Prácticas de la asignatura Procesadores de Lenguaje con la herramienta ANTLR

José A. Troyano, Francisco J. Galán, Vicente Carrillo,
Fernando Enríquez, Enrique J. García
Depto. de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
41012 Sevilla
e-mail: troyano@lsi.us.es

Resumen

En este artículo presentamos una experiencia docente en las prácticas de laboratorio de la asignatura troncal Procesadores de Lenguaje. Es muy común en este tipo de asignaturas el uso de herramientas para generar analizadores léxicos y sintácticos a partir de expresiones regulares y gramáticas, respectivamente; pero no está tan extendido el uso de herramientas para dar cobertura a las últimas etapas de un procesador de lenguaje, las que se encargan de procesar los aspectos semánticos del mismo. Nosotros hemos utilizado ANTLR, una herramienta que da soporte a todas las etapas del procesamiento de lenguajes formales, permitiendo generar analizadores léxicos, sintácticos, árboles de sintaxis abstracta y analizadores de árboles para el procesamiento semántico.

1. Introducción

Tradicionalmente las herramientas utilizadas en las asignaturas dedicadas a los procesadores de lenguaje (los clásicos compiladores) no cubren todos los contenidos presentados en la teoría. Mientras que para las primeras etapas (*front-end*) se suelen utilizar herramientas basadas en gramáticas y expresiones regulares, para la última parte (*back-end*) es difícil encontrar herramientas que den soporte a los problemas que allí se plantean.

Las etapas del análisis léxico y sintáctico de los lenguajes suelen estar resueltas por herramientas descendientes de la pareja clásica *lex* y *yacc*, como las implementaciones en C/C++ *flex* y *bison*, o más recientemente, las correspondientes herramientas en Java *jflex* y *cup*. Tanto el problema léxico como el sintáctico están perfectamente estudiados y hay soluciones satisfactorias para ellos desde hace muchos años, lo que hace que existan herramientas muy bien acabadas y de uso muy extendido. Una vez que el alumno comprende los conceptos de gramática y expresión regular, y ha aplicado las herramientas a unos pocos ejemplos, estos problemas empiezan a dejar de tener secretos y el diseño de sus soluciones se puede considerar casi metódico. Esto plantea una situación ideal desde el punto de vista docente, con problemas bien fundamentados teóricamente, herramientas consolidadas, y una metodología para construir la solución.

Sin embargo, la situación no es tan ideal para las etapas dedicadas al procesamiento semántico de los lenguajes. Por un lado, los problemas a resolver no son tan parecidos cuando pasamos de un lenguaje a otro. Por ejemplo, no es muy distinto escribir las gramáticas de un lenguaje imperativo y de un lenguaje declarativo, pero implementar sendos intérpretes plantea problemas muy diferentes. La otra diferencia está en las herramientas, en el caso del tratamiento semántico no existe ningún referente ampliamente usado y, salvo el limitado apoyo que supone el uso de gramáticas con atributos durante el reconocimiento sintáctico, la situación está muy cercana al "búsquese usted la vida".

Esta diferencia queda incluso patente en la bibliografía clásica de la materia, textos como [1] ó [8] dejan completamente cerrado el problema sintáctico con especificaciones basadas en gramáticas, mientras que en los capítulos dedicados al tratamiento semántico exponen los conceptos y la hora de presentar la solución lo hacen directamente apoyándose en la implementación con un pseudocódigo o lenguaje de programación. El caso de [2] se sale un poco de la norma al presentar de una manera más profunda

conceptos como gramáticas abstractas y gramáticas con atributos evaluadas sobre árboles, pero no lo complementa con una herramienta que dé apoyo a toda la metodología.

Nuestra propuesta académica para esta situación pasa por el uso de la herramienta ANTLR [4] en todas las etapas del procesamiento de lenguajes. Esta herramienta, que viene desarrollándose desde principios de los 90 con su antecesor PCCTS, ha alcanzado un grado de madurez tal que la hace una alternativa competitiva a las herramientas clásicas para la etapa sintáctica, con el valor añadido de disponer de una notación basada en gramáticas de árboles que permite especificar procesos como el análisis semántico o la generación de código de una manera abstracta y compacta.

En el resto de este trabajo presentaremos nuestra propuesta de diseño de prácticas que va acompañada del material docente correspondiente, publicado en [3] y que ha sido utilizado con muy buenos resultados en la asignatura *Procesadores de Lenguaje I* (en adelante PL1) durante el curso 2004-05.

2. Marco académico

Esta sección está dedicada a presentar el contexto académico en el que se encuadra la asignatura PL1 para la que se han diseñado las prácticas de laboratorio que constituyen el núcleo de este trabajo. En primer lugar nos dedicaremos a resumir brevemente las asignaturas previas y posteriores a PL1, para luego introducir el programa de la asignatura.

2.1. Asignaturas previas y posteriores

En nuestro plan de estudios de Ingeniería Informática disponemos de cuatro asignaturas cuatrimestrales dedicadas al procesamiento de lenguajes: Lenguajes Formales y Autómatas (LFA), Ampliación de Lenguajes Formales y Autómatas (ALFA) y Procesadores de Lenguaje I y II (PL1 y PL2). Las asignaturas LFA y ALFA están relacionadas con la materia troncal Teoría de Autómatas y Lenguajes Formales y en ellas se introducen los fundamentos teóricos de los lenguajes tanto desde el punto de vista generativo (gramáticas y expresiones regulares) como del reconocimiento (autómatas). Se imparten, respectivamente, en segundo y tercer curso.

La asignatura PL2 es la continuación natural de PL1. Tras haber presentado en PL1 los conceptos, las técnicas y las herramientas utilizadas en el desarrollo de procesadores de lenguaje, en PL2 se aplican a la implementación de compiladores. Se tratan las etapas de comprobación de semántica estática, optimización de código y generación de código para la máquina virtual de Java.

2.2. Programa de la asignatura PL1

Dentro de este esquema, la asignatura PL1 tiene como objetivo principal establecer un puente entre los conceptos teóricos y las aplicaciones sencillas desarrolladas en las asignaturas LFA y ALFA por un lado, y el enfoque más ingenieril y aplicado de PL2. En este sentido, los dos objetivos principales de la asignatura son 1) explicar cómo se implementan las herramientas que generan analizadores sintácticos y 2) presentar las herramientas que permiten abordar la implementación de un procesador de lenguaje.

Los temas en los que se organiza la asignatura son los siguientes:

1. Análisis léxico
2. Análisis sintáctico descendente
3. Análisis sintáctico ascendente
4. Recuperación de errores en analizadores descendentes
5. Gramáticas con atributos
6. Sintaxis abstracta y árboles de sintaxis abstracta
7. Gramáticas para árboles

Los cuatro primeros temas se dedican al análisis léxico y sintáctico, haciendo especial hincapié en los analizadores descendentes porque es el modelo que implementa ANTLR. El tema cinco presenta una primera aproximación a la semántica a través de las gramáticas con atributos evaluadas al vuelo (junto al análisis sintáctico). Por último los temas 6 y 7 presentan los árboles de sintaxis abstracta, su relación con las gramáticas abstractas y la especificación de recorridos a través de gramáticas para árboles.

3. Características básicas de ANTLR

En esta sección resumiremos los elementos más destacables de la herramienta ANTLR. Será una exposición muy breve, se pueden encontrar descripciones más completas en el manual de la herramienta [6] así como en los enunciados de las prácticas disponibles en la página de la asignatura. Casi todos los ejemplos utilizados para presentar la herramienta se corresponderán con el conocido lenguaje de la calculadora aritmética, que admitirá entradas como la siguiente:

```
(1+2)*3;
2+5*4;
6/5/2;
```

A pesar de ser un lenguaje extremadamente simple, sirve perfectamente para ilustrar las capacidades descriptivas de ANTLR. Se pueden encontrar soluciones a problemas más complicados en el material distribuido a los alumnos [3]. En estos ejemplos se ve que, siguiendo unas mínimas pautas de disciplina a la hora de escribir las especificaciones, se pueden conseguir descripciones muy abstractas y compactas incluso para problemas complejos.

3.1. Analizadores léxicos

Seguramente la especificación de analizadores léxicos sea el punto menos fuerte de la herramienta ANTLR. No porque no resuelva bien este problema, sino por la decisión de utilizar el esquema recursivo descendente, en lugar de autómatas finitos, para su implementación. Esta decisión de los autores de la herramienta se debe a que se plantean como una virtud la homogeneidad de modelos de reconocimiento entre distintos analizadores (léxico, sintáctico e incluso de árboles), aplicando en todos los casos el modelo recursivo descendente. En la práctica esto supone algunas molestias, ya que hay que resolver algunos casos de ambigüedad a nivel léxico que no habrían aparecido con autómatas finitos. En el siguiente fragmento se muestra la especificación léxica para el lenguaje ejemplo:

```
class Analex extends Lexer;
  BLANCO: (' '|'\t'|"\r\n")
    {$setType(Token.SKIP);};
  protected DIGITO : ('0'..'9');
  NUMERO: (DIGITO)+
          ('.'(DIGITO)+)?;
  OPERADOR: '+'|'-'|'/'|'*';
  PARENTESIS: '('|')';
  SEPARADOR: ';';
```

Cada regla se corresponde con un *token*, salvo que estén protegidas o que explícitamente se indique que dicho *token* debe ignorarse y no transmitirse al analizador sintáctico, lo que se hace produciendo el *token* `SKIP`. Las reglas protegidas sirven de reglas auxiliares y no compiten con las demás en el intento de construir *tokens* a partir de los caracteres de entrada. Por lo demás las reglas son bastante claras, pudiéndose utilizar los operadores clásicos de las expresiones regulares (`+`,`?`,`*` y `|`).

3.2. Analizadores sintácticos

Desde el punto de vista de la notación, la aportación más interesante de ANTLR en la descripción de gramáticas es el uso de la notación EBNF. Gracias a esta notación, las partes derechas de las reglas pueden incluir, además de símbolos, los operadores clásicos de las expresiones regulares lo que la hace mucho más expresiva. He aquí la especificación sintáctica del lenguaje ejemplo:

```
class Anasint extends Parser;
  instrucciones : (exp ";")* ;
  exp : exp_mult
        (("+"|"-") exp_mult)*
      ;
  exp_mult : exp_base
             (("*"|"/") exp_base)*
           ;
  exp_base : NUMERO
           | "(" exp ")"
           ;
```

Con toda seguridad, la parte menos clara de la especificación anterior es la dedicada a definir el símbolo `exp`. Se ha hecho de esta forma porque, al generar ANTLR analizadores descendentes, no permite describir gramáticas recursivas por la izquierda (esta limitación es parte de la condición LL(1)), que son imprescindibles para especificar la gramática de las expresiones de una forma más declarativa. Precisamente por ser un problema sintáctico complejo, el de las expresiones ha sido muy bien estudiado y existen soluciones asentadas

para distintos tipos de reconocedores (ascendentes, descendentes y de precedencia). Por decirlo de alguna forma, la solución es más compleja que en el resto de los elementos que suelen aparecer en los lenguajes pero no la tenemos que "diseñar", la podemos "reutilizar".

A partir de la gramática, ANTLR implementa un reconocedor recursivo en el que para cada símbolo no terminal existe un método encargado de reconocer el lenguaje asociado a dicho símbolo.

3.3. Soluciones al problema LL

Desde el punto de vista teórico, de los dos modelos básicos de reconocimiento sintáctico, el descendente y el ascendente, es éste último el más potente. El modelo descendente genera analizadores más naturales, pero es más exigente con respecto a las gramáticas que puede procesar. Deben satisfacer la condición LL(1), lo que en ocasiones obliga a transformar las gramáticas. ANTLR propone dos soluciones a este problema: la implementación de analizadores LL(k) y los predicados sintácticos. La primera no es una solución nueva y está ampliamente estudiada en la bibliografía [2] y para una presentación más formal [7]. La única aportación de la herramienta en este sentido es adaptarse a cada caso y no generar consultas de alcance *k* salvo si es estrictamente necesario. La solución de los predicados sintácticos sí es bastante original y permite resolver de forma puntual conflictos de la gramática aunque no puedan resolverse con gramáticas LL(k), siendo *k* fijo. He aquí un ejemplo de predicado sintáctico:

```
instr : (ID ":=") => asigna
      | llamada ;
 asigna : ID ":=" exp ";" ;
 llamada : ID "(" exp ")" ";" ;
```

La gramática anterior describe dos instrucciones de un determinado lenguaje, la asignación y la llamada a un procedimiento. Ambas comienzan con el símbolo `ID`, por lo que la gramática no es LL(1), sin embargo ANTLR no tendría ningún problema en procesar esta gramática gracias al predicado sintáctico `(ID ":=")=>` que aparece antes del símbolo `asigna`. Con él se indica que es necesario comprobar la aparición de los símbolos `ID` y `":="` antes de decidir que la instrucción que se está reconociendo es una asignación. Los predicados sintácticos pueden incluir cualquier expresión EBNF, de manera que si incluyesen una expresión con los operadores `*` ó `+` estarían estableciendo una condición que requeriría de un número variable de símbolos para su validación, superando así en poder descriptivo a las gramáticas LL(k).

3.4. Gramáticas con atributos

La implementación de los analizadores a través del modelo recursivo hace muy fácil la integración de los atributos en la gramática. Dado que cada símbolo tiene asociado un método que lo implementa, los atributos heredados se modelan como parámetros de entrada del método y los sintetizados se resuelven a través del valor de retorno asociado al método. Esta idea, junto con una sintaxis apropiada para declarar los atributos en el seno de una regla es lo único que necesita ANTLR para implementar las gramáticas con atributos. En el siguiente ejemplo se muestra cómo evaluar expresiones aritméticas haciendo uso del atributo sintetizado `[int res]` de los símbolos `exp`, `exp_mult` y `exp_base`:

```
class Anasint extends Parser;
instrucciones {int e;}
   : (e=exp ";"
       {System.out.println(e);})*
   ;
exp returns [int res=0] {int e;}
   : res=exp_mult
   (("+" e=exp_mult {res=res+e;})
   |("-" e=exp_mult {res=res-e;})
   )*
   ;
exp_mult returns [int res=0]
        {int e;}
   : res=exp_base
   (("*" e=exp_base {res=res*e;})
   |("/" e=exp_base {res=res/e;})
   )*
   ;
exception
   catch [ArithmeticException ae]
     {res = 0;}

exp_base returns [int res=0]
        {int e;}
```

```
    : n:NUMERO
    {res= Integer.
          parseInt(n.getText());}
    | "(" e=exp ")" {res = e;}
    ;
```

Hay dos tipos de informaciones que se pueden recuperar de los símbolos, un objeto de la clase `Token` para el caso de los símbolos terminales a través de una etiqueta (como `n:NUMERO`) o un atributo sintetizado por un símbolo no terminal (como `e=exp`).

Por su parte, la notación usada para los atributos heredados es muy similar a la que habitualmente se usa para declarar parámetros de métodos, cambiando los paréntesis por corchetes. Por ejemplo, si tuviésemos expresiones con variables que heredasen un contexto del que extraer los valores de dichas variables lo podríamos describir así:

```
exp_let returns [int res]
        {Hashtable ctx;}
  : "(" LET ctx=vars
            IN res=exp[ctx] ")"
  ;
exp [Hashtable ctx]
     returns [int res] {int e;}
  : res=exp_mult[ctx]
    ("+" e=exp_mult[ctx]
     {res=res+e;})*
  ;
```

El símbolo `vars` sintetiza el atributo `ctx` que a su vez es heredado por el símbolo `exp`, que lo utiliza para calcular el valor de la expresión.

3.5. Árboles de sintaxis abstracta

Los árboles de sintaxis abstracta son un recurso muy útil en el procesamiento de lenguajes ya que proporcionan una flexibilidad de cálculo que no puede ser alcanzada con gramáticas con atributos evaluadas junto al análisis sintáctico (como las de *bison*). Cuando nos enfrentamos al procesamiento de lenguajes complejos, los árboles pasan de ser útiles a imprescindibles, ya que proporcionan una representación intermedia del lenguaje que es compartida por todas las etapas del procesador y que ayuda a la implementación modular del mismo. En [2] se puede encontrar una exposición más extensa de las ventajas del uso de los árboles de sintaxis abstracta.

ANTLR proporciona una manera bastante compacta de anotar gramáticas para construir árboles de sintaxis abstracta. Para empezar, construye automáticamente nodos para todos los símbolos procesados durante el análisis sintáctico. Posteriormente estos nodos son descartados (`!`) o elegidos como raíces (`^`) simplemente anotándolos con el operador correspondiente. Así para nuestro ejemplo la gramática quedaría así:

```
class Anasint extends Parser;
options {buildAST = true;}
tokens {INSTRS;}
instrucciones : (exp ";"!)*
   {##=#(#[INSTRS,"INSTRS"],##);}
   ;
exp : exp_mult
      (("+"^|"-"^) exp_mult)*
   ;
exp_mult : exp_base
      (("*"^|"/"^) exp_base)*
   ;
exp_base : NUMERO
         | "("! exp ")"!
   ;
```

En los casos en los que el árbol propuesto de forma automática no nos sirva, como en el caso de la lista de expresiones `(exp ";"!)*` en la que no hay ningún símbolo que pueda servir de raíz de la lista, se puede indicar explícitamente cómo se desea que se construya el árbol. Para ello se pueden utilizar, entre otras, las operaciones de construcción de nodos `#[...]`, de construcción de árboles `#(...)` y de acceso al árbol construido de forma automática con el atributo predefinido `##`.

En cualquier caso, lo que es innegable es que la especificación es extremadamente simple y expresiva. Así, en nuestro ejemplo, con sólo una instrucción y la inserción de siete operadores, se consigue describir lo que de otra manera nos habría costado decenas de líneas de código.

3.6. Gramáticas para árboles

Una vez que el árbol de sintaxis abstracta ha sido construido, las últimas etapas del procesador de lenguaje consisten básicamente en recorridos de dicho árbol para calcular atributos, transformar el

árbol o generar el resultado final. Todos estos recorridos pueden ser expresados mediante gramáticas para árboles. Estas gramáticas son similares a las gramáticas independientes del contexto salvo que en la parte derecha de las reglas en lugar de secuencias de símbolos aparecen patrones de árboles. La siguiente especificación describe la manera en la que se recorren los árboles del lenguaje ejemplo para calcular los valores de las expresiones:

```
class Evaluador
      extends TreeParser;
instrucciones {int e;}
  : #(INSTRS (e=exp
    {System.out.println(e);})*)
  ;
exp returns [int res=0]
            {int e1=0, e2=0;}
  : n:NUMERO
    {res= Integer.
          parseInt(n.getText());}
  | #("+" e1=exp e2=exp)
      {res = e1 + e2;}
  | #("-" e1=exp e2=exp)
      {res = e1 - e2;}
  | #("*" e1=exp e2=exp)
      {res = e1 * e2;}
  | #("/" e1=exp e2=exp)
      {res = e1 / e2;}
  ;
exception
catch [ArithmeticException ae]
  {res = 0;}
```

Los patrones son expresiones de la forma *#(<raíz> <hijo$_1$>...<hijo$_n$>)* que describen árboles si todos los hijos son símbolos terminales, o familias de árboles si alguno de los hijos es un no terminal. Además, los patrones pueden estar anidados, por lo que las consultas sobre la estructura de los árboles se pueden realizar a distintos niveles. El resto de los elementos de las gramáticas para árboles tienen el mismo significado que en las gramáticas para texto, pudiéndose definir tanto atributos heredados como sintetizados, así como incluir acciones en cualquier punto de la parte derecha de la regla.

4. Diseño de las prácticas

En esta sección presentaremos la manera en la que hemos diseñado las prácticas de la asignatura Procesadores de Lenguaje 1. Dado que el lenguaje con el que están más familiarizados nuestros alumnos cuando llegan a la asignatura es Java, éste ha sido elegido de entre las distintas alternativas de generación que proporciona ANTLR. El conjunto de herramientas necesarias para el desarrollo de las prácticas se completa con Eclipse [5], un entorno de desarrollo configurable para el que ANTLR tiene definido un *plugin* que permite, entre otras cosas, colorear los ficheros de gramática y compilarlos automáticamente. Tanto Eclipse como ANTLR son de libre distribución y están implementados en Java, por lo que los alumnos pueden instalarlos en casa con el sistema operativo que prefieran.

4.1. Metodología y planificación

Hay siete prácticas para todo el cuatrimestre, las dos primeras dedicadas a introducir las herramientas ANTLR y Eclipse, y las cinco siguientes dedicadas a profundizar en las características principales de ANTLR. Los enunciados de las prácticas son documentos de seis o siete páginas en los que se describen los aspectos de las herramientas con los que se va a trabajar durante la sesión de práctica. Aunque no pretenden ser un sustitutivo de los correspondientes manuales, la idea es que los alumnos los utilicen como referente a la hora de aprender el manejo de las herramientas y que recurran a los manuales para buscar cuestiones más específicas y teniendo una idea bastante clara de lo que quieren encontrar.

En cada práctica se plantearán varios ejercicios en los que los alumnos tendrán que utilizar los elementos presentados en el enunciado. La dificultad de los ejercicios será variable, el primero será bastante simple y se proporcionará su solución. De hecho, dicha solución habrá sido presentada a lo largo del enunciado para ilustrar con ejemplos la explicación. De esta forma la primera tarea encomendada a los alumnos será compilar y ejecutar los fuentes presentes en el enunciado y comprobar su correcto funcionamiento. A partir de este primer ejercicio los siguientes irán aumentando en complejidad.

El material se completa con la solución de uno de estos ejercicios complejos, que tiene como objetivo mostrar a los alumnos cierto "estilo de especificación" que asegure descripciones legibles incluso para problemas complejos. Las prácticas son:

1. Instalación y prueba de Eclipse
2. Instalación y prueba de ANTLR
3. Análisis léxico
4. Análisis sintáctico
5. Gramáticas con atributos evaluadas al vuelo
6. Construcción de árboles de sintaxis abstracta
7. Recorridos de árboles mediante gramáticas para árboles

Los enunciados, junto con las soluciones a los ejercicios seleccionados, los enlaces a las páginas de las herramientas y demás material de utilidad están publicados en la página de la asignatura [3].

4.2. Resultados de la experiencia docente

La asignatura PL1 es cuatrimestral y se imparte en el primer cuatrimestre. El curso 2004-05 ha sido el primero en el que hemos utilizado ANTLR en las prácticas de PL1 y podemos catalogar como muy satisfactoria esta primera experiencia. Hasta ahora utilizábamos *flex* y *bison* para generar analizadores léxicos y sintácticos, y un par de herramientas desarrolladas en nuestro departamento para la construcción de árboles de sintaxis abstracta y la especificación de recorridos. Todas ellas generaban programas en C y éste era por tanto el lenguaje elegido para desarrollar los procesadores de lenguajes.

La mayor ventaja del uso de ANTLR es sin duda la homogeneidad. No es lo mismo trabajar con distintas herramientas que han sido implementadas por separado, y que por tanto plantean muchos problemas de coordinación entre ellas, que trabajar con una herramienta integral en la que las interfaces entre módulos están totalmente resueltas. La otra gran ventaja es la cobertura, ANTLR cubre todas las etapas de un procesador de lenguaje, llenando de forma natural el hueco que queda cuando sólo se usan herramientas que dan soporte a las etapas léxica y sintáctica. Todo esto hace que resulte bastante cómodo especificar un procesador de lenguaje con esta herramienta incluso en los casos más complejos, y prueba de ello es que ANTLR está siendo cada vez más usada en el desarrollo de estas aplicaciones. Según afirman los propios autores, la media de descargas mensual es de 5000, una cifra nada desdeñable teniendo en cuenta que el colectivo de usuarios que demanda este tipo de herramientas es relativamente reducido.

Tras haber puesto en marcha por primera vez este conjunto de prácticas y observar cómo los alumnos han ido asimilando los distintos elementos de teoría y laboratorio nuestras principales sensaciones son las siguientes:

- El uso del modelo descendente para los analizadores resulta un poco chocante en la etapa de análisis léxico, pero después se acepta con naturalidad ya que pasar al análisis sintáctico y de árboles sólo requiere la adaptación de la misma idea a problemas distintos.
- Los predicados sintácticos son considerados como una idea interesante y ayudan al alumno a reflexionar sobre la ambigüedad en el análisis sintáctico.
- La equivalencia entre símbolo (en la gramática) y método (en el modelo de ejecución Java) hace que comprender los conceptos de atributos sintetizados y heredados sea más sencillo ya que en todo momento el símil del parámetro como atributo les sirve de referencia.
- La notación para especificar la construcción de árboles de sintaxis abstracta es asimilada con facilidad y hasta cierto punto sorprende como con tan poco se puede expresar tanto.

En lo tocante al lenguaje de desarrollo, el uso de Java ha supuesto una mejora importantísima, especialmente en cuanto al tiempo empleado en la depuración de programas. En nuestra experiencia anterior con el lenguaje C comprobábamos constantemente que la combinación de un lenguaje poco disciplinado como C, con la generación automática de programas suponía una mezcla explosiva. Encontrar un error en alguno de los fuentes en juego era una tarea ardua. La disciplina a la que Java obliga ha aliviado considerablemente esta tarea.

Tanto la integración de todas las etapas (léxica, sintáctica y semántica) en una única herramienta como el uso de Java como lenguaje de desarrollo han facilitado considerablemente el desarrollo de un procesador de lenguaje. La

impresión que nos queda tras un cuatrimestre de experiencia es que los alumnos deben dedicar menos tiempo a resolver los problemas que le plantea el entorno de desarrollo y pueden por tanto dedicar más tiempo a lo realmente importante: comprender el lenguaje que se pretende procesar y especificar el reconocedor de la manera más abstracta y segura posible.

En una encuesta realizada a los alumnos al final del cuatrimestre, el uso de la herramienta ANTLR en el laboratorio ha sido muy bien valorado, este dato es especialmente importante teniendo en cuenta que en encuestas de años anteriores este aspecto era de los que más críticas recibía.

Una opinión que nos interesaba conocer especialmente era la de los alumnos repetidores. Ellos conocen el método antiguo y pueden comparar de una forma, en ciertos aspectos, más objetiva que los profesores que promueven el cambio. En este sentido, los alumnos repetidores consultados opinan que la asignatura ha ganado con la nueva herramienta, excepto que para ellos dicho cambio les ha supuesto tener que "estudiarse" algo nuevo.

5. Conclusiones

En este trabajo hemos presentado nuestra experiencia docente con el uso de ANTLR como herramienta para las prácticas de laboratorio de la asignatura Procesadores de Lenguaje. Después de un curso con ella estamos en condiciones de aconsejar su uso por todas las razones que hemos ido desgranando a lo largo del artículo. Tras utilizar varias herramientas relacionadas con el procesamiento de lenguajes, en ANTLR hemos encontrado muchas soluciones que la convierten en una alternativa muy seria para la construcción de compiladores.

Nuestra experiencia aún está a la mitad, ya que en nuestro plan de estudios la materia troncal Procesadores de Lenguaje está distribuida en dos asignaturas: PL1 y PL2. Evidentemente el cambio de herramienta también afecta a PL2, en ella profundizaremos en el proceso de compilación de lenguajes de programación y estudiaremos en detalle cada una de sus etapas. Desde la comprobación de la semántica estática a la generación de código que se realizará sobre la máquina virtual de Java con la ayuda del ensamblador simbólico Jamaica. Cuando en julio de 2005 finalicemos el segundo parcial habremos completado la primera iteración de nuestra experiencia. Pero por ahora, y tras haber finalizado PL1, podemos decir que el camino elegido no nos está defraudando.

Bibliografía

[1] Aho, A.V., Sethi, R., Ullman, J.D., *Compiladores. Principios, técnicas y herramientas*, Addison-Wesley, 1990.

[2] Fischer, C.N., LeBlanc, R.J., *Crafting a Compiler with C*, Benjamin Cummings, 1991.

[3] Página de la asignatura Procesadores de Lenguaje 1, *http://www.lsi.us.es/~troyano /mat_pl1.htm*

[4] Página oficial de la herramienta ANTLR, *http://www.antlr.org/*

[5] Página oficial del entorno de trabajo Eclipse, *http://www.eclipse.org/*

[6] Parr, T., *ANTLR Reference Manual. 2.7.4*, disponible en *http://www.antlr.org/*, 2004.

[7] Sippu, S., Soisalon-Soininen, E., *Parsing Theory*, Springer Verlag, 1988.

[8] Watt, D.A., Brown, D.F., *Programming Language Processors in Java*, Prentice Hall, 2000.