Generación automática de analizadores sintácticos a partir de esquemas de análisis*

Carlos Gómez Rodríguez y **Jesús Vilares** y **Miguel A. Alonso**
(solrac888@yahoo.com, jvilares@udc.es, alonso@udc.es)
Departamento de Computación, Universidade da Coruña
Campus de Elviña, 5
15071 A Coruña

Resumen: Los esquemas de análisis sintáctico son un formalismo de definición de algoritmos de análisis sintáctico que permite abstraer detalles de bajo nivel. En este trabajo, presentamos una técnica de compilación para transformar automáticamente un esquema de análisis sintáctico en una implementación ejecutable. Concretamente, a partir de un esquema obtendremos el código Java que implementa el analizador correspondiente, incluyendo técnicas de indexación adaptadas a cada esquema concreto para garantizar eficiencia. La técnica presentada es general, permitiendo trabajar con todo tipo de esquemas para gramáticas independientes del contexto, e incluye mecanismos de extensibilidad para definir nuevos elementos susceptibles de aparecer en dichos esquemas, al tiempo que es fácilmente generalizable a otros formalismos gramaticales.
Palabras clave: Análisis sintáctico, esquemas de análisis sintáctico, gramáticas independientes del contexto, compilación

Abstract: The parsing schemata formalism allows us to describe parsing algorithms in a simple way by capturing their fundamental semantics while abstracting low-level detail. In this work, we present a compilation technique allowing automatic transformation of parsing schemata to executable implementations of their corresponding algorithms. Taking a simple description of a schema as input, our technique generates Java code for the corresponding parsing algorithm, including schema-specific indexing code in order to attain efficiency. Our technique is general enough to be able to handle all kinds of schemata for context-free grammars, providing an extensibility mechanism which allows the user to define custom notational elements, and it could also be easily generalized to other grammatical formalisms.
Keywords: Parsing, parsing schemata, context-free grammars, compiling

1. *Introducción*

El análisis sintáctico, que permite obtener la estructura de una frase de acuerdo con una descripción formal del lenguaje en forma de gramática, es un paso de gran relevancia en la cadena de actividades implicadas en el análisis automático de frases en lenguaje natural. En las últimas décadas se han ido desarrollando diferentes algoritmos para llevar a cabo esta tarea. Muchos de estos algoritmos difieren en gran medida en la manera de acometer el análisis, utilizando aproximaciones distintas para llegar a los mismos o parecidos resultados, y cada uno se adapta más a un tipo de situación concreta.

Los esquemas de análisis sintáctico, introducidos en (Sikkel, 1997), proporcionan una manera uniforme de describir, analizar y comparar diferentes algoritmos de análisis sintáctico. Para ello, se basan en considerar dicho análisis como un proceso de generación de resultados intermedios denominados *ítems*. La frase que se desea analizar genera un conjunto inicial de ítems, y el análisis consiste en la aplicación de una serie de reglas que permiten generar nuevos ítems que contienen información sobre la estructura de la frase, hasta que se llega a alguno que contiene explícitamente el árbol sintáctico, o bien garantiza su existencia y permite obtenerlo

* Parcialmente financiado por el Ministerio de Educación y Ciencia y FEDER (TIN2004-07246-C03-02), y por la Xunta de Galicia (PGIDIT02PXIB30501PR, PGIDIT02SIN01E y PGIDIT03SIN30501PR).

con facilidad.

Casi todos los analizadores conocidos se pueden ver desde esta perspectiva (excluyendo los no constructivos, como los basados en redes neuronales). Para ello, para cada algoritmo deberemos determinar qué tipos de ítems existirán, qué reglas deductivas permitirán generar nuevos ítems a partir de los iniciales, y cuáles serán los *ítems "finales"* que determinan que la frase ha sido analizada.

En (Sikkel, 1997) se hace una explicación detallada y rigurosa de los esquemas de análisis sintáctico. Aquí nos limitaremos a una breve explicación basada en un ejemplo: el esquema para el algoritmo de Earley, que se describe en (Earley, 1970). Dada una gramática independiente del contexto $G = (N, \Sigma, P, S)$[1] y una oración de longitud n, denotada a_1 a_2 ... a_n, dicho algoritmo viene descrito por el siguiente esquema[2]:

Conjunto de ítems:
$\{[A \rightarrow \alpha.\beta, i, j] \mid A \rightarrow \alpha\beta \in P \wedge 0 \leq i < j\}$

Ítems iniciales (hipótesis):
$\{[a_i, i-1, i] \mid 0 < i \leq n\}$

Pasos deductivos:

EARLEY INITTER: $\dfrac{}{[S \rightarrow .\alpha, 0, 0]}\ S \rightarrow \alpha \in P$

EARLEY SCANNER: $\dfrac{\begin{array}{c}[A \rightarrow \alpha.a\beta, i, j]\\ [a, j, j+1]\end{array}}{[A \rightarrow \alpha a.\beta, i, j+1]}$

EARLEY PREDICTOR: $\dfrac{[A \rightarrow \alpha.B\beta, i, j]}{[B \rightarrow .\gamma, j, j]}\ B \rightarrow \gamma \in P$

EARLEY COMPLETER: $\dfrac{\begin{array}{c}[A \rightarrow \alpha.B\beta, i, j]\\ [B \rightarrow \gamma., j, k]\end{array}}{[A \rightarrow \alpha B.\beta, i, k]}$

Ítems finales:
$\{[S \rightarrow \gamma., 0, n]\}$

Los ítems en este algoritmo se componen de tres elementos: una regla gramatical a la que se le añade un símbolo especial (punto) en alguna posición de su parte derecha y dos números enteros que denotan posiciones de la cadena de entrada. El significado concreto de un ítem $[A \rightarrow \alpha.\beta, i, j]$ en este algoritmo se puede interpretar como: "La gramática permite construir un árbol de raíz A, donde los hijos directos de A son los símbolos de la cadena $\alpha\beta$, y los nodos hoja de los subárboles que parten de los símbolos de α forman la parte $a_{i+1} \ldots a_j$ de la cadena de entrada".

El algoritmo producirá un análisis sintáctico completo de la oración si se llega a algún ítem de la forma $[S \rightarrow \gamma., 0, n]$, pues según la interpretación vista, este ítem nos garantiza la existencia de un árbol sintáctico de raíz S cuyos nodos hoja son $a_1 \ldots a_n$, es decir, los símbolos de la cadena de entrada.

Los pasos deductivos $\dfrac{\eta_1 \ldots \eta_m}{\xi}\ \Phi$ permiten deducir el ítem especificado en su consecuente ξ a partir de los que aparecen en sus antecedentes $\eta_1 \ldots \eta_m$. Las *condiciones laterales* Φ especifican qué valores pueden tomar las variables que aparecen en los antecedentes y consecuente, y hacen referencia a reglas de la gramática. En nuestro caso, los pasos deductivos INITTER y PREDICTOR permiten inicializar el análisis, produciendo ítems con el punto en la primera posición de la parte derecha de sus reglas, que representan la aplicación de una regla de producción sin haber reconocido todavía ningún carácter de la cadena de entrada. En realidad, podríamos generar en el INITTER todos los ítems de la forma $[A \rightarrow .\alpha, j, j]$ para cada regla $A \rightarrow \alpha$ de la gramática, pues la presencia de estos ítems no implica ninguna suposición incorrecta sobre la cadena de entrada. Si hiciésemos esto, tendríamos el algoritmo "bottom-up Earley". Esta versión de Earley, sin embargo, utiliza el paso PREDICTOR para intentar evitar la generación de ítems de este tipo que no vayan a ser útiles en el análisis.

El punto en las producciones, como hemos visto, delimita el fragmento de la parte derecha que ya ha sido reconocido, y los pasos SCANNER y COMPLETER permitirán ir desplazándolo hacia la derecha. El paso SCANNER representa la operación de leer y reconocer un símbolo terminal de la cadena de entrada, mientras que el COMPLETER representa el reconocimiento de un símbolo no terminal que se sabe que puede ser sustituido por una parte de la cadena de entrada.

[1] Donde N representa el conjunto de símbolos no terminales, Σ el de símbolos terminales, P las reglas de producción y S el axioma.

[2] De aquí en adelante se utilizará la convención usual de denotar los símbolos no terminales de una gramática mediante letras mayúsculas (A, B...), los símbolos terminales mediante minúsculas (a, b...) y las cadenas de símbolos, terminales y no terminales, con letras griegas (α, β...).

2. *Motivación*

Los esquemas de análisis sintáctico se ubican en un nivel de abstracción superior al de los algoritmos. Como se puede ver en el ejemplo, un esquema especifica unos pasos a ejecutar y unos resultados intermedios a obtener para llegar a un análisis sintáctico de la cadena de entrada; pero abstrae detalles de implementación como el orden en que se deben de ejecutar los pasos o las estructuras de datos para representar y almacenar los ítems.

Esta abstracción de los detalles de bajo nivel hace que los esquemas sean muy útiles, permitiéndonos definir analizadores sintácticos de manera muy sencilla, así como compararlos y analizar aspectos como su corrección y completitud o su complejidad computacional. Sin embargo, si queremos comprobar en un computador los resultados que produce el analizador descrito por un esquema, no tendremos más remedio que implementarlo en algún lenguaje de programación, teniendo que preocuparnos de los aspectos que el esquema ocultaba.

La técnica que aquí presentamos automatiza este trabajo, permitiendo compilar esquemas de análisis sintáctico a una implementación del analizador correspondiente en lenguaje Java.

Tomaremos como entrada una representación muy sencilla de un esquema de análisis sintáctico, prácticamente coincidente con la notación formal que vimos antes. Por ejemplo, una descripción del esquema correspondiente al algoritmo de Earley sería:

```
@step EarleyInitter
----------------------------- S -> alpha
[ S -> . alpha , 0 , 0 ]

@step EarleyScanner
[ A -> alpha . a beta , i , j ]
[ a , j , j+1 ]
---------------------------------
[ A -> alpha a . beta , i , j+1 ]

@step EarleyCompleter
[ A -> alpha . B beta , i , j ]
[ B -> gamma . , j , k ]
---------------------------------
[ A -> alpha B . beta , i , k ]

@step EarleyPredictor
[ A -> alpha . B beta , i , j ]
--------------------------------- B -> gamma
[ B -> . gamma , j , j ]
```

Aplicando la técnica de compilación a este fichero de entrada, se generará un conjunto de ficheros con código Java que a su vez se pueden compilar directamente, produciendo un ejecutable que implementa el algoritmo de Earley para gramáticas y frases dadas. El proceso se muestra en la figura 1.

3. *Descripción general de la técnica de compilación*

Para transformar la descripción declarativa del esquema de análisis sintáctico en su implementación Java, procederemos a grandes rasgos de la siguiente manera:

- Cada uno de los pasos deductivos del esquema dará lugar a una clase.
- El programa generado creará una instancia de dicha clase por cada posible manera de satisfacer las condiciones laterales.
- Estas clases cuentan con un método `paso` que intenta aplicar el paso deductivo a un ítem dado, devolviendo los nuevos ítems que se obtienen si es que el paso es aplicable. Para ello, se comprueba si el ítem dado empareja con alguno de los antecedentes del paso deductivo, y para cada emparejamiento con éxito se buscan combinaciones de ítems que hayamos generado anteriormente para satisfacer el resto de los antecedentes. Cada combinación de ítems que satisfaga todos los antecedentes dará lugar a una instanciación de las variables del paso que generará un ítem a partir del consecuente.
- La ejecución de los pasos deductivos está coordinada por la *máquina deductiva de análisis sintáctico*, un algoritmo genérico cuya implementación será la misma para cualquier esquema, respondiendo al siguiente seudocódigo:

```
pasos = {pasos deductivos instanciados};
ítems = {ítems asociados a la frase};
agenda = [ítems asociados a la frase];
Para cada paso deductivo de antecedente
    vacío (p) en pasos {
 resultados = p.paso([]);
 ítems.añadir(resultados);
 agenda.añadirAlFinal(resultados);
 pasos.quitar(p);
}
Mientras agenda no vacía {
 ItemActual = agenda.quitarPrimero();
 Para cada paso deductivo aplicable a
    ItemActual (p) en pasos {
  resultados = p.paso(ItemActual);
  ítems.añadir(resultados);
  agenda.añadirAlFinal(resultados);
 }
}
Devolver ítems;
```

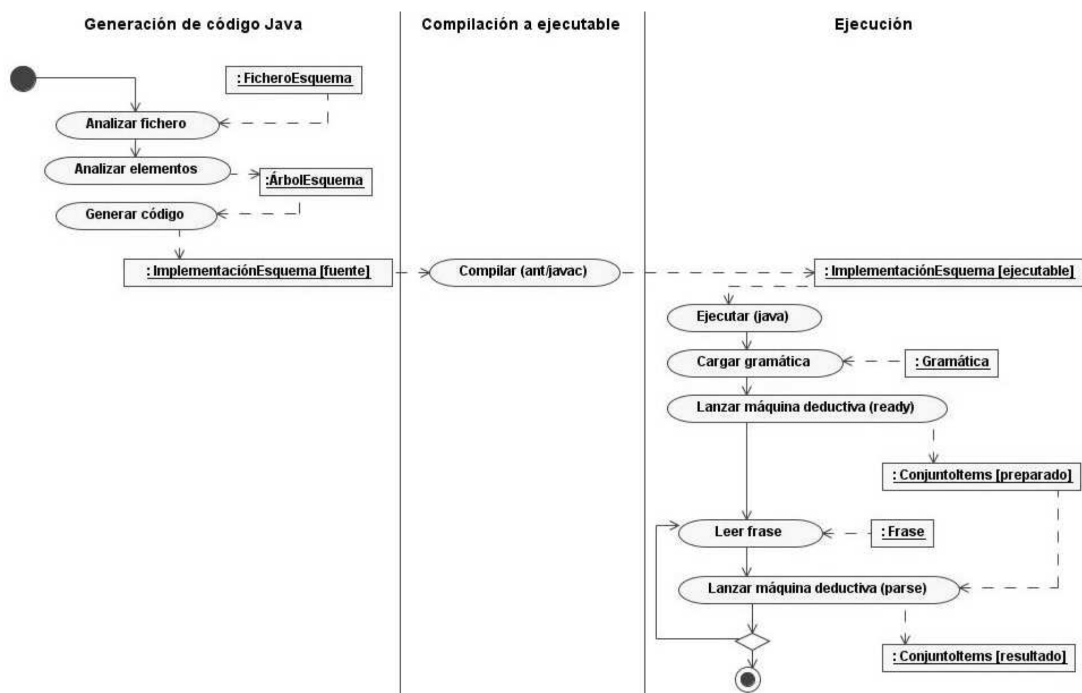

Figura 1: Diagrama de actividad de la técnica de compilación.

El algoritmo se basa en mantener un conjunto con todos los ítems que se han generado, ya sea como hipótesis del algoritmo o como resultado de la aplicación de pasos, y una *agenda* (implementada como una cola) conteniendo los ítems que todavía no hemos intentado utilizar para generar otros nuevos. Cuando la agenda se vacíe habremos generado todos los ítems posibles, y la frase de entrada pertenecerá al lenguaje si hay algún ítem final en el conjunto de ítems. La corrección y completitud del algoritmo se puede probar de forma sencilla por inducción.

Para que la implementación generada sea computacionalmente eficiente, se utilizan tres tipos de indexación:

- *Índices para búsquedas*: se utilizan en la aplicación de los pasos deductivos para buscar rápidamente ítems que se puedan emparejar con los antecedentes del paso.
- *Índices de existencia*: se utilizan cuando la aplicación de un paso genera un ítem, para ver si ese ítem ya existe en el conjunto. De este modo se evita añadir ítems duplicados.
- *Índices sobre pasos deductivos*: se utilizan para restringir el conjunto de "pasos deductivos aplicables" que la máquina deductiva intenta ejecutar sobre los ítems, permitiendo descartar de entrada aquéllos que no pueden emparejar con el ítem dado.

Los elementos sobre los que conviene indexar varían con cada esquema de análisis sintáctico, con lo cual la generación del código para estos índices debe tener en cuenta las características de cada esquema concreto. Para la generación de los índices de búsqueda se tiene en cuenta qué variables estarán instanciadas en el momento en que el código de los pasos haga una búsqueda de ítems para emparejar con un antecedente, aspecto que se conoce en tiempo de compilación del esquema, y se crearán índices sobre ítems utilizando estos elementos instanciados. Por ejemplo, para el paso COMPLETER del algoritmo de Earley se crearán dos índices de búsqueda: uno que indexa los ítems por el símbolo inmediatamente posterior al punto y por la segunda posición de cadena de entrada, para buscar ítems que emparejen con el primer antecedente cuando ya se ha emparejado el segundo; y otro que indexa por el símbolo de la parte izquierda de la producción y la primera posición de cadena de entrada, para buscar ítems de la forma del segundo antecedente cuando se ha partido del primero.

La generación de los índices de existencia es parecida pero más sencilla, pues se podrá indexar por cualquier componente de los ítems, dado que cuando se quiere decidir

si existe un ítem éste siempre es totalmente conocido y todas sus variables están instanciadas, al contrario de lo que sucede en las búsquedas.

La generación de los índices sobre pasos deductivos se basa en tener en cuenta qué variables del paso tomarán valor durante su instanciación, es decir, cuáles aparecen en las condiciones laterales. Estas variables tendrán un valor concreto para cada instancia de un paso deductivo, con lo cual nos sirven para discriminar a priori qué pasos pueden ser aplicables a un ítem dado y cuáles no.

4. Elementos de los esquemas

Los tipos de elementos que pueden aparecer en un esquema de análisis sintáctico no están limitados a los que se han visto en el ejemplo de Earley. La notación de los esquemas es abierta, y podría interesar incluir en un esquema cualquier tipo de objeto matemático.

Evidentemente es imposible reconocer de entrada cualquier tipo de elemento que se quiera incluir en un esquema, así que hemos optado por proporcionar una serie de elementos por defecto y un mecanismo de extensibilidad que permite incorporar nuevos tipos de elementos de forma muy sencilla. Dicho mecanismo se basa en clasificar los elementos notacionales en cuatro tipos, de acuerdo con el tratamiento que reciben en la fase de generación de código, de manera que cualquier nuevo tipo de elemento que se quiera añadir debería poder encuadrarse en uno de estos tipos:

- *Elementos simples:* Son elementos atómicos, sin estructura, que en cada momento pueden estar o no instanciados. En el caso de que lo estén, tomarán un valor, acotado o no, que normalmente será convertible a una clave para indexar. Ejemplos: símbolos gramaticales, enteros, posiciones de cadena de entrada, el punto de las producciones de Earley, probabilidades...

- *Expresiones*: Representan expresiones sobre elementos simples o sobre otras expresiones. Por ejemplo, $i + 1$ sería una expresión sobre enteros, y $tree[A, B]$ sobre símbolos no terminales. Cuando todos los elementos simples que aparezcan en una expresión estén instanciados a un valor concreto, la expresión se tratará como un elemento simple cuyo valor se obtendrá de aplicar la operación que representa (por ejemplo, la suma). Dicha operación se debe definir mediante una expresión en Java al crear el tipo de elemento: de este modo el generador de código podrá, por ejemplo, convertir las sumas de posiciones de cadena de entrada que aparecen en los esquemas a sumas de enteros en el código generado.

- *Elementos compuestos*: Representan una secuencia de elementos de cualquier tipo de longitud finita y conocida. Los elementos compuestos sirven para dar estructura a los ítems. Así, por ejemplo, un ítem $[A \to \alpha.B\beta, i, j]$ del algoritmo de Earley será un elemento compuesto con tres componentes: la primera componente es a su vez un elemento compuesto, representando la regla, y las otras dos son elementos simples de tipo posición en cadena de entrada.

- *Secuencias*: Representan una secuencia de elementos de un mismo tipo, de longitud finita pero desconocida hasta que se haya instanciado la secuencia a un valor. Las cadenas α, β y γ que aparecen en el esquema del algoritmo de Earley visto como ejemplo son elementos de este tipo, dado que pueden representar cadenas de símbolos de cualquier longitud. Este hecho será tenido en cuenta por el generador de código a la hora de llevar a cabo los emparejamientos al ejecutar los pasos deductivos.

Para añadir un nuevo tipo concreto de elemento al compilador de esquemas, habrá que definirlo como subclase de uno de estos cuatro tipos básicos e implementar la interfaz de dicho tipo de manera acorde con la semántica de nuestro elemento. Además, será necesario incluir en un fichero de configuración (o bien directamente en el fichero de especificación de los esquemas que utilicen ese elemento) una o varias expresiones regulares que definan el formato de las cadenas que representarán ese tipo de elemento en los esquemas. El analizador que lee el fichero de esquemas cuenta con un mecanismo para leer estas expresiones regulares y llamar al método que definamos para crear una instancia de nuestro tipo de elemento a partir de la cadena correspondiente. Esta llamada se hace mediante los mecanismos de reflexión del lenguaje Java, con lo cual no es necesario recompilar el sistema; sino que él mismo cargará dinámicamente la clase añadida. Esto proporciona un alto grado de extensibilidad, permitiendo trabajar fácilmente con esquemas que contengan todo tipo de ítems no predefinidos.

5. Ejemplos y rendimiento

A continuación se muestran una serie de ejemplos de la utilización de nuestra técnica de compilación con diferentes algoritmos y gramáticas. Concretamente, nos centraremos en tres conocidos algoritmos de análisis para gramáticas independientes del contexto: CYK (Kasami, 1965; Younger, 1967), Earley (Earley, 1970) y Left-Corner (Rosenkrantz y Lewis II, 1970).

Para el algoritmo de Earley, se utilizará el fichero de esquema que se describió anteriormente[3]. Para el algoritmo CYK, el esquema de análisis es:

```
@step CYKMain
[ B , i , j ]
[ C , j , k ]
--------------------- A -> B C
[ A , i , k ]

@step CYKAux
[ a , i , j ]
--------------------- A -> a
[ A , i , j ]

@goal [ S , 0 , length ]
```

En cuanto a Left-Corner, se ha utilizado el esquema para la variante *sLC* que se describe en (Sikkel, 1997):

```
@step SimplifiedLCRelationshipInitter
----------------------------- A -> alpha
[ . , . , A , A ]

@step SimplifiedLCRelationshipClosureFinder
[ . , . , A , B ]
----------------------------- B -> C alpha
[ . , . , A , C ]

@step SimplifiedLCInitter
----------------------------- S -> gamma
[ S -> . gamma , 0 , 0 ]

@step SimplifiedLCRuleToItem
----------------------------- B -> beta
[ B -> A beta ]

@step SimplifiedLCNonterminal
[ E , i ]
[ A -> alpha . , i , j ]
[ . , . , E , B ]
[ B -> A beta ]
------------------------------
[ B -> A . beta , i , j ]

@step SimplifiedLCTerminal
[ E , i ]
[ a , i , i+1 ]
[ . , . , E , B ]
[ B -> a beta ]
-------------------------------
[ B -> a . beta , i , i+1 ]

@step SimplifiedLCEpsilon
[ E , i ]
[ B -> ]
-------------------------------
[ B -> . , i , i ]

@step SimplifiedLCPredictor
[ C -> gamma . E delta , k , i ]
-------------------------------
[ E , i ]

@step SimplifiedLCScanner
[ A -> alpha . B beta , i , j ]
[ B , j , j+1 ]
-------------------------------
[ A -> alpha B . beta , i , j+1 ]

@step SimplifiedLCCompleter
[ A -> alpha . B beta , i , j ]
[ B -> gamma . , j , k ]
-------------------------------
[ A -> alpha B . beta , i , k ]

@goal [ S -> alpha . , 0 , length ]
```

Los algoritmos resultantes de la compilación de los tres esquemas se probaron con frases de dos gramáticas diferentes: por un lado, la gramática inglesa del corpus Susanne (Sampson, 1994), una gramática de lenguaje natural con más de 17.000 reglas; por otro lado, una gramática sencilla que tiene como lenguaje asociado el conjunto de cadenas de paréntesis balanceadas:

$S \rightarrow SS$
$S \rightarrow (S)$
$S \rightarrow \epsilon$

Que, en el caso del algoritmo CYK, se pasa a forma normal de Chomsky para dar lugar a:

$S \rightarrow SS$
$S \rightarrow (C$
$S \rightarrow ()$
$C \rightarrow S)$

Los resultados para la gramática del corpus Susanne nos dan una idea de cómo funcionan los diferentes algoritmos en casos reales de análisis de lenguaje natural, mientras que la gramática de los paréntesis nos permi-

[3]En realidad los resultados corresponden a un esquema para Earley ligeramente modificado, que tiene diferentes pasos PREDICTOR según la longitud de la parte derecha de la regla gramatical con que se instancia (0, 1, 2 o más de 2 símbolos). Esta modificación del esquema hace que la indexación de existencia sea más efectiva.

te hacer pruebas en casos extremos, dada su gran ambigüedad. Para la gramática de Susanne, las pruebas se han hecho con respecto a frases de diferentes longitudes generadas automáticamente. Esto nos permite evaluar el comportamiento de los analizadores llegando a longitudes mayores de lo que cabe esperar encontrar en un entorno real (hasta 512 palabras).

En el caso de la gramática de los paréntesis también se han probado frases de diferentes longitudes; pero para cada longitud se ha tomado una cadena aleatoria, una sin ambigüedad (de la forma (((...(())...)))) y una altamente ambigua (de la forma ()()()...()()). Los tiempos de ejecución para todas estas ejecuciones se muestran en las figuras 2 y 3.

De los tiempos obtenidos[4] podemos extraer las siguientes conclusiones:

- Aunque los tres algoritmos estudiados tienen una complejidad teórica de $O(n^3)$ en el peor caso, lo cierto es que en casos prácticos se comportan mucho mejor. Los tiempos obtenidos para las palabras generadas con la gramática del Susanne están cerca de ser lineales con respecto a su longitud. En el caso de la gramática de los paréntesis también sucede esto, excepto en el caso de las frases altamente ambiguas, que se acercan al peor caso.
- El algoritmo que más rápidamente analiza con respecto a la gramática del Susanne es CYK, seguido de Earley y Left-Corner. Sin embargo, los tiempos de Earley parecen crecer algo más moderadamente que los de los otros algoritmos, con lo cual es posible que Earley llegase a ser más rápido para alguna longitud mayor que las probadas.
- En el caso de la gramática de los paréntesis, el algoritmo más rápido sigue siendo CYK; pero en este caso Earley supera a Left-Corner para las longitudes probadas. Esto no resulta extraño teniendo en cuenta que la gramática es muy simple, las relaciones Left-Corner entre símbolos nos proporcionan menos información que en el caso de Susanne, y los beneficios que se obtienen de considerar estas relaciones se ven superados por los inconvenientes de tener un esquema más complicado, con más pasos deductivos, tipos de ítems y modos de indexación. En general, la simplicidad de los ítems y de su gestión puede influir en el hecho de que CYK proporcione los mejores rendimientos.
- En general, podemos comprobar que la gramática y el tipo de frases a analizar influyen en el rendimiento de los algoritmos, siendo cada uno de ellos idóneo para distintas situaciones. La técnica de compilación de esquemas de análisis sintáctico aquí presentada proporciona una forma sencilla de evaluar este rendimiento en distintos casos y decidir qué algoritmo utilizar en cada momento.

[4]Para las pruebas se ha utilizado una máquina con las siguientes características: procesador Intel Pentium M a 1500 MHz, 512 MB de RAM, máquina virtual Java HotSpot de Sun en su versión 1.4.2_01-b06 ejecutándose sobre Windows XP.

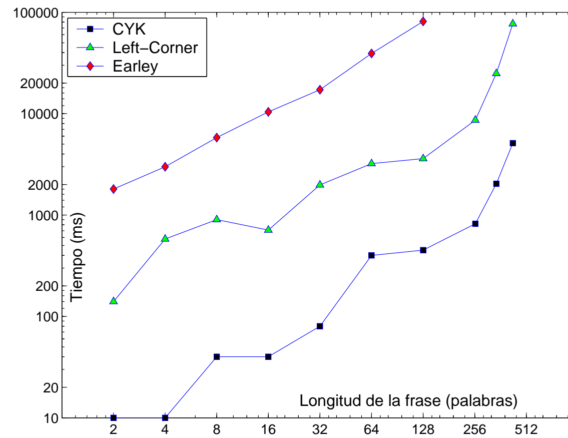

Figura 2: Rendimientos para la gramática del corpus Susanne.

6. *Conclusiones y trabajo futuro*

En este trabajo hemos presentado una técnica de compilación que permite transformar, de manera directa y automática, un esquema de análisis sintáctico en una implementación ejecutable del algoritmo descrito por el esquema. Como hemos visto, la técnica es válida para todo tipo de esquemas asociados a gramáticas independientes del contexto, como pueden ser los correspondientes a los algoritmos Earley, CYK o Left-Corner.

Los índices adaptados a cada esquema que se incluyen en el código generado hacen que los tiempos de ejecución de los algoritmos generados se ajusten a lo que cabe esperar de su complejidad teórica.

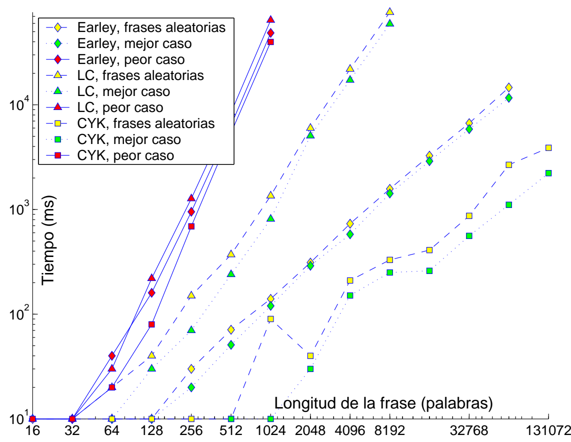

Figura 3: Rendimientos para la gramática de los paréntesis.

La compilación de esquemas de análisis sintáctico resulta muy útil a la hora de diseñar, analizar y prototipar algoritmos de análisis, permitiéndonos probar los esquemas y comprobar sus resultados y su rendimiento. Comparando los algoritmos de Earley, CYK y Left-Corner para diferentes gramáticas y palabras, hemos visto que no existe un algoritmo idóneo para cualquier tipo de gramática y frase; sino que convendrá más utilizar uno u otro en cada caso concreto. Compilando los esquemas podemos comparar fácilmente rendimientos y escoger el algoritmo adecuado.

La técnica de compilación presentada se podría generalizar en el futuro para soportar, además de gramáticas independientes del contexto, otros formalismos gramaticales adecuados para lenguajes naturales, como por ejemplo las gramáticas de adjunción de árboles (Joshi y Schabes, 1997).

Bibliografía

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Eisner, Jason, Eric Goldlust, y Noah A. Smith. 2004. Dyna: A declarative language for implementing dynamic programs. En *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (Companion Volume)*, páginas 218–221, Barcelona.

Joshi, Aravind K. y Yves Schabes. 1997. Tree-adjoining grammars. En Grzegorz Rozenberg y Arto Salomaa, editores, *Handbook of Formal Languages. Vol 3: Beyond Words*. Springer-Verlag, Berlin/Heidelberg/New York, capítulo 2, páginas 69–123.

Kasami, T. 1965. An efficient recognition and syntax algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Lab., Bedford, Massachussetts.

Rosenkrantz, D. J. y P. M. Lewis II. 1970. Deterministic Left Corner parsing. En *Conference Record of 1970 Eleventh Annual Meeting on Switching and Automata Theory*, páginas 139–152, Santa Monica, CA, USA, Octubre. IEEE.

Sampson, G. 1994. The Susanne Corpus, Release 3. School of Cognitive Computing Sciences, University of Sussex, Falmer, Brighton, England.

Sikkel, Klaas. 1997. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Texts in Theoretical Computer Science — An EATCS Series. Springer-Verlag, Berlin/Heidelberg/New York.

Younger, D. H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.