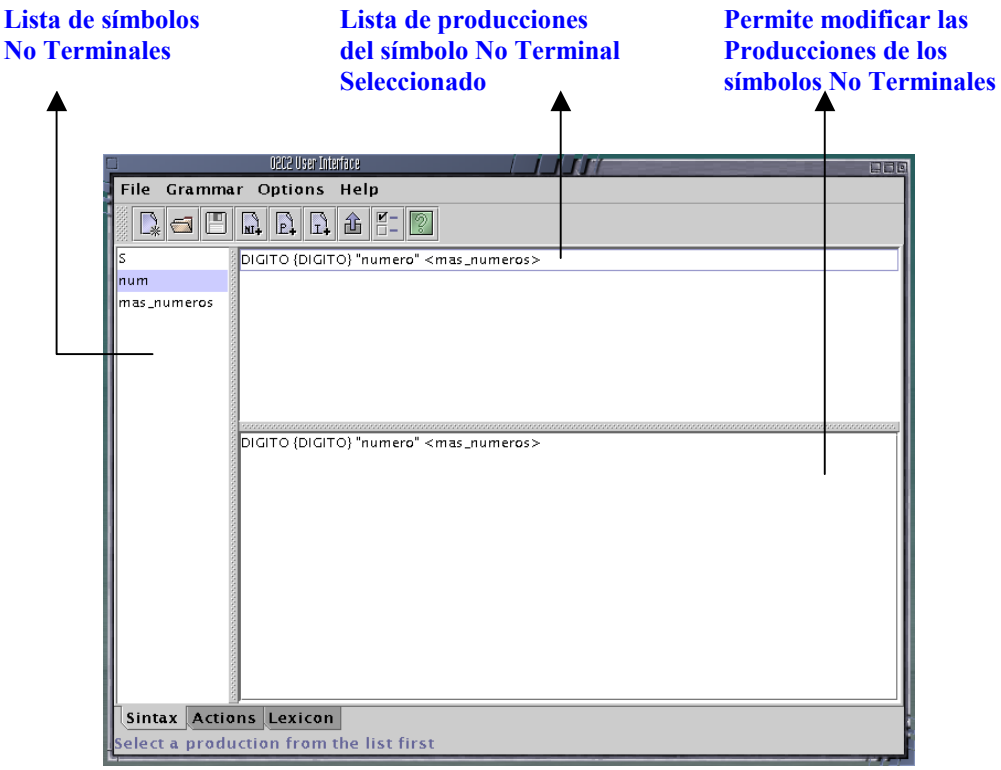


Figura 10.10 Ventana de especificación sintáctica

Cuando se introduce una producción para un no terminal, se analiza si dicha producción introduce nuevos elementos *terminales*, *no terminales* o anclas. Si se detecta un elemento nuevo, entonces se incluye una referencia a dicho elemento introduciendo cambios en la gramática que describe la sintaxis. Si es un elemento terminal, los cambios serán recogidos en la ventana de especificación léxica. Si es un elemento *ancla*, los cambios serán introducidos en la ventana que trata el análisis semántico.

Para realizar esta tarea, O2C2ui cuenta con un pequeño analizador interno de producciones, basado en el subsistema O2C2Conv y generado por el propio sistema O2C2.

10.12.4.2 Análisis léxico

No es necesario introducir de manera explícita los elementos que componen el léxico del lenguaje. En la ventana asociada al análisis léxico (se puede observar en la *figura 10.11*), habrá una lista de todos los tokens que se han encontrado en la gramática introducida por el usuario. Se pueden incluir también las definiciones regulares (en forma de expresiones regulares) asociadas a dichos tokens.

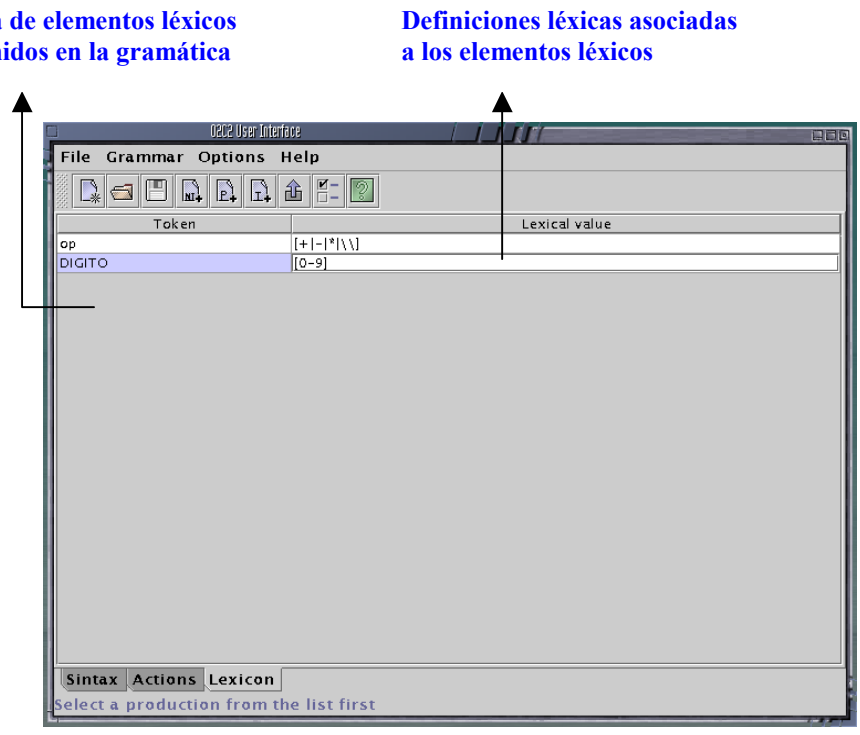


Figura 10.11 Ventana de especificación léxica

10.12.4.3 Análisis semántico

En la ventana asociada al análisis semántico se encuentra una lista con todos los *anclas* definidos en la especificación gramatical. Seleccionando sus nombres se obtiene un esqueleto de un listener que se podrá modificar para que se adapte a

unas necesidades marcadas por el tipo de procesador de lenguaje que se desea generar.

El contenido de la ventana se muestra en la *figura 10.12*. En su parte izquierda aparecen los distintos anclas definidos en la gramática, mientras que en el lado derecho aparece el código de acción asociado a un ancla en forma de clase listener. Si el usuario no hubiera creado previamente dicha clase, O2C2ui genera de forma automática un esqueleto de la misma para facilitar su labor.

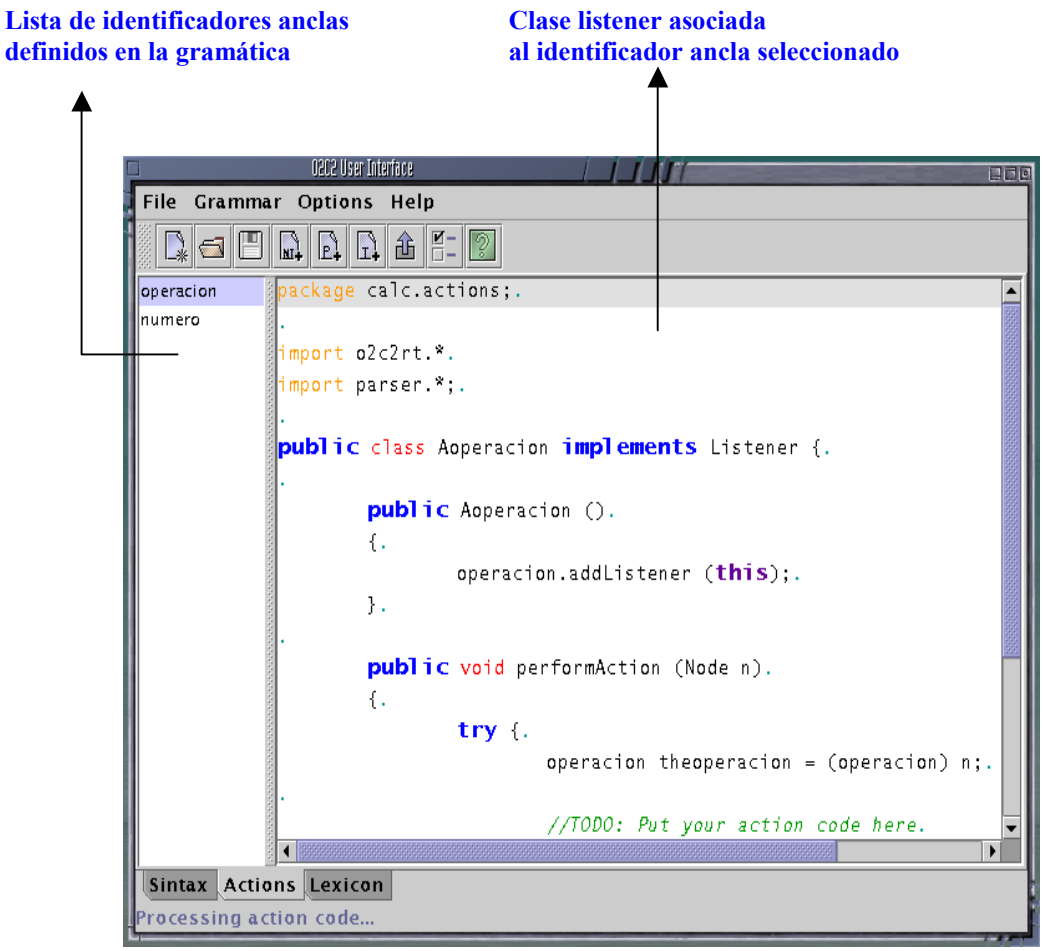


Figura 10.12 Ventana de especificación semántica

10.12.5 Ficheros generados por O2C2ui

Salvo que el procesador de lenguaje que se pretenda diseñar e implementar gracias a O2C2ui esté completamente acabado, es más cómodo almacenar todos los datos del proyecto en un sólo fichero que sea fácilmente transportable, en vez de utilizar numerosos ficheros distribuidos a lo largo de diversos directorios.

O2C2ui genera ficheros cuyo procesamiento por parte de un compilador Java da lugar a un procesador de lenguaje completo pero también permite almacenar los datos que forman parte de un determinado proyecto en un solo fichero en forma de árbol de información.

Para llevar a cabo esta tarea es necesario buscar una estructura diferente a la que imponen los ficheros de texto, ya que son ficheros planos, y los ficheros binarios, que no permiten la existencia de otras herramientas, aparte de las que los generó, para tratar dichos ficheros.

La solución a esos problemas pasa por utilizar una aplicación XML. Se basa en aplicar el metalenguaje XML [BPS00] sobre un lenguaje de marcas concreto.

Con la utilización de XML, es posible almacenar la información de forma no plana. De esta forma, el entorno de desarrollo O2C2ui podrá almacenar los datos del proyecto que se está desarrollando en forma de árbol de información.

Al ser un fichero que contiene texto y al mismo tiempo información sobre la estructura del mismo, aunque no la semántica de este, es necesario utilizar una herramienta que pueda realizar tareas con los datos almacenados en estos ficheros.

La estructura de los ficheros en formato XML generados por O2C2ui se puede observar en el *ejemplo 10.2*.

Ejemplo 10.2 *El siguiente ejemplo muestra el contenido del fichero generado a partir de las especificaciones de una calculadora sencilla.*

```
<?xml version="1.0"?>
<analyzer>
<syntax>
<nonterm>
   <name> s </name>
   <production> <![CDATA[<num> OP <num> "operation"]]>
     </prodution>
</nonterm>
<nonterm>
   <name> num </name>
   <production> <![CDATA[DIGITO {DIGITO} "number"]]>
     </prodution>
</nonterm>
</syntax>
<lexicon>
<token>
   <name> DIGITO </name>
   <value> <![CDATA[0-9]]> </value>
</token>
<token>
     <name> OP </name>
     <value> <![CDATA[+|-]]> </value>
   </token>
</lexicon>
<actions>
<action>
   <name> Aoperation </name>
  <link> operation </link>
  <code> <![CDATA[package calculadora.actions;
import o2c2rt.*
import parser.*;
```

```
public class Aoperation implements Listener {
public Aoperation ()
   {
      operation.addListener (this);
   }
   public void performAction (Node o)
   {
      operation n = (operation) o;
      // Put your action code here
   }
};
]]> </code>
   </action>
<action>
   <name> Anumber </name>
    <link> number </link>
<code> <![CDATA[package calculadora.actions;

import o2c2rt.*
import parser.*;

public class Anumber implements Listener {
public Anumber ()
{
   number.addListener (this);
}
public void performAction (Node o)
{
   number num = (number) o;
   // Put your action code here
}
};
]]> </code>
   </action>
</actions>
</analyzer>
```

La estructura consta de tres partes:

- La primera sección, corresponde al **análisis sintáctico** y consiste en una sucesión de símbolos *no terminales* junto a sus producciones.

- La segunda sección, corresponde al **análisis léxico.** Contiene una lista de símbolos *terminales* con sus correspondientes definiciones léxicas.

- La tercera sección del fichero, corresponde al **análisis semántico**. Contiene una conjunto de *anclas* con su código de acción asociado. Para incluir este código en el fichero XML, en vez de entrecomillarlo como podría ser habitual, se ha incluido dentro de un *tag* llamado **CDATA**. Esto permite no tener que anteceder con secuencias de escapes los entrecomillados que puedan estar incluidos dentro del propio código.

10.12.6 Tratamiento de ficheros XML en O2C2ui

El entorno de desarrollo necesita incorporar un procesador de lenguaje que sea capaz de interpretar los datos almacenados en los ficheros con formato XML. Este procesador fue generado por el propio sistema O2C2 fundamentalmente por tres razones:

- Se prueba una vez más la capacidad del sistema O2C2 para generar procesadores de lenguaje de forma rápida y sencilla.
- Se adquiere experiencia para observar puntos débiles en el proceso de construcción de O2C2.
- Se evita utilizar herramientas externas.

Especificación gramatical

La especificación gramatical utilizada en este caso para la construcción del procesador de lenguaje fue la siguiente:

```
<Grammar>    ::= <Header> <Analyzer>;
<Header>     ::= XMLHDR;
<Analyzer>   ::= OPENANALYZERTAG <Syntax><Lexicon><Actions>
                 CLOSEANALYZERTAG;
<Syntax>     ::= OPENSYNTAG <Nonterm> {<Nonterm>} CLOSESYNTAG;
<Nonterm>    ::= OPENNTTAG <Name> <Production> {<Production>}
                 CLOSENTTAG;
<Name>       ::= OPENNAMETAG VALUE CLOSENAMETAG;
<Production> ::= OPENPRODTAG CODE CLOSEPRODTAG;
<Lexicon>    ::= OPENLEXTAG <Token> {<Token>} CLOSELEXTAG;
<Token>      ::= OPENTKTAG <Name> <Value> CLOSETKTAG;
<Value>      ::= OPENVALTAG CODE CLOSEVALTAG;
<Actions>    ::= OPENACTIONSTAG <Action> {<Action>}
                 CLOSEACTIONSTAG;
<Action>     ::= OPENACTTAG <Name> <Link> <Code> CLOSEACTTAG;
<Link>       ::= OPENLNKTAG VALUE CLOSELNKTAG;
<Code>       ::= OPENCODETAG CODE CLOSECODETAG;
```

CAPÍTULO 11

VENTAJAS DEL SISTEMA DISEÑADO Y APLICACIONES

En este capítulo se van a describir las principales ventajas aportadas por el modelo diseñado, que corroboran la utilidad de éste, así como sus posibles aplicaciones.

11.1 Ventajas del sistema diseñado

11.1.1 Eficiencia

La eficiencia de los sistemas que permiten generar procesadores de lenguajes se puede medir utilizando dos enfoques diferentes. Por una parte habría que analizar la eficiencia de los procesadores de lenguajes generados, y por otra sería necesario analizar dicha eficiencia desde el punto de vista de la propia herramienta.

Este concepto de eficiencia, a su vez, puede ser entendido de dos formas diferentes. En primer lugar, la eficiencia de una aplicación puede ser considerada como la medida de recursos, tanto de uso de procesador como de memoria que ésta mantenga. En segundo lugar, en el caso de un sistema generador, es también relevante hacer esta consideración en la medida en que éste ayude a ser eficiente al usuario del sistema.

11.1.1.1 Eficiencia de los procesadores de lenguajes generados

El sistema diseñado, al estar basado en técnicas orientadas a objetos, ofrece un conjunto de características (modularidad, reusabilidad, extensibilidad, y mantenimiento) que hacen posible disminuir la complejidad de las aplicaciones generadas. Estas aplicaciones no van a ser en general más rápidas que las generadas por algunas herramientas creadas y utilizadas con fines análogos (su descripción puede verse en el capítulo 4), pero sí se ha hecho especial énfasis en obtener procesadores de lenguajes eficientes.

En cualquier caso, las pequeñas diferencias de rendimiento no tienen porque ser especialmente relevantes, una vez superado el hecho de que las llamadas a métodos de clases en el sistema propuesto son equivalentes a llamadas a funciones en sistemas similares. Aunque teóricamente ésto es menos eficiente, no existe ninguna otra razón por la cual un procesador de lenguaje generado por el sistema diseñado tenga por qué ser menos eficiente.

Para poder conseguir este objetivo, en el diseño e implementación del sistema se han contemplado diferentes estrategias que permiten una mejora del rendimiento. Una prueba de ello es que el texto fuente a analizar nunca está completamente en memoria sino que se va leyendo en función de las necesidades. Así mismo, los diferentes objetos del framework sólo son inicializados cuando es absolutamente necesario y justo antes de ser utilizados.

De hecho, al no exigir el sistema que se realice un tratamiento semántico (código de acción asociado a una sección gramatical) en todas y cada una de las secciones de la gramática para que su valor asociado no se pierda, se consigue que los procesadores de lenguajes generados tengan menos código y por tanto sean más eficientes. Otra ventaja que incorpora el sistema es la modularidad de las aplicaciones generadas, lo cual puede repercutir en el desarrollo de proyectos grandes donde el rendimiento tiene especial importancia, y que contrasta con los sistemas tradicionales en los que las aplicaciones generadas pueden tener bastante código redundante.

No obstante, no se han realizado pruebas específicas comparando el rendimiento de las aplicaciones generadas por el sistema diseñado y las demás técnicas de construcción de procesadores de lenguajes. Esta decisión tiene su justificación por el hecho de que las diferencias de rendimiento no serían muy apreciables sobre los equipos actuales en los que la velocidad de cálculo y las capacidades de almacenamiento son cada vez mayores.

11.1.1.2 La eficiencia del sistema propuesto

Una de las principales ventajas del sistema diseñado es precisamente la optimización del ciclo de desarrollo del software.

Como se puede apreciar en el diagrama situado a la izquierda de la *figura 11.1* con las técnicas tradicionales de construcción de procesadores de lenguajes el ciclo de desarrollo pasa por los siguientes pasos. En primer lugar el programador escribe la descripción de la gramática junto con el código de acción semántica en un mismo fichero de especificaciones. A continuación se genera el código de la aplicación. Este código debe ser compilado a un programa ejecutable para ser probado y depurado. El problema de este ciclo radica en que el código de acción semántica está mezclado con las especificaciones de la gramática. Si se localiza algún error durante la ejecución del programa debe ser el programador el que se encargue de localizar el código correspondiente en el fichero de especificaciones, corregirlo y volver a realizar los pasos necesarios hasta obtener nuevamente un programa ejecutable.

El diagrama situado a la derecha de la *figura 11.1* muestra cómo este ciclo de *edición – compilación – depurado,* se acorta por la separación clara entre la especificación de la gramática y el código de acción semántica asociado. Esto significa que cuando se necesita realizar un cambio en el código de acción semántica, solamente se tienen que recompilar las clases afectadas. De esta forma, se permite realizar el proceso de depurado en un entorno integrado de desarrollo.

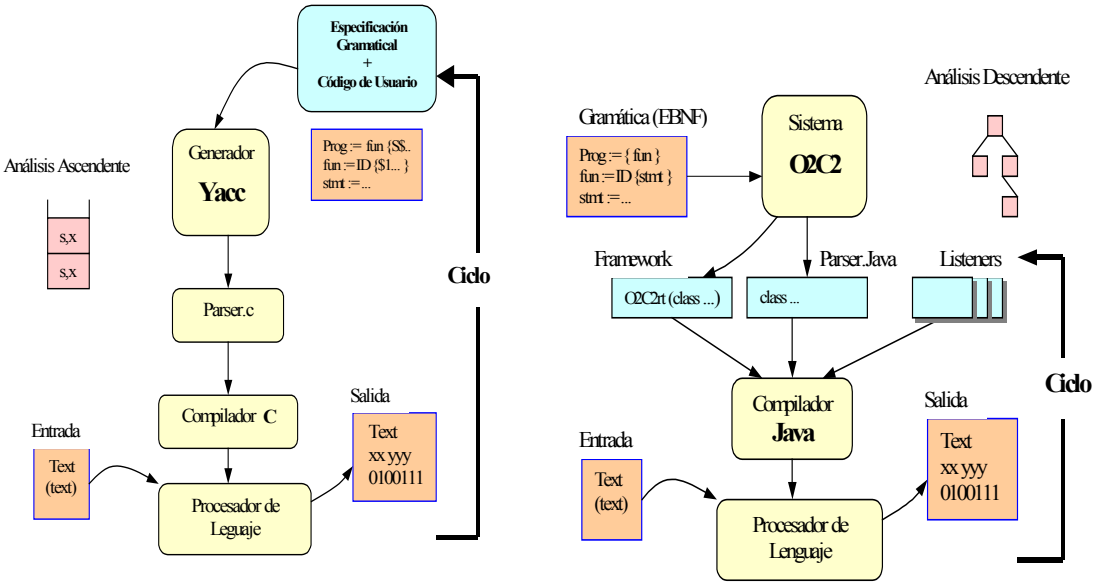


Figura 11.1 Ciclo de desarrollo de las técnicas tradicionales *versus* sistema O2C2

Una de las formas más importantes de mejorar la productividad en la ingeniería del software consiste en proporcionar descripciones apropiadas de los problemas y sus soluciones. El sistema aquí presentado reduce el coste de producción de los procesadores de lenguajes.

11.1.2 Robustez

Las técnicas de construcción complicadas, que incorporan la interacción de muchos elementos en términos conceptuales, están expuestas a que se cometan más errores y se incremente el esfuerzo necesario durante el desarrollo de las aplicaciones. En este caso, la aplicación de técnicas orientadas a objetos de manera uniforme hace que resulte muy simple conceptualmente, y su implementación sea conocida por los usuarios en general. De esta forma, se consigue minimizar de manera efectiva la curva de aprendizaje del modelo diseñado, puesto que no estaría influenciado por diversas convenciones y opciones necesarias para operar como sucede con otros sistemas.

Por otro lado, el sistema, al no estar basado en herramientas desarrolladas de forma independiente unas de otras, permite integrar la funcionalidad básica siguiendo el mismo modelo orientado a objetos. Podríamos decir por tanto, que la eliminación de barreras adicionales que dificultan el uso, hacen de él un sistema robusto.

11.1.3 Aceptación

Un buen sistema generador de procesadores de lenguajes debe ser fácil de utilizar para que sea aceptado por los usuarios y cumplir así su función. La facilidad de aprendizaje supone que ha de estar basado en un conjunto uniforme de conceptos familiares para el usuario. El sistema diseñado utiliza el modelo orientado a objetos para proporcionar la funcionalidad básica descrita en el capítulo 7, y la notación EBNF para la descripción de las gramáticas, no añade otros conceptos. Esto permite una utilización fácil y cómoda por usuarios no experimentados o con conocimientos limitados de los conceptos teóricos de los procesadores de lenguajes.

11.1.4 Diseño abierto

El sistema propuesto ha sido ideado para ofrecer un diseño abierto en el que se puedan incorporar fácilmente nuevas capacidades. De esta forma, resulta fácil operar con analizadores léxicos construidos fuera del propio sistema, en este caso solamente sería necesario proporcionar el método `yylex()` encargado de reconocer los tokens correspondientes de la entrada. Por otra parte, aunque inicialmente se ha diseñado el sistema para realizar un análisis sintáctico descendente y poder obtener así aplicaciones más legibles, la misma filosofía de trabajo puede ser aplicada para realizar un análisis sintáctico ascendente. La inclusión en el sistema de un modelo que construya una tabla de análisis a partir de la especificación de la gramática, permitiría al usuario poder elegir el tipo de análisis.

Así mismo, aunque el tratamiento semántico se realiza a través de las clases que proporciona el usuario, también sería posible incorporar un sistema de tipos genérico extendiendo las interfaces que ofrece el framework.

11.2 Aplicaciones

11.2.1 Plataforma de experimentación

Una de las aplicaciones más inmediatas del sistema diseñado consiste en servir de plataforma de experimentación para la implementación de lenguajes de manera automática. La utilización de tecnologías orientas a objetos, la aplicación de la modularidad para el desarrollo de las aplicaciones y la incorporación de un entorno integrado de desarrollo hacen posible la descripción de diferentes definiciones formales de lenguajes de programación usando la generación automática de procesadores de lenguajes de una forma simple y rápida.

El hecho de contar con una plataforma como ésta, ayudará a proporcionar el soporte necesario para la implementación de lenguajes en el plano educacional y comercial.

11.2.2 Lenguajes de dominio específico

Tradicionalmente los analizadores sintácticos han sido utilizados en la construcción de compiladores, sin embargo, también resultan muy útiles para realizar diferentes tareas relacionadas con la programación. Un uso común de estos analizadores incluye: *lenguajes de programación (Java, C++, ...), lenguajes de marcas (HTML, XML, ...), formatos estándar comerciales (IDL, ODL), formatos de ficheros (RTF, Postscript, ...), lenguajes de bases de datos (SQL, ...), lenguajes de modelado (VRML), procesamiento en línea de comandos, lenguajes de propósito especial, protocolos (http, ftp, ...).*

De hecho, el desarrollo de dichas aplicaciones ocupa un lugar central y está ganando cada día mayor terreno en el proceso de construcción del software, por lo que la aplicación de tecnologías robustas que faciliten estas tareas resulta una necesidad. Como ya se ha señalado anteriormente, las personas que trabajan en esas áreas pueden ser expertas en dichos campos, pero no necesariamente en el campo de los procesadores de lenguajes.

Con el sistema diseñado, la aplicación de técnicas orientadas a objetos a través de frameworks resultan adecuadas para su aplicación en el campo de los lenguajes de dominio específico. "*Analizando una entrada y luego llamando a uno o más métodos generados por el sistema, la entrada puede ser transformada de una forma muy potente utilizando simples especificaciones* " [Joh00].

CAPÍTULO 12

CONCLUSIONES

12.1 Discusión general

Al inicio de esta memoria se mostró la importancia de los generadores para los desarrolladores de procesadores de lenguajes. A partir de las especificaciones que como mínimo, deben incluir la gramática del lenguaje que se va a procesar, es posible obtener analizadores léxicos y sintácticos, o incluso generar todas las fases de un procesador de lenguaje sin demasiado esfuerzo, aunque siempre condicionado por las características que imponga el sistema utilizado.

Posteriormente se dedicó el trabajo a estudiar los sistemas más relevantes utilizados en la actualidad, y se observó que estos sistemas no estaban exentos de problemas. Entre las carencias más importantes atribuibles a las técnicas de construcción de procesadores de lenguajes revisadas destacan: *reusabilidad, modularidad, extensibilidad, mantenimiento,* y por último, el soporte de *entornos visuales de desarrollo*.

De una forma constante estos sistemas mezclan las especificaciones sintácticas con el código de acción semántica y en ocasiones también las especificaciones léxicas. Esto supone:

- Un incremento de la *complejidad* tanto en la entrada al generador, ya que resulta más difícil comprender la estructura y el sentido de una especificación cuando al lado de cada uno de los elementos gramaticales se incluye el código de acción asociado, como en las aplicaciones generadas.

- Una disminución de la *reusabilidad* de las especificaciones gramaticales y el código de acción semántica, puesto que estas acciones y los tipos de datos asociados a los símbolos de la gramática, pueden cambiar según determinadas circunstancias y en diferentes aplicaciones prácticas.

- Un aumento del *ciclo de desarrollo* del software generado, puesto que la duplicación del código de acción en la especificación y en el programa generado, deja al programador la responsabilidad de mantener

actualizados el fichero de especificaciones y el programa resultante, lo cual es claramente ineficiente.

- Un incremento de las tareas de *depuración,* al necesitarse un mayor esfuerzo para poder solucionar los inevitables errores cometidos por el programador al no estar familiarizado con el, probablemente, complejo código generado debido a que estos errores solo son visibles en el programa generado.

Por otra parte es necesario destacar la *falta de integración* de estos sistemas en los denominados entornos de desarrollo. Esto significa que para la construcción de procesadores de lenguajes, se utilizarán lenguajes de programación, pero sin poder emplear los entornos de desarrollo específicamente diseñados para ellos.

Se planteó entonces la necesidad de desarrollar un nuevo sistema, basado en la aplicación de tecnologías orientadas a objetos, capaz de solventar las limitaciones encontradas en las técnicas revisadas y verificar la factibilidad del mismo.

En el capítulo 5 se mostraron los beneficios de las técnicas orientadas a objetos y se destacaron los aspectos que favorecen una arquitectura basada en frameworks. Así mismo, en el capítulo 6 se señalaron las ventajas del uso de máquinas abstractas orientadas a objetos en el desarrollo de los procesadores de lenguajes. La dotación de características como la flexibilidad y portabilidad hacen de ellas una plataforma idónea de trabajo.

Los requisitos generales que debe cumplir el sistema propuesto se especificaron en el capítulo 7 y la arquitectura del sistema propuesto se presentó en el capítulo 8 de esta memoria, analizándose sus componentes, propiedades y características. Una de las aplicaciones más inmediatas del sistema diseñado consiste en servir de plataforma de experimentación para la implementación de lenguajes de manera automática.

Con el fin de evaluar la bondad del sistema diseñado se implementó el prototipo descrito en los capítulos 9 y 10. Todos los analizadores utilizados internamente en el prototipo han sido generados por el propio sistema probándose de esta forma, su capacidad inmediata para generar procesadores de lenguajes de forma rápida y sencilla.

Las principales ventajas aportadas por el modelo diseñado, que corroboran la utilidad de éste, así como sus posibles aplicaciones son presentadas en el capítulo 11.

12.2 Comparación

Para obtener una valoración del sistema O2C2 es importante realizar una comparación con los diferentes sistemas que se pueden utilizar para obtener procesadores de lenguajes. Este proceso permitirá determinar las características que poseen y realizar una comparación cualitativa.

12.2.1 Relación con otros trabajos

Existen dos trabajos que aparecieron después de haberse iniciado el desarrollo del sistema descrito en esta tesis y que guardan cierta relación en algunos aspectos. Su estudio se incluye en el este apartado para poder realizar una comparación después de haber visto los requisitos, el diseño e implementación del sistema O2C2.

En primer lugar, el proyecto *SableCC*, intenta introducir técnicas orientadas a objetos para generar de forma automática árboles sintácticos abstractos (AST) en la construcción de procesadores de lenguajes. Sin embargo, su estructura es poco modular, dinámica, flexible y reutilizable.

En segundo lugar, el proyecto *ProGrammar*, proporciona un entorno visual de desarrollo de analizadores sintácticos, pero su uso está limitado por la plataforma y entorno de trabajo que se utilice.

A continuación se describen sus características principales destacando las diferencias existentes entre estos dos trabajos y el sistema descrito en esta memoria.

SableCC

El proyecto SableCC [GH98] ofrece una funcionalidad similar a la de la mayoría de los sistemas revisados en el capítulo 4, permite construir analizadores léxicos basados en autómatas finitos determinísticos, analizadores sintácticos ascendentes basados en gramáticas LALR(1) y árboles sintácticos abstractos. Su principal diferencia, estriba en la introducción de técnicas orientadas a objetos para generar un conjunto de clases que representan el árbol sintáctico abstracto (AST), a partir de las producciones y elementos de la gramática proporcionada.

SableCC necesita que el usuario le proporcione un fichero de especificaciones formado por las descripciones léxicas y sintácticas en notación EBNF del lenguaje. Esto permite una buena integración de los analizadores léxicos y sintácticos, sin embargo, el soporte de gramáticas en notación EBNF es deficitario puesto que no se adapta bien a la forma de trabajar internamente la aplicación y es necesario convertirlas en otras equivalentes en notación BNF con la consiguiente perdida de información, puesto que el nombre de las clases que se generan para construir el AST no es el que inicialmente espera el usuario y tiene que ser consciente de los cambios que se produzcan. Por otra parte, el usuario también debe proporcionar código adicional que le permita recorrer y tratar el AST generado. Todo el código se integra en una sola clase dificultando la modularidad y reusabilidad del mismo.

Este proyecto guarda cierta relación con el sistema O2C2 en el sentido de que ambos generan un conjunto de clases a partir de la especificación gramatical proporcionada por el usuario. Sin embargo, existen numerosas e importantes diferencias. En primer lugar, SableCC no incorpora un modelo de objetos uniforme puesto que para realizar el análisis léxico se basa en expresiones

regulares y autómatas finitos deterministas, mezclando su descripción con las especificaciones sintácticas. En segundo lugar, la estructura de SableCC no facilita una buena integración del análisis sintáctico y semántico al carecer del modelo de delegación de eventos presente en el sistema O2C2, esto hace que sea poco modular, dinámica, flexible y no permita la reutilización del código proporcionado por el usuario. Por último, el proyecto no permite generar frameworks, no ofrece un entorno integrado de desarrollo y el tratamiento y recuperación de los errores es deficitario.

ProGrammar

El proyecto ProGrammar^{TM} [Pro01] divide las tareas de desarrollo de un procesador de lenguaje en dos fases.

En la primera fase, se utiliza un entorno integrado de desarrollo para describir la gramática. Este entorno proporciona un componente (*Parse Engine*) que examina el texto fuente y lo analiza de acuerdo con la descripción de las reglas gramaticales generando un árbol sintáctico abstracto que representa los datos del fichero de entrada.

En la segunda fase, los programas cliente deben interactuar con el componente llamando a diferentes métodos en tiempo de ejecución. ProGrammar soporta un conjunto de métodos proporcionados en forma de API para realizar un conjunto de operaciones como cargar gramáticas, analizar datos, realizar un tratamiento de errores, recorrer, buscar y recuperar datos del AST. Está disponible en plataformas Windows (32 bits) a través de controles ActiveX o una librería nativa de C++. Actualmente puede ser utilizado en los entornos Visual C++, Visual Basic, C++ Builder y Delphi.

Para la descripción de las gramáticas debe utilizarse una notación especial proporcionada por un lenguaje que ofrece características orientadas a objetos para describir la estructura de los datos de entrada. Se facilita de esta forma la creación de gramáticas reutilizables que pueden ser extendidas o modificadas.

Es importante señalar que ProGrammar es de aparición muy reciente y se ha incluido en este apartado por ofrecer un entorno visual de desarrollo, no obstante, una funcionalidad similar existe en el sistema O2C2 y las diferencias son notables. El proyecto está limitado por la plataforma de desarrollo, no es posible su utilización a nivel de línea de comandos, maneja su propio lenguaje de descripción de reglas gramaticales y no incorpora técnicas orientadas a objetos de manera uniforme para las distintas fases de desarrollo de un procesador de lenguaje.

12.2.2 Comparación cualitativa

En la tabla 12.1 se presenta una *comparación cualitativa* entre los diferentes sistemas utilizados para la generación de procesadores de lenguajes y el sistema propuesto en esta memoria. En este análisis comparativo se incluyen las diversas técnicas de construcción de procesadores de lenguajes descritas en el capítulo 4 y los dos trabajos presentados en la sección anterior.

Se han tenido en cuenta características que son muy importantes para definir la calidad y eficiencia de un sistema: modularidad, reusabilidad, extensibilidad, mantenimiento, entorno visual de desarrollo, ciclo de desarrollo, tipo de generador (total o parcial) y tratamiento de errores eficaz. Las cuatro primeras, corresponden a características esenciales que deberían ser aplicables en su totalidad para obtener un buen generador. El resto de las características, están relacionadas con otros conceptos que de forma general, pueden ser atribuidos a los sistemas generadores.

La tabla utiliza el siguiente esquema. Si el generador X admite la característica Y, entonces en la celda (X,Y) aparece la letra **S**. Si no admite la característica, aparece la letra **N** y si la admite de forma parcial, se indica mediante la letra **P**.

GENERADOR DE PROCESADORES DE LENGUAJES	Modularidad	Reusabilidad	Extensibilidad	Mantenimiento	Ciclo de desarrollo corto	Entorno visual de desarrollo	Generador total	Tratamiento de errores eficaz
Accent	N	N	N	N	N	N	P	N
Antlr	N	N	N	N	N	N	P	S
Cocktail	N	N	N	N	N	N	P	P
Eli	P	P	P	P	N	N	S	P
Gentle	N	N	N	N	N	N	P	N
JavaCC	N	N	N	N	N	N	P	S
Lemon	N	N	N	N	N	N	P	N
Lex/Yacc	N	N	N	N	N	N	P	N
ProGrammar	P	P	P	S	S	S	P	P
SableCC	P	P	P	S	S	N	P	N
Spirit	N	P	P	P	N	N	P	N
Sistema O2C2	**S**	**S**	**S**	**S**	**S**	**S**	**P**	**S**

Figura 12.1 Comparación entre diferentes Generadores de Procesadores de Lenguajes

A continuación se describen cada una de las características tenidas en cuenta para su valoración conjunta.

- *Modularidad.* Si se logra separar las especificaciones del código de acción semántica, se permitiría encapsular los detalles de la implementación, ayudando de esta forma a entender y manejar mejor las aplicaciones generadas, ya que cualquier cambio realizado en la especificación no modificaría el contenido semántico y a la inversa. En este sentido, *SableCC* permite separar las especificaciones del código proporcionado por el usuario pero esta modularidad es parcial ya que todo el código

semántico debe estar integrado en una sola clase. *ProGrammar* separa la especificación gramatical del código semántico, puesto que son las aplicaciones cliente las encargadas de decidir que hacer con el AST generado, llamando a los métodos proporcionados en forma de API desde las plataformas disponibles. *Eli* es un sistema que describe la naturaleza de un problema, usando solamente especificaciones declarativas, y hace uso de un conjunto de herramientas para traducir dichas especificaciones a código máquina. No obstante, es importante señalar que utiliza un conjunto amplio de especificaciones que deben ser descritas por diversos lenguajes especializados. El resto de los sistemas mezclan código de acción y especificaciones.

- *Reusabilidad.* Se consigue reutilizando el propio código semántico y parte de las especificaciones léxicas y sintácticas. Cuatro son los sistemas revisados que incorporan parcialmente esta característica. *Eli* da soporte a una librería de especificaciones reusable. *Spirit* resuelve de forma parcial este problema, permitiendo crear analizadores de mayor complejidad a través de la composición de clases primitivas. *SableCC* junta especificaciones léxicas y sintácticas en un solo fichero de entrada, pero dificulta la reutilización del código semántico al estar integrado en una única clase. *ProGrammar* permite que las gramáticas puedan ser extendidas o modificadas, y los analizadores sintácticos generados, pueden ser utilizados por las aplicaciones cliente en entornos diferentes.

- *Extensibilidad.* Se puede incorporar proporcionando mecanismos que permitan extender el sistema para adaptar su comportamiento a las nuevas condiciones impuestas. En este sentido, los sistemas que apliquen técnicas orientadas a objetos podrán resolver mejor esta característica. Los sistemas *Spirit* y *SableCC*, incorporan un conjunto de clases que se puede extender para adaptar su comportamiento, sin embargo, estos sistemas no permiten generar frameworks. *Eli* permite extender su librería de especificaciones y *ProGrammar* también puede adaptar su comportamiento al proporcionar un conjunto de métodos en forma de API.

- *Mantenimiento.* Permite modificar las aplicaciones añadiendo o quitando funcionalidad, pero esta tarea será más sencilla si se consiguen construir aplicaciones modulares y basadas en técnicas orientadas a objetos. En general, debido a la estructura rígida de los sistemas tratados, este problema no está resuelto en la mayoría de los casos. Una solución parcial estaría proporcionada por el sistema *Eli*, que mantiene una librería de especificaciones aunque no incorpora características orientadas a objetos y el sistema *Spirit*, que proporciona un pequeño grupo de clases primitivas. Las propiedades que incorporan los sistemas *ProGrammar* y *SableCC* se adaptan mejor para dar solución a este problema.

- *Ciclo de desarrollo.* Un ciclo de desarrollo de software será largo si el sistema generador deja al programador la tarea de mantener actualizados el fichero de especificaciones y el programa resultante, por el hecho de mezclar el código semántico con las especificaciones gramaticales. Aunque el sistema *Eli* usa un conjunto amplio de especificaciones

declarativas, y el código es generado por las herramientas internas que incorpora, se deben modificar las especificaciones si el código de acción generado no es el adecuado. *SableCC* y *ProGrammar* disminuyen el ciclo de desarrollo al conseguir separar las especificaciones del tratamiento semántico concreto.

- *Entorno visual.* De los sistemas analizados solamente *ProGrammar* ofrece un entorno visual para el desarrollo de procesadores de lenguajes.

- *Generador total.* Se considera un generador total si ayuda a construir procesadores de lenguajes completos, realizan el análisis y la generación-optimización de código para máquinas reales. De los sistema revisados solamente el sistema *Eli* trata todas las fases de construcción de un compilador (dirigido por un sistema experto).

- *Tratamiento de errores eficaz.* Se deben detectar los posibles errores en las especificaciones y en las aplicaciones generadas, para que puedan recuperarse y seguir funcionando correctamente, proporcionando para ello mecanismos que ayuden a localizar y corregir dichos errores. De los sistemas revisados, solo *Antlr* y *JavaCC* han prestado gran importancia a esta característica, *Eli, Cocktail y SablerCC* proporcionan un tratamiento parcial.

12.3 Algunos resultados destacables

La incorporación al sistema de las características tenidas en cuenta en la valoración anterior supone una mejora de la usabilidad [Goo87, Gou85, Sun01]. Es un concepto que muchas personas asocian solamente a las aplicaciones cuya interfaz de usuario juega un papel importante. Sin embargo, las ventajas de considerar la usabilidad en los generadores de procesadores de lenguajes se traduce en un incremento de la calidad del software generado.

A continuación se resumen algunos resultados que se pueden destacar agrupados en los seis atributos que comprenden la idea de la usabilidad: *utilidad, facilidad de aprendizaje, eficiencia, retención de ideas, errores y satisfacción*.

12.3.1 Utilidad

El primer objetivo de un sistema es ser útil para algún propósito. Con el sistema diseñado se consigue una utilidad análoga a la de los sistemas revisados, sin embargo, si antes los sistemas estaban más orientados a la construcción de compiladores e intérpretes, hoy en día la utilidad de los procesadores de lenguajes es mucho más amplia. El sistema propuesto está pensado para desarrollar procesadores de lenguaje más ubicuos y dar solución a numerosas tareas como el tratamiento de ficheros de configuración complejos, analizadores XML, editores con sintaxis coloreada, etc. Al generar procesadores basados en jerarquías de

clases, también están más preparados para ser la base de aplicaciones software grandes y complejas.

12.3.2 Facilidad de aprendizaje

La construcción de generadores de procesadores de lenguajes está pensada para que puedan ser utilizados por personas que posean unos conocimientos teóricos mínimos. Sin embargo, para que puedan ser usados con éxito, tienen que ser fáciles de aprender, y esto se consigue si están basados en un grupo pequeño de conceptos.

En este punto existen algunas diferencias entre el sistema propuesto y los sistemas revisados. Las especificaciones de los generadores revisados son en general bastante complejas, lo cual supone una barrera de entrada al usuario que carezca de experiencia previa. Para solventar este problema, el sistema diseñado limita la especificación de entrada a la información mínima necesaria, concentrándose en la propia estructura gramatical y en el conjunto de tokens a reconocer.

Además, al incluir el sistema un entorno visual de desarrollo, es mucho más sencillo para un usuario no experimentado en este campo, comenzar a crear procesadores de lenguajes funcionalmente completos.

12.3.3 Eficiencia

Como ya se comentó en el capítulo 11, la eficiencia se puede enfocar desde dos puntos de vista. En primer lugar, si se mide la eficiencia teniendo en cuenta la velocidad de ejecución del programa y el consumo de recursos, es probable que muchos de los sistemas revisados anteriormente puedan ser considerados más eficientes. Esta diferencia tiene varias explicaciones, por un lado es natural que el código del sistema O2C2 de reciente creación, no sea tan maduro y esté menos optimizado que el de otros sistemas de aparición anterior. Por otro lado, el que el sistema esté constantemente haciendo llamadas a métodos de clases diferentes en vez de llamadas a funciones supone una penalización de la eficiencia del sistema. Sin embargo estas diferencias no son lo bastante importantes como para que puedan ser apreciadas durante la implementación de proyectos con las características de los ordenadores actuales.

En segundo lugar, si se mide la eficiencia de un generador de procesadores de lenguajes en base a lo eficiente que convierta a un usuario con el uso de dicho generador, el sistema O2C2 es claramente más eficiente puesto que permite realizar más cómodamente y con más eficiencia el trabajo de desarrollar un procesador de lenguaje. Una apreciación más objetiva es el hecho de que el sistema O2C2 separa el código de usuario del código que genera automáticamente, por tanto, no es necesario utilizar la herramienta constantemente. Solamente será usada cuando se produzca un cambio en la especificación gramatical. Al conseguir un ciclo de desarrollo más corto hace posible que el sistema propuesto sea más eficiente.

12.3.4 Retención de ideas

Cuando una aplicación fuerza al usuario a tratar con muchos datos de forma simultánea, o cuando éste tiene que ocuparse de muchos detalles para crear un procesador de lenguaje, es señal de que la abstracción que proporciona el sistema utilizado no es lo suficientemente buena.

En los generadores tradicionales, este problema tiene especial relevancia puesto que la especificación gramatical incluye también el código de acción semántica. Esto significa que incluso para lenguajes sencillos, las especificaciones rápidamente se vuelven grandes y complejas, por lo que el usuario puede llegar a perderse entre tanto detalle.

Como ya se comentó en el apartado anterior, el sistema O2C2 separa la especificación gramatical y el código de acción que proporciona el usuario. El sistema pretende ser simple y esa simplicidad se traduce en que es posible generar un procesador de lenguaje completo con una interacción mínima por parte del usuario.

Por otra parte, con el objeto de mejorar el grado de retención de ideas, el número de conceptos necesarios para poder usar el sistema son mínimos y además, estar basado en un modelo orientado a objetos (frameworks y delegación de eventos), conocido por la mayor parte de los usuarios.

12.3.5 Errores

El tratamiento y recuperación de errores no ha recibido mucha atención por los diseñadores de generadores tradicionales.

Para llevar a cabo un tratamiento y recuperación de errores efectivo es necesario distinguir dos tipos de usuarios. En primer lugar, se encuentran los usuarios del propio sistema que tienen que tratar con las especificaciones requeridas y modo de operar del generador. En este caso, no solo se deben detectar los propios errores de la especificación, sino que además es necesario proporcionar mecanismos que ayuden a localizar y corregir dichos errores. En segundo lugar, se encuentran los usuarios de los procesadores de lenguajes generados (*usuario final*). Si el generador crea aplicaciones en las que resulta difícil incorporar código para el tratamiento de los errores que pudieran producirse, es muy probable que no se añada este código especializado, y por consiguiente, resultará difícil para una aplicación recuperarse de los errores y seguir funcionando correctamente.

El tratamiento y recuperación de errores en el sistema O2C2 usa el mismo modelo de delegación de eventos (*patrón listener*) que para el código de acción semántica. De esta forma, es muy fácil, de cara al usuario, añadir código especializado para conseguir que los procesadores generados sean tolerantes a los fallos que pudieran producirse. Por otra parte, también detecta los errores producidos en la entrada al generador facilitando su localización y corrección.

12.3.6 Satisfacción

Este último atributo de la usabilidad es difícil de medir objetivamente, puesto que lo que para un usuario puede ser una ventaja, para otro puede ser un inconveniente. No obstante, una aplicación con un comportamiento seguro e intuitivo será satisfactoria para la mayor parte de los usuarios.

Uno de los objetivos que se debe alcanzar cuando se construye un generador de procesadores de lenguajes es que no cause frustración al usuario. La mayoría de las frustraciones están originadas por una excesiva e innecesaria complejidad de las especificaciones y del sistema, a una falta de documentación y ejemplos para su utilización de forma adecuada, y a un comportamiento inesperado de las aplicaciones generadas.

En el sistema O2C2 las especificaciones que se deben aportar son simples, el comportamiento de los procesadores de lenguajes que genera y del propio sistema es fácil de entender, los conceptos que usa son conocidos por la mayoría de los programadores, y la documentación proporcionada intenta ser lo más pedagógica posible. Además, tiene el potencial de incrementar la productividad de los programadores y la calidad de sus productos, porque ha sido diseñado con técnicas orientadas a objetos teniendo en cuenta la usabilidad, lo que permitiría ser usado de forma satisfactoria en proyectos de media y gran escala.

12.4 Trabajo y líneas de investigación futuras

La tesis presentada deja abiertos diferentes trabajos y líneas de investigación, para mejorar y completar lo ya existente, y para desarrollar nuevas facetas. A continuación se señalan las líneas de investigación más inmediatas relacionadas con los temas discutidos en esta tesis.

12.4.1 Generalización del tipo de análisis

Debido a que el tipo de análisis realizado en la versión inicial del prototipo es LL(1), una tarea inmediata consistiría en aplicar su generalización para obtener analizadores LL(k). Su implementación práctica permitiría mejorar el conjunto de gramáticas libres de contexto reconocibles por el sistema.

Por otro lado, también sería posible extender el sistema para permitir un análisis ascendente a través de la implementación de tablas de análisis LALR. Este análisis podría realizarse aplicando las mismas técnicas orientadas a objetos, y permitiría trabajar con gramáticas cuya especificación requiera un mayor grado de flexibilidad, impuesta a su vez por el tipo de lenguaje que describa.

12.4.2 Dar soporte al resto de las funcionalidades

Cuando se describió la arquitectura del sistema, se identificaron los principales módulos que componían el sistema pero no se describieron los bloques que

dependían directamente de la fase de síntesis de un compilador y por consiguiente de la máquina objeto. Sería interesante poder añadir esta funcionalidad para permitir que el sistema se comporte como un generador total, permitiendo de esta forma desarrollar compiladores optimizados para máquinas reales.

12.4.3 Mejora en el entorno visual de desarrollo

La construcción de un entorno de desarrollo más completo es otro aspecto interesante. Se trataría de estudiar la mejor manera de seleccionar los objetos encargados de reconocer los diferentes símbolos del lenguaje y de proporcionar diagramas gráficos que permitan introducir las reglas de la gramática. Este entorno operaría en un sentido bidireccional, de forma que el usuario podría hacer modificaciones tanto a escala visual como en el ámbito de la especificación textual y no existirían problemas de sincronización. En este sentido se podría favorecer la reutilización de estructuras gramaticales y componentes léxicos mediante la encapsulación de los mismos como componentes.

12.4.4 Implementación de un sistema de tipos

El propósito fundamental de un sistema de tipos consiste en prevenir la aparición de errores durante la ejecución de un programa. Sin embargo, esta ausencia de errores no es una propiedad trivial y como consecuencia, la clasificación, descripción, y estudio de los sistemas de tipos, ha emergido como una disciplina formal.

Los sistemas de tipos proporcionan herramientas conceptuales mediante las cuales es posible juzgar la adecuación de aspectos importantes en las definiciones de los lenguajes. Muchas descripciones de los lenguajes ofrecen especificaciones poco detalladas de sus estructuras, permitiendo implementaciones ambiguas en las que distintos compiladores para un mismo lenguaje, implementan sistemas de tipos diferentes. De una forma ideal, la solución pasaría por que los sistemas de tipos formaran parte de la definición de todos los lenguajes de programación con tipos (*typed languages*). Solamente los programas que cumplan con el sistema de tipos especificado podrán ser ejecutados, el resto de los programas serían rechazados.

Una línea de investigación interesante consistiría en desarrollar un sistema de tipos genérico, integrado con el sistema propuesto y capaz de proporcionar los medios adecuados que permitan incorporar la seguridad en los programas a través de las **comprobaciones estáticas**. Un sistema de tipos bien diseñado debe garantizar el buen comportamiento de los programas, capturando los errores antes de que se produzcan durante su ejecución.

12.4.5 Sistema de prototipado de lenguajes

Existe un creciente interés en el desarrollo modular de lenguajes de programación con el propósito de obtener rápidamente prototipos de lenguajes de dominio específico. Esta modularidad puede ser lograda identificando los

diferentes aspectos del procesador de lenguaje [MPV99] o mediante la utilización de mónadas. El concepto de mónada ha sido adaptado de la Teoría de Categorías al desarrollo modular de especificaciones semánticas de lenguajes de programación. Recientemente, el autor de esta tesis doctoral ha participado en el desarrollo de un Sistema de Prototipado de Lenguajes [LCL+01a]. Dicho sistema ha sido aplicado a la especificación de lenguajes funcionales [LCL+01b,LCL+01c], imperativos [LCL+01a], lógicos [LCL+01d] y orientados a objetos [Lab01].

El sistema de prototipado de lenguajes parte de un AST ya generado sin tener en cuenta el proceso de obtención de dicho árbol. Mediante el sistema O2C2 sería posible asociar las reglas semánticas al árbol generado, consiguiendo un sistema modular de creación de procesadores de lenguajes a partir de su especificación semántica.

Anexo A

Manual de Usuario del Entorno Integrado de Desarrollo O2C2ui

A.1 Introducción

Como se pudo observar en la descripción general del prototipo del capítulo 9, su estructura integra varios subsistemas que dan soporte a la funcionalidad básica del prototipo permitiendo generar procesadores de lenguajes. La función del subsistema O2C2ui, consiste en facilitar al usuario, especialmente a los menos experimentados, el uso del propio prototipo para construir de forma integral un procesador de lenguaje completo y funcional. No obstante, se supone que el usuario posee un conocimiento, al menos básico, de la teoría de procesadores de lenguajes.

Este subsistema constituye por tanto, el entorno visual de desarrollo y intenta proporcionar las condiciones de un entorno integrado (edición, compilación y montaje, gestión de proyectos, manejo de errores, etc.) aplicadas al desarrollo de programas.

Se ha seguido para su elaboración, la guía de estilo de aplicaciones Java[1], procurando que la aplicación gráfica sea coherente con otras herramientas Java, fácil de utilizar, intuitiva y estéticamente agradable.

A.2 Descripción general del entorno

El entorno de desarrollo muestra una configuración de acceso inicial al sistema como la mostrada en la *figura A.1*.

El entorno consta de las siguientes partes:

- Un **área de título**, con el cual se puede mover la ventana en la que está la aplicación. A su izquierda se encuentra el menú de control de la ventana

[1] Uso del framework Swing, una jerarquía de clases para la construcción de interfaces que utiliza el modelo MVC (Model-View-Controller).

(con el que se puede cerrar la aplicación, mover, cambiar de tamaño, etc.) y a su derecha los botones de minimizar y maximizar. La aplicación comienza inicialmente maximizada.

- La **barra de menús**, donde se encuentran las principales opciones del programa clasificadas por funciones.

- Una **barra de control**, que simplifica la elección de las opciones más usuales del programa sin necesidad de moverse por los menús.

- Un **área de trabajo**, sobre el que se sitúan las diferentes ventanas asociadas a una determinada aplicación. El área de trabajo se tratará posteriormente con mayor detalle.

- Una **barra de mensajes**, en donde se proporciona información adicional.

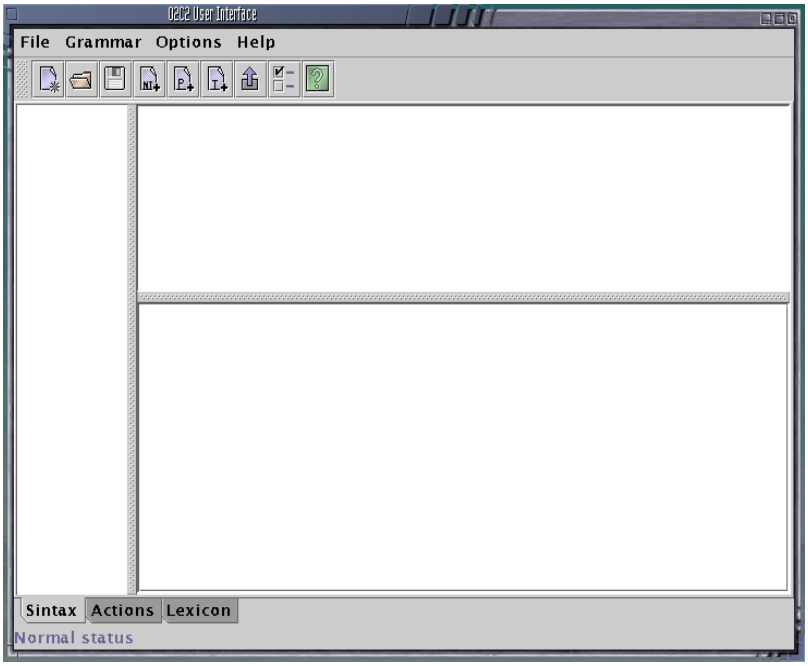


Figura A.1 Ventana principal de O2C2ui

A.3 Introducción de la gramática

El objetivo principal de O2C2ui es crear procesadores de lenguaje completos y funcionales. Para lograr esta tarea, la interfaz de usuario se centra alrededor del análisis sintáctico. Es importante por tanto, comprender cómo se debe introducir la gramática que describe la sintaxis del lenguaje.

En la versión actual del prototipo, O2C2ui procesa gramáticas descritas a través de la notación EBNF, realiza un análisis descendente y comprueba que las gramáticas cumplen la condición LL(1). Es necesario destacar que a diferencia de otros sistemas de características análogas, una especificación de entrada al sistema O2C2 es muy simple, puesto que sólo contiene la sintaxis de la gramática que se

quiere tratar acompañada opcionalmente de enlaces (etiquetas delimitadas por comillas dobles).

Un ejemplo sencillo de cómo puede ser esta gramática es el siguiente:

```
<S> ::= <num> OP <num> "operacion";
<num> ::= DIGITO {DIGITO} "numero";
```

Este ejemplo muestra algunas características de las entradas que acepta el sistema. En primer lugar, se distinguen los símbolos *terminales* (opcionalmente en mayúsculas) de los símbolos *no terminales* en que estos últimos van acompañados de los caracteres "<" y ">".Un símbolo no terminal puede tener varias producciones separadas por el carácter "|".

El mecanismo sobre el que se basa el subsistema O2C2Conv, excluye la necesidad de proporcionar un nombre para cada una de las producciones que pueda tener un elemento no terminal en la gramática. Esto es debido a que con el sistema de anclajes, no es necesario que el usuario trate directamente con las clases generadas por el prototipo.

En contrapartida, el usuario tendrá que poner **anclas**, cadenas de texto entrecomilladas, en los puntos en los que quiere atender a los eventos que se puedan generar al ser reducidas las expresiones gramaticales a su izquierda.

En el ejemplo, se ha colocado el ancla `"operación"`, lo cual significa que el usuario está interesado en el evento que se generará cuando ante una entrada que siga las reglas de la gramática aquí especificada, se reduzca la expresión.

Para que una secuencia de símbolos de la gramática pueda aparecer un número no determinado de veces, se acompaña dicha secuencia por "{" y "}". Así `{DIGITO}`, siendo `DIGITO` un token proporcionado por el analizador léxico, suponiendo que se refiera a un dígito (0-9), hará que el analizador léxico generado entienda que tiene que leer cualquier secuencia de dígitos, un número indeterminado de veces.

A.3.1 Palabras reservadas.

Los símbolos *terminales* y *no terminales* pueden tener prácticamente cualquier nombre siempre y cuando no contenga espacios y utilice caracteres Unicode. Además, el prototipo para algunas operaciones intermedias genera producciones y símbolos *no terminales* que no tienen mayor repercusión para el usuario, aunque impide el uso de variables que se llamen o3_vacio o que empiecen por o3_tmp.

A.3.2 Creación de un símbolo no terminal

Para introducir un símbolo no terminal en la gramática se pueden utilizar dos formas: recurriendo a la entrada *New non terminal* del menu *Grammar* o pulsando el botón que contiene la palabra *NT()*. Como resultado de esa operación aparecerá el diálogo de la *figura A.2*.

Figura A.2 Ventana para la creación de no terminales

Es importante tener en cuenta que el primer símbolo no terminal de la gramática se considera el símbolo inicial desde el cual comenzará el análisis descendente del procesador de lenguaje que se genere.

A.3.3 Creación de las producciones

Una vez que se tenga al menos un símbolo no terminal, aunque es posible introducir todos los *no terminales* de la gramática mediante este procedimiento, ya es posible introducir las producciones asociadas a estos símbolos.

Para introducir una nueva producción se puede recurrir al menú (*Grammar -> New production*) o mediante el botón que contiene la letra P ( ). Esta operación da lugar al siguiente dialogo de la *figura A.3*.

Figura A.3 Ventana para la creación de nuevas producciones

En la parte superior del mismo se introduce la producción siguiendo la notación descrita anteriormente. Debajo de la producción se selecciona el símbolo no terminal del cual la producción formará parte, dentro una lista de todos los *no terminales* existentes en la gramática,.

Con este diálogo sólo se puede introducir una producción del símbolo no terminal, no es posible introducir varias producciones. Si fuera necesario introducir varias producciones, se repetiría esta operación tantas veces como fuera necesario.

Si se produjese algún error introduciendo esta información en la gramática se podría corregir más adelante de forma fácil y sencilla.

A.3.4 Repercusiones

Como resultado de introducir producciones, se pueden estar introduciendo cambios tanto en la gramática que describe la sintaxis, como en el análisis léxico y semántico. De esta forma, cada producción que contenga *anclas* hará que el programa considere las anclas en la pestaña del análisis semántico (pestaña etiquetada como *actions*). Igualmente, todas los elementos *terminales* o *tokens*

que aparezcan en una producción serán recogidos en la pestaña del análisis léxico (etiquetada en inglés como *lexicon*).

Cuando se tenga seleccionada la pestaña *sintax*, todos los símbolos *no terminales* que hayan aparecido en la gramática, bien por haber sido introducidos explícitamente o por haber aparecido dentro de una producción, aparecerán en una lista situada en el extremo izquierdo de la pantalla.

Cuando se seleccione un símbolo no terminal de esta lista, todas las producciones de este elemento aparecerán situadas a su lado, en el extremo superior derecho. Cuando se seleccione una de estas producciones, el contenido de la misma aparecerá en la parte inferior derecha de la pantalla (*figura A.4*).

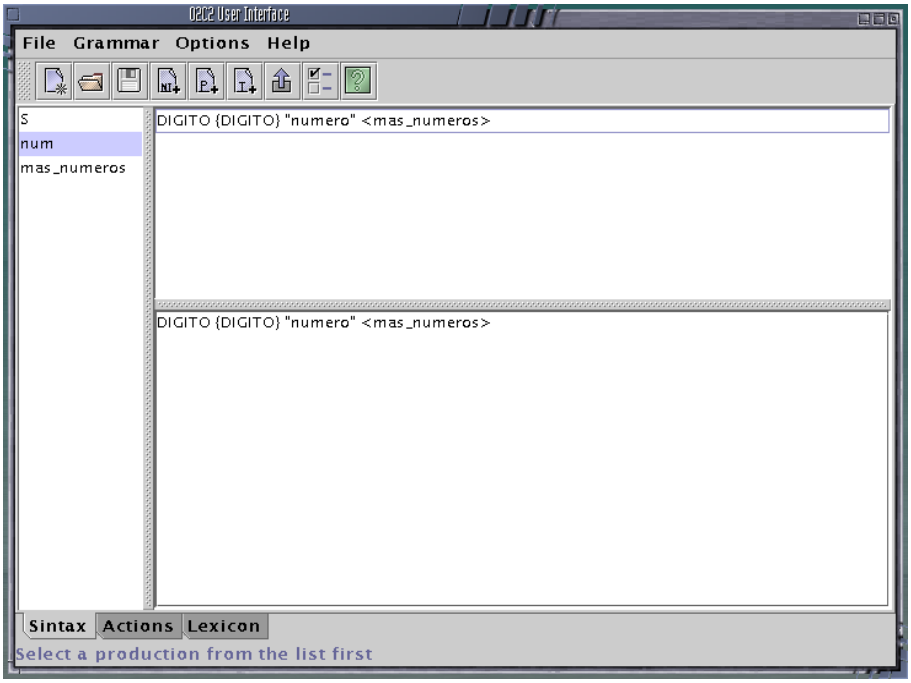


Figura A.4 Ventana principal de O2C2ui

Las producciones así seleccionadas son modificables, por lo que si existe alguna razón por la que se quiera modificar el contenido de una producción que ya ha sido incluida en la gramática, éste es el lugar adecuado para realizarlo. Si en la modificación se introducen nuevos símbolos *terminales*, *no terminales* o *anclas*, también se incluirán dentro del sistema para poder ser referenciados en el tratamiento léxico, sintáctico y semántico.

Para que las modificaciones sobre una producción gramatical tengan efecto no es necesario pulsar ningún botón de confirmación, tan sólo es necesario dejar de tener activo el texto que se está editando y pasar a tener activo cualquier otro punto de la ventana.

Es posible borrar los elementos *no terminales* así como sus producciones, utilizando el menú *Grammar* o sencillamente pulsando el botón de *suprimir* dentro de la lista de elementos teniendo seleccionado el elemento que se quiera eliminar.

A.4 Introducción del léxico

No es necesario introducir de manera explícita los elementos que componen el léxico del lenguaje. Cada vez que se introduce una producción, como se ha podido ver en la sección anterior, todos los elementos *terminales* definidos, son reconocidos y aparecen automáticamente en la pestaña dedicada al análisis léxico.

Cuando se selecciona la pestaña del análisis léxico (etiquetada *lexicon*) el resultado es el que se muestra en la ventana de la *figura A.5*.

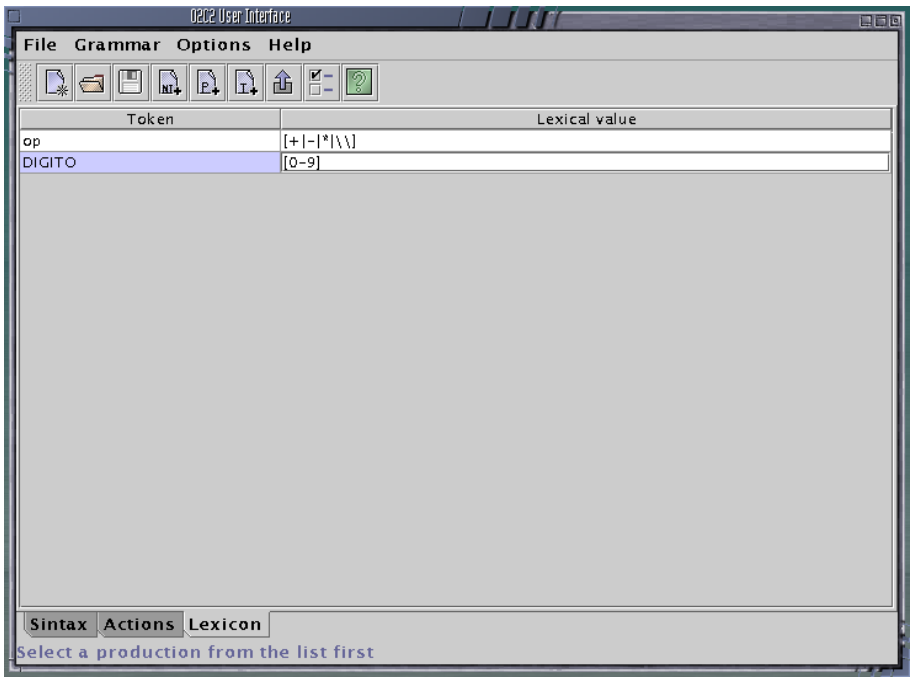


Figura A.5 Ventana de especificación del léxico

Como se puede observar, en el extremo izquierdo aparecen los símbolos *terminales*, definidos en la gramática de ejemplo. A su lado, a la derecha, aparecen las definiciones léxicas asociadas a éstos. El contenido de dichas definiciones son las expresiones regulares asociadas a los símbolos *terminales* de la gramática.

Las expresiones regulares pueden definirse de varias formas dentro de este entorno, siempre y cuando se corresponda con la notación que es capaz de entender JFLEX [Jfl01]. Suponiendo que **a** y **b** sean ya expresiones regulares:

- **a | b** es otra expresión regular que agrupa a cualquier expresión que sea reconocida por a o b.

- **a b** hace referencia a entradas que agrupan a cualquier expresión que sea reconocida por a y b.

- **a*** es otra expresión regular que agrupa cero o más repeticiones de la entrada identificable por la expresión regular a.

- **a+** es equivalente a aa*.
- **a?** hace referencia a una entrada vacía o una entrada identificable por a.
- **a (4)** representa la expresión regular que identificaría cuatro veces la entrada reconocible por a.
- **(a)** es equivalente a la expresión regular a.

Es importante recordar que los cambios se toman en consideración cuando se pasa el foco de la aplicación a otro punto del entorno, por ejemplo, señalando a otro elemento léxico.

Aunque todos los elementos *terminales* de la gramática, aparecen en la pestaña del léxico de forma automática, es posible que por alguna razón especial, el usuario necesite introducir nuevos elementos léxicos cuya utilidad sea servir de apoyo a los existentes.

Para introducir nuevos elementos léxicos en el sistema, se puede recurrir a la opción del menú *Grammar->New Token* o pulsar el botón que contiene la letra T en el icono ( ). El resultado es el que se muestra en cuadro de dialogo de la *figura A.6*.

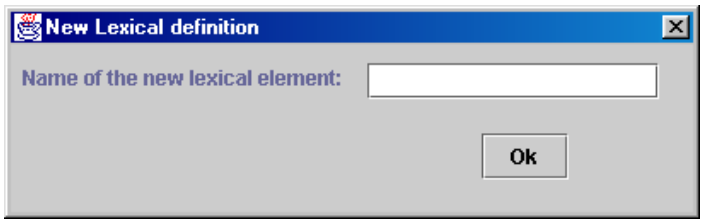


Figura A.6 Ventana para añadir un nuevo elemento léxico que no se corresponde con un terminal de la gramática

Una vez obtenidas las especificaciones léxicas, con la información presentada en la ventana asociada a los componentes léxicos, O2C2ui generará una entrada para el programa JFLEX [Jfl01] que ha de estar instalado en el sistema, el cual se encargará de generar el correspondiente analizador léxico.

A.5 Creación de clases de usuario

Con las operaciones descritas anteriormente, el entorno permite generar un analizador sintáctico capaz de integrarse con el analizador léxico proporcionado por una herramienta externa. Con la combinación de ambos analizadores se obtiene un procesador de lenguaje capaz de reconocer y analizar distintas entradas, sin embargo, debe ser el usuario el encargado de proporcionar el código de acción semántica necesario para completar el procesador de lenguaje generado. En el caso de un compilador podría ser generar código para una máquina determinada, en el caso de un editor con sintaxis coloreada podría ser decidir que color corresponde en un momento dado.

El proceso de análisis semántico se realiza de la siguiente forma. Cuando el analizador sintáctico procesa una entrada que puede ser asociada a una sección de la gramática, se dice que se ha producido una reducción de esa expresión gramatical. Si la expresión gramatical reducida se encuentra inmediatamente a la izquierda de un *ancla* definida por el usuario, entonces esta reducción gramatical da lugar a un evento que puede ser escuchado por el usuario.

A.5.1 Edición de listeners

El mecanismo que se aplica para que el usuario pueda añadir código que se encargue de dar un contenido semántico al procesador de lenguaje generado por el entorno, es el de los **Listeners**. A través de este mecanismo, diversas clases se subscriben a una clase asociada a una determinada ancla incluida en la gramática por el usuario, y está entiende que aquellas son listeners. Evidentemente una misma clase del analizador que represente un ancla de la gramática puede tener registrados varios subscriptores, y un listener puede estar subscrito a eventos de varias anclas.

La idea es que el código que añade el usuario sea incluido en esas clases *listeners,* en vez de incluirlo directamente en las clases sintácticas que van a ser generadas de forma automática por el sistema.

Mediante este **sistema de delegación de eventos** gramaticales, los eventos son generados por las anclas incluidas en la gramática. Así, cuando se ha conseguido asociar una serie de símbolos *terminales* obtenidos del análisis léxico con un ancla colocada a la derecha de una sección de la gramática y dicha sección es contemplada por una clase del analizador sintáctico, se dice que se ha reducido una expresión, y este hecho es el que genera un evento.

Las clases que quieran utilizar esta técnica tienen que implementar la interfaz Listener del paquete o2c2rt. Esta interfaz obliga a implementar el método:

```
void performAction (Node n)
```

Este es el método que invoca la clase semántica cuando desea efectuar algún trabajo. El parámetro del método hace referencia a la clase semántica que actúa como listener, es decir, al nodo objeto que invocó dicho método.

Dentro del código de este método, es importante crear una instancia de la clase a la que nos hemos subscrito. Esto es debido a que los datos que se van a utilizar en dicha clase son estáticos y están presentes en cualquier instancia de la clase.

Es posible acceder a los elementos de una producción, con la que se desea operar en un momento dado, a partir de variables identificadas como f0, f1, etc. Estas variables, son a su vez objetos que forman parte del analizador sintáctico de los que se puede obtener su valor con el método **getValue(int)** y cambiarlo con el método **setValue()**.

A.5.2 Integración de listeners en el analizador

Todas las anclas incluidas en la gramática aparecen en la pestaña etiquetada como *Actions*. Si se ha definido un ancla en la gramática, se entiende que el usuario está interesado en captar el evento sintáctico asociado con la reducción de la sección gramatical situada a la izquierda de donde se haya colocado dicho ancla.

En principio, cada ancla tiene asociado un listener, que es la clase en donde el usuario introducirá el código de acción asociado a un sección gramatical. El contenido de la ventana cuando se tiene seleccionada la pestaña *Actions* es el que se muestra en la *figura A.7*.

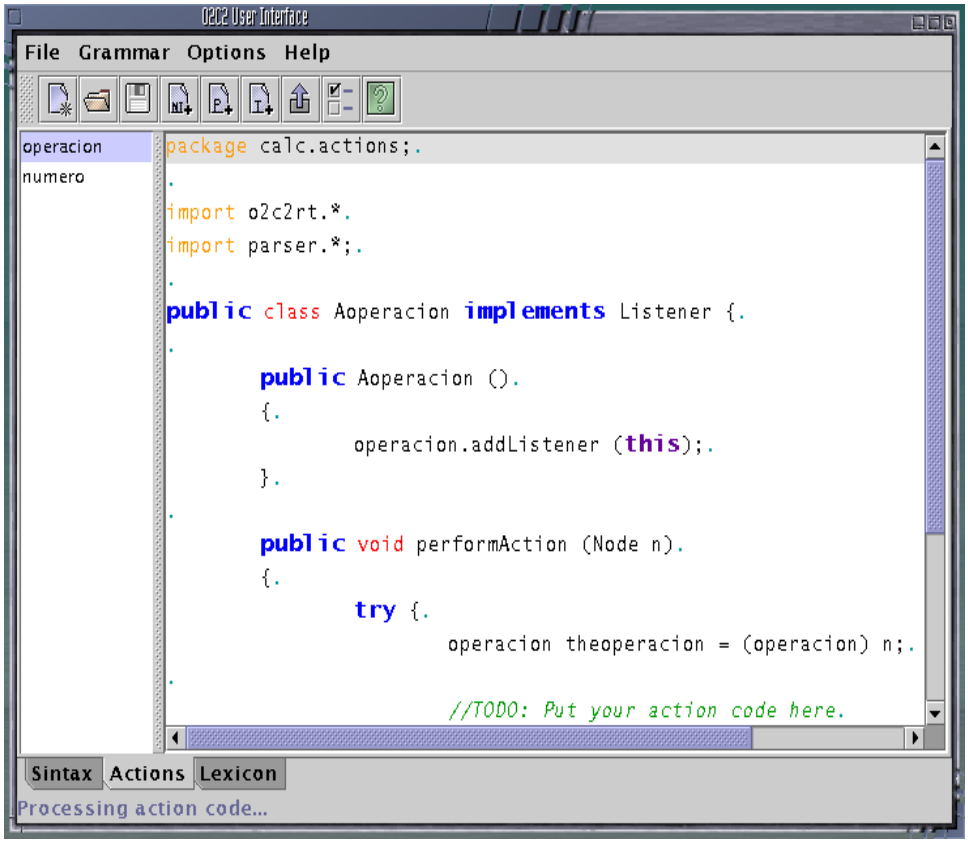


Figura A.7 Ventana con la especificación semántica y código de acciones

En el extremo izquierdo aparecen los diferentes anclas definidos en la gramática mientras que en el lado derecho aparece el código de acción asociado a un ancla en forma de una clase **listener.** En caso de que el usuario no hubiera creado previamente dicha clase, O2C2ui generará de forma automática un esqueleto de la misma para facilitar la labor en el momento en que se activa un ancla.

No obstante, antes de crear un listener de forma automática es necesario proporcionar al entorno O2C2ui alguna información sobre el proyecto, y en este punto es importante conocer el nombre que el usuario quiere dar al paquete bajo el cual se agruparán las acciones asociadas a los anclas.

A.6 Otras tareas

En los apartados anteriores se han descrito las tareas más importantes para definir todos los aspectos necesarios en la creación de un procesador de lenguaje utilizando el entorno visual de desarrollo. A continuación se explicarán otras características de O2C2ui.

A.6.1 Parámetros del proyecto

El proyecto tiene un conjunto de características referidas a los paquetes y a los directorios que se van a generar a partir del proyecto. Esta información es requerida en diversas opciones del entorno y cuando no se presente el dialogo de forma automática, es posible llegar a él mediante la entrada del menú *File->New* o mediante el botón de propiedades( ) (*figura A.8*).

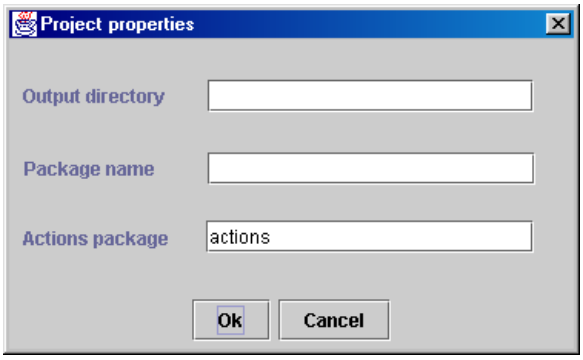


Figura A.8 Propiedades de un proyecto

En **Output directory** se especifica el directorio a partir del cual se van a crear nuevos subdirectorios y los ficheros del proyecto.

En **Package name** se especifica el paquete bajo el cual se van a colocar todas las clases del procesador de lenguaje generado.

En **Actions package** se especifica el paquete, que a su vez va a ser subpaquete del paquete definido en la línea anterior.

A.6.2 Gestión de proyectos

Es bastante improbable que un proyecto de tamaño grande se pueda realizar en una sola jornada de trabajo, resulta por tanto necesario que el sistema permita almacenar la información del trabajo realizado para poder proseguir en otro momento.

A pesar de que el contenido de un proyecto hace referencia a varios ficheros, cuando se almacena el contenido de un proyecto, este se almacena en un sólo fichero que utiliza un lenguaje de marcas conforme al estándar XML [BPS00]. Este fichero puede luego ser transportado a cualquier otro entorno y contiene toda

la información necesaria para continuar con el proyecto en el punto donde se dejo, no es necesario disponer de otros ficheros adicionales.

Para almacenar el contenido de un proyecto se puede recurrir tanto al botón de la barra de iconos con forma de disquete (), como a la entrada de menu *File->Save* o *File->Save As*. En caso de que haya que dar un nombre al fichero, se recomienda uno con la extensión .o2c2prj.

Para abrir un proyecto, es posible seguir un proceso análogo. Se puede recurrir al botón de la barra de iconos o mediante la entrada de menu *File->Open* en cuyo caso se abrirá una imagen como la mostrada en la *figura A.9*.

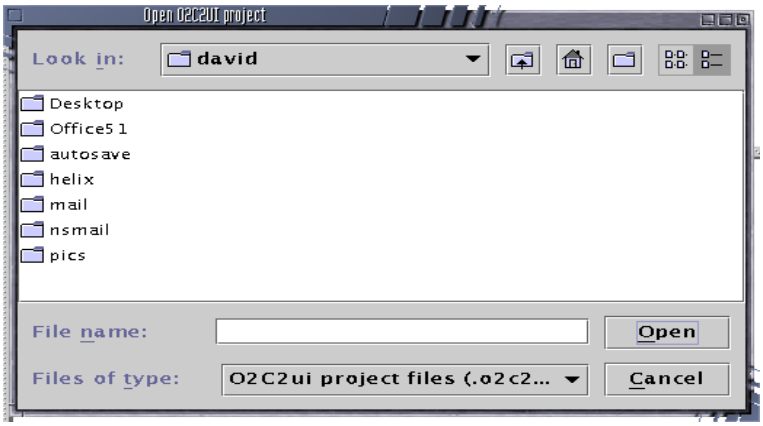


Figura A.9 Abriendo un fichero de proyecto

A.6.3 Generando un proyecto

Cuando se ha completado un proyecto y se quieren probar los resultados, o cuando se quiere comprobar la evolución del mismo, se necesitan generar todos los ficheros que componen el procesador de lenguaje que se está construyendo.

Para ello se debe pulsar el botón de la barra de iconos que contiene una flecha hacia arriba(). Esto dará lugar a un dialogo como el que se muestra en la *figura A.10*.

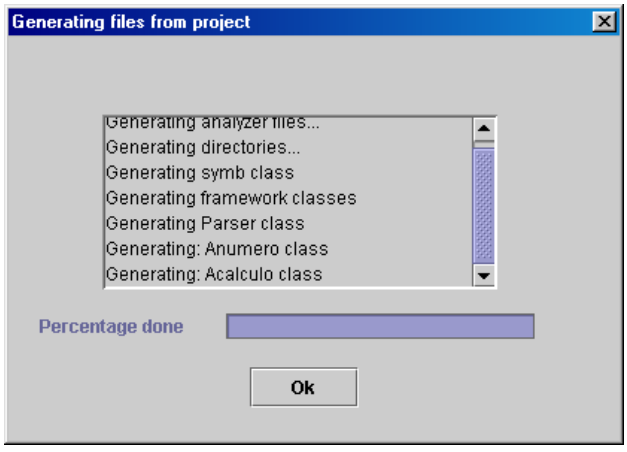


Figura A.10 O2C2ui generando los ficheros del analizador

Si no ha habido problemas, el siguiente paso a realizar seria compilar y ejecutar los ficheros generados.

A.7 La ayuda

Generar procesadores de lenguajes no es una labor sencilla, aunque se incluya un buen entorno de desarrollo.

Figura A.11 Ventana principal de ayuda

Para ayudar al programador durante todo el proceso de construcción, O2C2ui incluye un completo sistema de ayuda bastante intuitivo, en forma de un mininavegador. De hecho, es posible proporcionar al usuario páginas de ayuda que estén situadas remotamente.

La ayuda puede ser activada desde el botón representado por el icono () o desde la entrada del menú *Ayuda->Help Topics*. El resultado es un cuadro de dialogo como el mostrado en la *figura A.11*.

Los botones de la barra de iconos son los clásicos de cualquier navegador. El situado en primer lugar significa navegar hacia páginas ya visitadas, el situado en segundo lugar, regresa a la página principal de la ayuda que contiene el índice y el situado en tercer lugar, permite navegar por páginas hacia adelante. Cualquier usuario que haya utilizado previamente un sistema de ayuda o un navegador no tendrá problemas al utilizar el que se presenta en este entorno.

A.8 Un ejemplo sencillo

Se partirá del siguiente ejemplo sencillo para mostrar los pasos generales que se deben seguir en la construcción de un procesador de lenguaje usando el entorno

visual de desarrollo O2C2ui. Las reglas de la siguiente gramática corresponden a una simple calculadora de números enteros.

```
<S>   ::= <num> OP <num> "operation" ;
<num> ::= DIGIT {DIGIT} "numero" ;
```

El primer paso consiste en inicializar la aplicación. Después de esta operación se mostrará una ventana como la representada en la *figura A.12*.

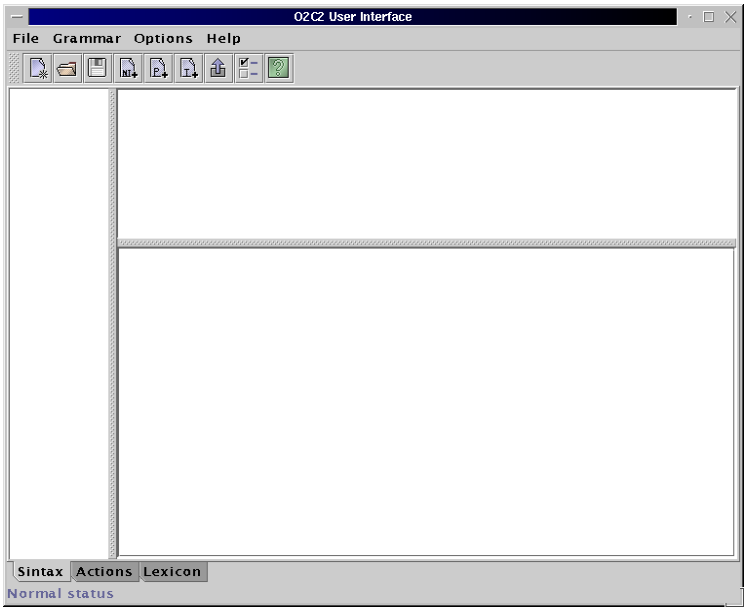


Figura A.12 Ventana principal

A.8.1 Introducción de la gramática

En primer lugar se deben introducir los símbolos *no terminales* de la gramática (*figura A.13*). Es obligatorio introducir el primer símbolo ya que se corresponderá con el símbolo inicial de la gramática, los otros símbolos irán apareciendo a medida que se vayan introduciendo las producciones asociadas a cada elemento no terminal.

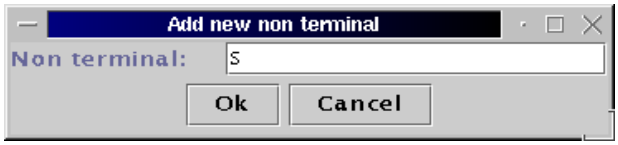


Figura A.13 Inserción de nuevo elemento no terminal

Después de este paso, ya se ha definido el primer símbolo no terminal de la gramática, a continuación se pueden introducir las producciones asociadas a este símbolo o al que se haya seleccionado previamente. Ver *figura A.14*.

Figura A.14 Inserción de una nueva producción

Como resultado de haber introducido esta producción, la aplicación refleja la aparición de un nuevo símbolo no terminal *num*, un ancla *operacion* y el token *OP*.

Para terminar la definición de la gramática, se introduciría la producción correspondiente al nuevo símbolo no terminal *num* siguiendo los pasos anteriores. La aplicación refleja en este caso la aparición del símbolo terminal *DIGITO* y el ancla *numero*. La pantalla principal estaría actualizada en todo momento como se puede observar en la *figura A.15*.

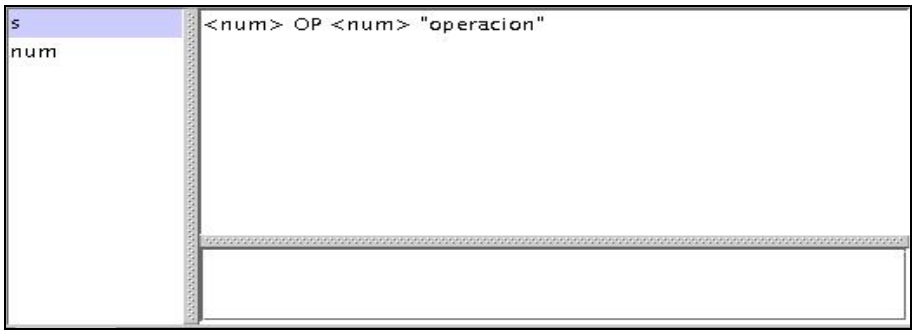


Figura A.15 Actualización de la pantalla principal al introducir las producciones

Es necesario asegurarse de que todos los símbolos *no terminales* que aparezcan en la lista tengan alguna producción, puesto que en caso contrario se estaría definiendo una gramática que podría incumplir los requisitos para ser analizada. No obstante por defecto, los símbolos *no terminales* sin producciones se interpretan como *landa*, con producciones vacías.

El entorno permite corregir los posibles errores cometidos durante la edición de la gramática. Los símbolos *no terminales* se pueden borrar, las producciones se pueden eliminar y modificar, si se seleccionan de la lista, aprovechando la incorporación del editor de textos en a parte inferior derecha como se muestra la *figura A.16*.

Después de completar la definición gramatical, se puede comenzar a realizar el análisis léxico o el análisis semántico. Es importante señalar que durante todo el proceso de desarrollo se permiten hacer cambios en la especificación de la gramática si se observa que son necesarios.

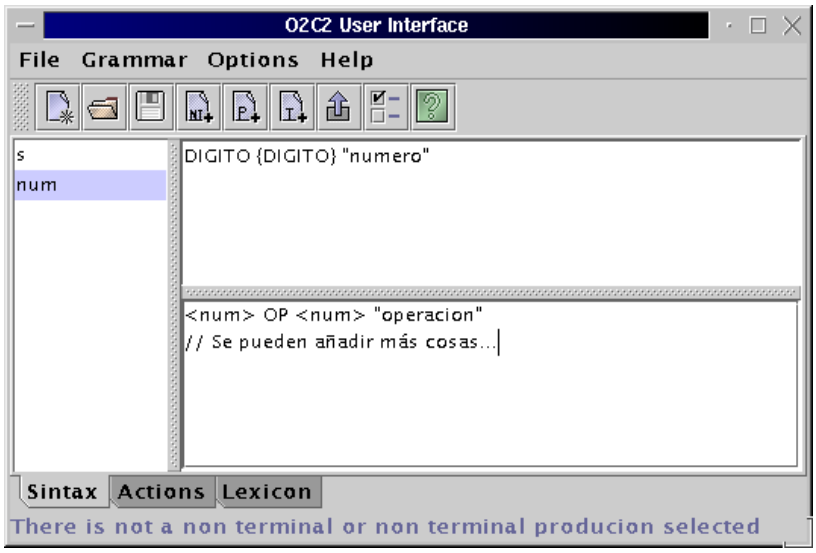


Figura A.16 Modificación de una producción introducida previamente

A.8.2 Introducción del léxico

Todos los símbolos *terminales* definidos en la especificación gramatical quedan reflejados en la pestaña asociada al análisis léxico.

Como se puede observar en la *figura A.17*, los tokens cuyo nombre aparece en la columna izquierda de la imagen corresponden a los símbolos *terminales* introducidos en la gramática. En este caso es obligatorio definirlos puesto que son necesarios para llevar a cabo el proceso de análisis.

Para asociar un valor a un token determinado, simplemente se debe seleccionar el cuadro situado a la derecha del símbolo e introducir el texto correspondiente. En la versión actual del prototipo solamente se permiten describir los tokens mediante expresiones regulares.

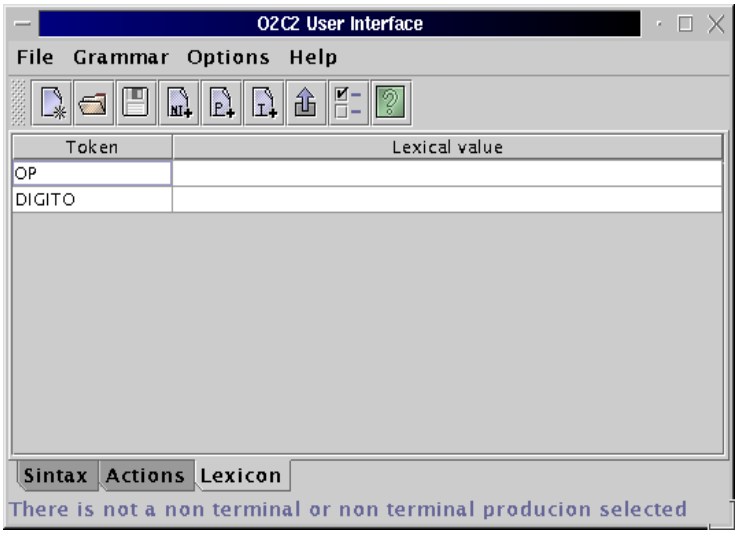


Figura A.17 Ventana con los tokens definidos en la gramática

En la *figura A.18* se muestran los valores de los símbolos *terminales* introducidos. No obstante, es posible definir otros elementos léxicos además de los que aparecen de forma automática. Esos elementos léxicos pueden servir de base para definir a partir de ellos otros símbolos.

Token	Lexical value
OP	[+|-|\\\|*]
DIGITO	[0-9]

Figura A.18 Definición léxica de los tokens

A.8.3 Introducción de las acciones

Otra operación necesaria para completar el proceso de análisis del ejemplo, consiste en introducir las clases *listener* encargadas de efectuar las tareas del análisis semántico.

Para ello se debe seleccionar la pestaña asociada al análisis semántico que está etiquetada con la palabra *actions*. La información que se muestra es la que corresponde a la imagen de la *figura A.19*.

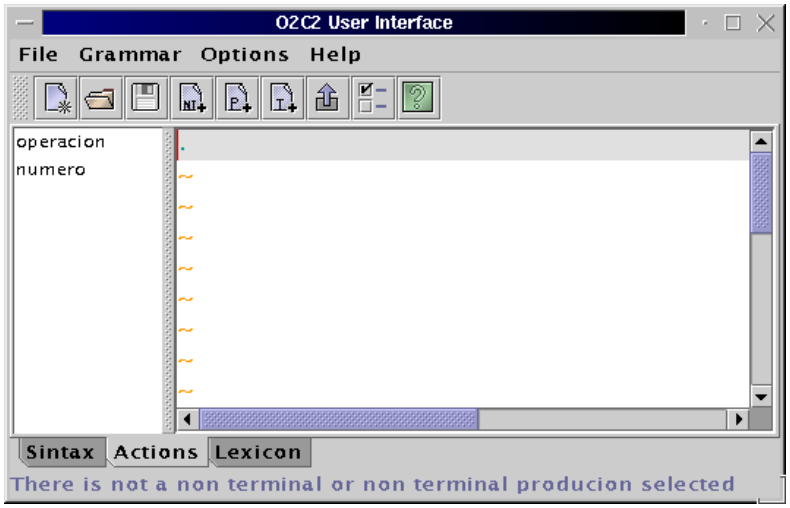


Figura A.19 Ventana principal del análisis semántico

Si en este paso, aún no se han establecido los directorios y paquetes del proyecto, se debe suministrar esta información a la aplicación. La opción de menú *options->Project options* mostrada en la imagen de la *figura A.20* permite realizar esta tarea.

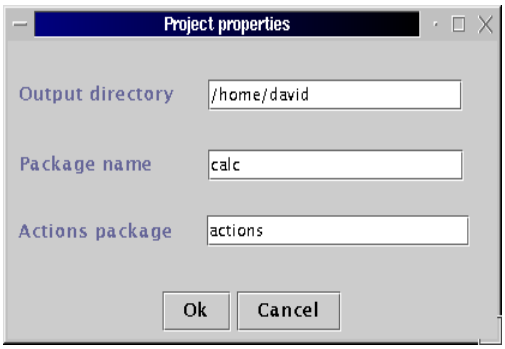


Figura A.20 Propiedades del proyecto

Una vez definidas las opciones del proyecto, el sistema ya conoce el nombre del paquete bajo el cual se van a generar las distintas clases, incluyendo el paquete que va a contener el código semántico proporcionado por el usuario.

Desde la ventana principal del análisis semántico, activando cualquiera de los anclas que se encuentran en una lista situada en la parte izquierda, aparecerá el esqueleto de la clase semántica que se debe completar puesto que aún no se han definido los *listeners* (*figura A.21*).

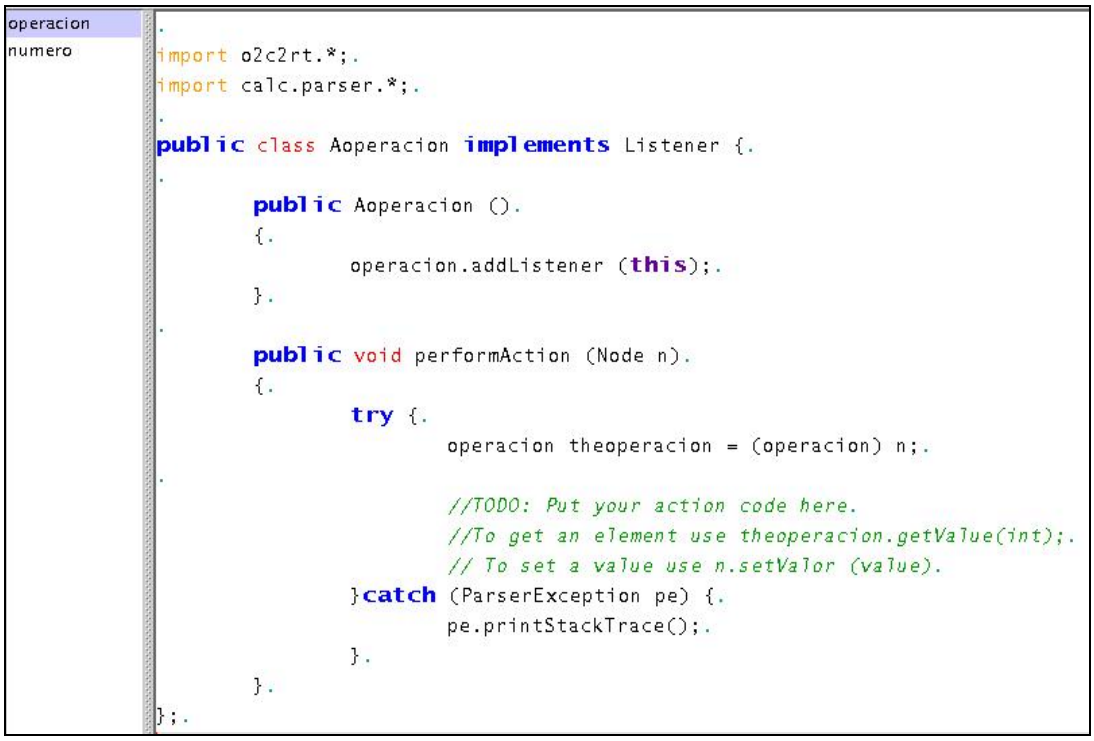


Figura A.21 Esqueleto de listener generado automáticamente por la herramienta

Para el ejemplo concreto que se está considerando en este apartado, se debe completar el método `performAction()` de las clases `Aoperacion` y `Anumero`.

El código del **método** `performAction()` en la **clase** `Aoperacion` es el siguiente.

```
public void performAction (Node n)
{
  int resul= 0;
   int num1, num2;
   String op;

   try {
        operation theoperation = (operation) n;
        num1 = ((Integer)theoperation.getValue(0)).intValue();
        num2 = ((Integer)theoperation.getValue (2)).intValue();
        op = (String) theoperation.getValue(1);
        if (op.compareTo("-")==0) resul=num1-num2;
        else if (op.compareTo("+")==0) resul=num1+num2;
        else if (op.compareTo("*")==0) resul=num1*num2;
        else if (op.compareTo("/")==0) {
         if (num2==0) {
           throw new SemanticException ("Error, división por
                                        cero");
         }
         else resul = num1/num2;
       }
      System.out.println ("\nResultado: "+resul);
   } catch (ParserException pe) {
      pe.printStackTrace();
    }
}
```

El código del **método** `performAction()` en la **clase** `Anumero` es el siguiente:

```
  public void performAction (Node n)
  {
    try {
          number num = (number) n;
          String s1 = (String) num.getValue(0);
          String s2 = "";
           Vector v1 = (Vector)num.getValue(1);
          if (v1!= null) {
             for (int contador=0;contador<v1.size(); contador++){
                String aux = (String) v1.elementAt(contador);
                s2 = s2+aux;
             }
          }
         num.setValue (new Integer(s1+s2));
     } catch (ParserException pe) {
        pe.printStackTrace();
     }
}
```

En este punto es necesario aclarar dos cosas. En primer lugar que el código de este último método no es todo lo simple que da a entender la gramática del ejemplo debido a que se necesitan realizar muchas comparaciones para poder determinar el operador aritmético, y en segundo lugar que todo el código ha sido escrito deliberadamente en ingles (*operación* y *numero* se corresponden a *operation* y *number* en el código).

A.8.4 Generación de código

El último paso consiste en obtener el código del procesador de lenguaje. Activando el botón asociado al proceso de generación de código de la ventana principal, aparecerá una imagen como la mostrada en la *figura A.22* que informará al usuario del estado de la generación de dicho código:

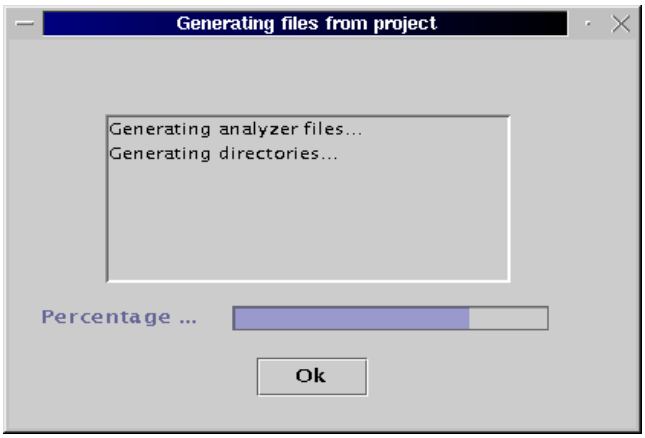


Figura A.23 Dialogo informando del proceso de generación de código

O2C2ui genera ficheros que contienen el código del analizador léxico, el código del analizador sintáctico y el código de las clases semánticas. Para compilar y ejecutar el código generado, se necesita disponer de la plataforma Java.

Anexo B

Conversión de términos Inglés - Español

En la siguiente tabla se incluye una lista de términos en inglés y su traducción al español. Se ha intentado realizar una traducción buscando el termino utilizado en la mayoría de los libros de texto en español. No obstante, algunos términos no se han traducido por no encontrar una traducción adecuada.

Término en Inglés	Traducciones Preferidas	Otras traducciones
ASTs	Árboles Sintácticos Abstractos	
Backtraking	Retroceso	
Bootstraping	*No se ha traducido*	
Back-end	Fase de síntesis	
Botton-up	Ascendente	
CFGs	Gramáticas Libres de Contexto	
Cross-Compiler	Compilador cruzado	
Debug	Depurado	
Framework	Marco de aplicación	Marco de trabajo
Front-end	Fase de análisis	
Interface	Interfaz (*femenino*)	
Lexer	Analizador léxico	
Packages	Paquetes	
Parser	Analizador sintáctico	
Parsing	Analizador sintáctico	
Scanner	Analizador léxico	
Tree-walker	Árbol de recorrido	
Templates	Plantillas	
Token	Componente Léxico	Entidad Léxica
Top-down	Descendente	

BIBLIOGRAFÍA

[Abs01] Abstract Window Toolkit
URL: http://java.sun.com/products/jdk/awt/index.html Julio 2001.

[Acc01] *The Accent Compiler Compiler.*
German National Research Center for Information Technology
URL: http://accent.compilertools.net/ Septiembre 2001.

[Adl96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steve Lucco y Robert Wahbe. *Efficient and Language-Independent Mobile Programs*. En Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation. Mayo de 1996. Pág. 127-136.

[Alv98] Darío Álvarez Gutiérrez. *Persistencia completa para un sistema operativo orientado a objetos usando una máquina abstracta con arquitectura reflectiva*. Tesis Doctoral. Departamento de Informática, Universidad de Oviedo. Enero 1998.

[App98] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press. 1998.

[Bas00] David Basanta Gutiérrez. *Análisis e Implementación de Técnicas Orientadas a Objetos para el Diseño y Construcción de Procesadores de Lenguajes*. Proyecto Fin de Carrera Nº 1002022. Escuela Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón. Universidad de Oviedo, 2000.

[Ber97] Elliot Berk. *Jlex: A lexical analyzer generator for Java™*, 1997. URL: http://www.cs.princeton.edu/~appel/modern/java/JLex/, 2001.

[BGH57] J.W. Backus, R. Goldberg, L.M. Haibt, R.J. Beeber, S. Best, H.L. Herrick, R.A. Nelson, D. Sayre, P.B. Sheridan, H. Stern, I. Ziller, R.A. Highes and R. Nutt. *The FORTRAN automatic coding system*. Western Joint Computer Conference, 1957.

[BLI+00a] D. Basanta Gutiérrez, M.C. Luengo Díez, R. Izquierdo Castanedo, J. E. Labra Gayo, J. M. Cueva Lovelle. *Improving the quality of compiler construction with object-oriented techniques*. ACM SIGPLAN. Volume 35, Nº 12. Pág. 41-51. Diciembre 2000.

[BLI+00b] D. Basanta Gutiérrez, M.C. Luengo Díez, R. Izquierdo Castanedo, J. E. Labra Gayo, J. M. Cueva Lovelle. *Constructing Language Processors using Object-Oriented Techniques*. 6th International Conference on Object Oriented Information Systems (OOIS 2000).
ISBN 1-85233-420-7, Springer-Verlag, 2000.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns.* Wiley and Sons Ltd., 1996. ISBN 0-471-95869-7

[Bol97] William J. Bolosky, Richard P. Draves, Robert P. Fitzgerald, Chistopher W. Fraser B. Jones, Todd B. Knoblock y Rick Rashid. *Operating System Directions for the Next Millenium.* En Proceedings of the 6th Hot Topics in Operating Systems (HotOs-VI). Cape Codd, Massachussets, EE.UU. Mayo de 1997.

[Boo94] Grady Booch. *Object-Oriented Análisis and Design with Aplications, 2nd edition*. Benjamín Cummings. 1994.
Versión en español: *Análisis y Diseño Orientado a Objetos con Aplicaciones, 2ª edición*. Addison-Wesley/Díaz de Santos. 1996.

[BRJ99] Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley. 1999.
Versión en español: *El Lenguaje Unificado de Modelado*. Addison-Wesley. 1999.

[Bos96] J. Bosch. *Delegating Compiler Objects*. Proceeding of Compiler Construction. 1996.

[Bos97] J. Bosch. *Delegating Compiler Objects: Modularity and Reusability in Language Engineering*. Nordic Journal in Computing 4, 66-92, 1997.

[BPS00] Tim Bray, Jean Paoli, C. M. Sperberg. *Extensible Markup Language (XML) 1.0 (Second Edition)*. Widw Web Consortium (W3C). Octubre 2000. URL: http://www.w3.org/TR/2000/REC-xml-20001006.

[Car97] L. Cardelli. *Type systems*. En Handbook of Computer Science and Engineering, capítulo 103. CRC Press, 1997.

[CE00] Krysztof Czarnecki y Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ISBN: 0-201-30977-7. Addison Wesley, 2000.

[Coc93] *The Cocktail Compiler Toolbox*. German National Research Center for Information Technology, 1993.
URL: http://www.first.gmd.de/cocktail/, Junio 2001.

[Cur01] Curtin Computer Science. *The Delegation Event Model*. Curtin University of Technology. WWW-based CS Units. Junio 2001
URL: http://weed.cs.curtin.edu.au/units/cg252-502/notes, Julio 2001.

[Dav00] M. Davis. *The universal computer. The road from Leibniz to Turing.* Ed. W.W. Norton & Company, N.Y., 2000.

[DKV00] Arie van Deursen, Paul Klint, y Joost Visser. *Domain-specific languages: An annotated bibliography*. ACM SIGPLAN Notices, 35(6):26–36, Junio 2000.

[DS95] Charles Donnelly and Richard Stallman. *Bison The YACC-compatible Parser Generator*. Version 1.25. November 1995.
URL: http://dinosaur.compilertools.net/bison/index.html, 2001

[Ear70] J. Earley. *An efficient context-free parsing algorithm*. Communications of the ACM, 13(2):94-102, 1970.

[FG93] A. E. Fischer y F. S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall International, 1993.

[Fow01] Amy Fowler. *A Swing architecture overview. The inside story on JFC Component Design*.
URL:http://java.sun.com/products/jfc/tsc/articles/architecture/index.html. Sun Microsystems. Junio 2001.

[FS97] M. Fayad, D. C. Schmidt. *Object-Oriented Application Frameworks*. Special Issue on Object-Oriented Application Frameworks. Comunications of the ACM, Vol. 40, No. 10, Octubre 1997.

[GH98] E. M. Gagnon, L. J. Hendren. *SableCC, an Object-Oriented Compiler Framework*. Proceeding of the Technology of Object-Oriented Languages and Systems, IEEE 1998.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Software Architecture*. Addison-Wesley, 1995.

[GHL+92] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, W.M. Waite. *Eli: A Complete, Flexible Compiler Construction System*. Communications of the ACM, Vol.35, No.2, pp.121-131. Feb.1992
URL: http://www.cs.colorado.edu/~eliuser/, Julio 2001.

[GJ98] Dick Grune y Ceriel Jacobs. *Parsing Techniques - A Practical Guide*. ISBN 0-13-651431-6. Septiembre 1998.
URL: http://www.cs.vu.nl/~dick/PTAPG.html, Junio 2001.

[GJS96] J. Gosling, B. Joy y G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Goo87] N. C. Goodwin. *Functionality and usability*. Communications of the ACM. Pág. 229-233. Marzo 1987.

[Gou85] J. D. Gould, C. Lewis. *Designing for usability: Key principles and what designers think*. Communications of the ACM. Pág. 300-311. Marzo 1985.

[Guz01] Joel de Guzmán. *Spirit Parser Compiler*. Isis Technologies. Versión 1.1. Julio 2001.
URL: http://spirit.sourceforge.net/, 2001.

[Hol95] Jim Holmes. *Object-Oriented Compiler Construction*. Prentice Hall. 1995.

[Höl95] Urs. Hölze. *Adaptative Optimization for Self: Reconciling High Performance with Exploratory Programming*. Tesis Doctoral, Department of Computer Science, Stanford University, EE.UU. Marzo de 1995.

[HU94] U. Hölze y D. Ungar. *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback*. En Proceedings of the 1994 SIGPLAN conference on Programming Language Design and Implementation. Orlando, Florida, EE.UU. Junio 1994.

[Hud97] Scott E. Hudson. *CUP parser generator for Java™*, 1997.
URL: http://www.cs.princeton.edu/~appel/modern/java/CUP/, 2001.

[Ins01] Magelang Institute. ANTLR 2.x.x, 2001. URL: http://www.antlr.org

[Iso96] ISO/IEC. "ISO-EBNF", ISO/IEC 14977:
URL: http://www.cl.cam.ac.uk/%7Emgk25/iso-14977.pdf, 1996.

[JBR99] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley. 1999.
Versión en español: *El Proceso Unificado de Desarrollo de Software*. Addison-Wesley. 1999.

[Jav01a] Java Foundation Classes
URL: http://java.sun.com/products/jfc/index.html, Julio 2001.

[Jav01b] Java Compiler Compiler™ (JavaCC) - The Java Parser Generator
http://www.webgain.com/products/java_cc/documentation.html,
Septiembre 2001

[JF88] R. E. Johnson, B. Foote. *Designing Reusable Classes*. Journal os Object-Oriented Programming. Vol. 1, No 2, Junio 1988.

[Jfl01] Jflex: The Fast Scanner Generator for Java
URL: http://www.jflex.de Octubre 2001.

[Joh00] S. C. Johnson. *Comunicación personal*.
Diciembre 2000.

[Joh75] S. C. Johnson. *YACC- yet another compiler compiler*. Technical Report Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[Kam90] S. Kamin. *Programming languages: an interpreter based approach*. Addison-Wesley, 1990.

[Kam98] S. Kamin. *Research on domain-specific embedded languages and program generators*. Electronic Notes in Theoretical Computer Science, Elsevier Press, 12, 1998.

[KJS96] Douglas Kramer, Bill Joy y David Spenhoff. *The Java™ Platform White Paper*. JavaSoft. Mayo de 1996.
URL: ftp://ftp.javasoft.com/docs/papers/JavaPlatform.pdf, 2001

[Knu90] Donald E. Knuth. *The genesis of attribute grammars*. En WAGA conference proceedings, Paris 1990 , páginas 1–12. Springer-Verlag, 1990. Lecture Notes in Computer Science 461.

[Knu65] D. Knuth, *On the translation of languages from left to righ*", Information and Control, Vol. 8, Num. 6, 1965, pp. 607 - 639.

[Kuh93] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 3rd edition, 1993.

[Lab01] J. Emilio Labra Gayo. *Tesis Doctoral – Desarrollo Modular de Procesadores de Lenguajes a partir de Especificaciones Semánticas Reutilizables*. Julio 2001, Departamento de Informática. Universidad de Oviedo.

[Lem01] *The Lemon Parser Generator*
URL: http://www.hwaci.com/sw/lemon/lemon.html, Junio 2001.

[Les75] M. E. Lesk. *Lex – a lexical analyzer generator*. Technical Report Computing Science Technical Report 39, AT&T Bell Laboratoires, Murria Hill, N.J.,1975.

[LCL+01a] J. E. Labra Gayo, J. M. Cueva Lovelle, M. C. Luengo Díez, M. González. *A Language Prototy-ping Tool based on Semantic Building Blocks*, Formal Methods and Tools for Computer Science, EUROCAST-2001, Workshop on Functional Programming and λ-Calculus. Lecture Notes in Computer Science, Springer-Verlag, 2001.

[LCL+01b] J. E. Labra Gayo, J. M. Cueva Lovelle, M. C. Luengo Díez, A. Cernuda del Río. *Modular Development of Interpreters from Semantic Building Blocks*. Nordic Journal of Computing. Vol. 8, Nº 3, pág. 391-407, 2001.

[LCL+01c] J. E. Labra Gayo, J. M. Cueva Lovelle, M. C. Luengo Díez, A. Cernuda del Río. *LPS: A Language Pro-totyping System Using Modular Monadic Semantics.* Workshop on Language. Descriptions, Tools and Applications Génova – Italia, 2001, Satellite events for ETAPS 2001, Electronic Notes in Theoretical Computer Science, vol. 44, Elsevier Science.

[LCL+01d] J. E. Labra Gayo, J. M. Cueva Lovelle, M. C. Luengo Díez, A. Cernuda del Río. *Specification of Logic Programming Languages from Reusable Semantic Building Blocks*. International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001). Kiel – Germany, 2001, Electronic Notes in Theoretical Computer Science, vol. 64, Elsevier Science.

[LCL+01e] M. C. Luengo, J. M. Cueva, J. E. Labra, N. García, D. Basanta. *Applying Frameworks and Object-Oriented Techniques for developing Language Processors Tools*. Primeras Jornadas sobre Programación y Lenguajes. Almagro – Ciudad Real, Noviembre 2001.
PROLE 2001: I Jornadas sobre programación y lenguajes. Editores: Fernando Orejas, Fernando Cuartero, y Diego Cazorla. Universidad de Castilla-La Mancha, 2001, Pág. 77-91. Depósito Legal AB-475-2001.

[LLC+01] M. C. Luengo, J. E. Labra, J. M. Cueva, N. García, D. Basanta. *Building Compiler Tools using Frameworks*. SISOFT 2001: Simposio Iberoamericano de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento. Bogota – Colombia, Agosto de 2001. ISBN: 95897030-3-8.

[LLD+98] M. C. Luengo Díez, J. E. Labra Gayo, F. Domínguez Mateos, A. Pérez Díaz, N. García Fernández y J. M. Cueva Lovelle. *Desarrollo de Compiladores en un Sistema Integral Orientado a Objetos*. Memoria del V Congreso Internacional de Investigación en Ciencias Computacionales CIICC´98, pp. 291-301, Aguascalientes (México), Noviembre de 1998. ISBN: 970-13-2261-4.

[LY97] T. LinDholm y F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[May87] C. May. *MIMIC: A Fast System/370 Simulator*. En Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, en SIGPLAN Notices, V. 22, N. 7. St. Paul, Minneapolis, EE.UU. Julio de 1987. Pág 1-13.

[McC63] John McCarthy. *Towards a mathematical science of computat*ion. En IFIP Congress 1962 . North Holland, 1963.

[McL87] Bruce J. McLennan. *Principles of Programming Languages*. Oxford Uni-versity Press, 2 edición, 1987.

[Mey99] Bertrand Meyer. *Construcción de Software Orientado a Objetos, 2ª edición*. Prentice Hall. 1999.

[Mic01] Microsoft Foundation Classes
URL:http://msdn.microsoft.com/visualc, Julio de 2001.

[Mos92] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[MPV99] O. De Moor, S. Peyton-Junes, E. Van Wyk. *Aspect Oriented Compilers*. First International Symposium on Generative and Component-Based Software Engineering. LNCS, Springer-Verlag, 1999.

[Nau60] P. Naur et al. *Revised report on the algorithmic language algol 60*. Communications of the ACM, 3(5):299 – 314, 1960.

[Par93] Terence J. Parr. *Obtaining Practical Variants of LL(k) and LR(k) for k>1 by Splitting the Atomic k-Tuple*. PhD thesis, Purdue University, West Lafayette, Indiana, Agosto 1993.

[Par97] Terence J. Parr. *Language Translation Using PCCTS & C++*. Autómata Publishing Company, 1997. ISBN: 0962748854
URL: http://www.antlr.org/papers/pcctsbk.pdf, 2001

[Pax95] Vern Paxson. *Flex - A fast scanner generator*. Versión 2.5. Marzo 1995. URL: http://dinosaur.compilertools.net/flex/index.html, 2001

[Pcl00] *PCLEX*. URL: http://www.abxsoft.com/pdf/pclex.pdf, Abraxas Sofware, Inc. 1986-2000.

[Pcy00] *PCYACC. Profesional Language Development Toolkit*. Version 9.0 URL http://www.abxsoft.com/pdf/pcyacc.pdf, Abraxas Sofware, Inc. Enero 2000.

[PDC91] T. J. Parr, H. G. Dietz y W. E. Cohen. *PCCTS Reference Manual* (Version 1.00). School of Electrical Engineering, Purdue University, West Lafayette, Agosto 1991.

[Ple97] M. Plezbert. Does just-in-time equal better-late-than-never? En 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Paris, France, 1997.

[PQ95] Terence J. Parr and Russell W. Quong. *ANTLR: A Predicated-LL(k) Parser Generator*. Journal of Software Practice and Experience, 1995.

[PQ96] Terence J. Parr and Russell W. Quong. LL and LR Translator Need k. SIGPLAN Notices Volumen 31, Nº 2. Febrero 1996.

[PZ96] T. W. Pratt y M. V. Zelkowitz. *Programming languages: Design and implementation*. Prentice-Hall Internationa, 1996.

[Pro01] *ProGrammar ™ Parser Development Toolkit*
URL: http://www.programmar.com/main.shtml, NorKen Technologies, Inc. Noviembre 2001.

[RJB99] James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual. Addison-Wesley. 1999.
Versión en español: El Lenguaje Unificado de Modelado Manual de referencia. Addison-Wesley.2000.

[RZ96] Dirk Riehle, Heinz Züllighoven. *Understanding and Using Patterns in Software Development.* Theory and Practice of Object Systems.
URL: http://citeseer.nj.nec.com/riehle96understanding.html

[Ste60] T.B. Steel Jr. "*UNCOL: Universal Computer Oriented Language Revisited*". Datamation. Enero/Febrero de 1960. Pág. 18.

[Ste98] Jr. Guy Lewis Steele. *Growing a Language*, octubre 1998. Transcript of invited talk at OOPSLA'98, Vancouver.

[Sun01] Sun Microsystems Usability Labs
URL: http://www.sun.com/usability, Noviembre 2001.

[Ter00] P. D. Terry, *Compilers and Compiler Generators - an introduction with C++*, published by International Thomson Computer Press, 1997. On-line edition, 2000.

[Tro97] Eric Trout. *Building the Virtual PC*. Byte. Noviembre de 1997. Pág. 51-52.

[Ulo01] Vladimir I. Ulogov. *Language list*. http://oop.rosweb.ru/, 2001.

[Vel95] T. Veldhuizen, *Using C++ template metaprograms*, C++ Report Vol. 7 No. 4 (May 1995), pp. 36-43.
URL: http://www.extreme.indiana.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html, 2001.

[Wat96] David A. Watt. *Why don't programming language designers use formal methods?* En Seminario Integrado de Software e Hardware - SEMISH'96, páginas 1–6. University of Pernambuco, Recife, Brazil, 1996.

[Way96] Peter Wayner. *Sun Gambles on Java Chips*. Byte. Noviembre de 1996. Pág. 79-88.

[Wil97] Friedrich Wilhelm Schröer. *The GENTLE Compiler Construction System*. ISBN: 3-486-247034-4. 1997.
URL: http://www.first.gmd.de/gentle/, Junio 2001.

[You67] D. H. Younger, *Recognition and Parsing of Context-Free Languages in Time n 3*, Information and Control, Vol. 10, Nº 2, Febrero 1967, Pág. 189-208.