IMPLEMENTACIÓN EN JAVA DEL ALGORITMO LR(1)

José Luis Díez Fernández[1], JosuKa Díaz Labrador[2], Andoni Eguíluz Morán[2]

[1] ESIDE - Universidad de Deusto, ENSTB - Telecom Bretagne , [2]ESIDE - Universidad de Deusto
jldiez@computer.org , josuka@inf.deusto.es , eguiluz@inf.deusto.es

1. INTRODUCCIÓN

Para quién ha tenido algún contacto con las computadoras, o aún mejor, con la programación, la palabra ***compilador*** tiene un sentido claro: sistema informático que permite preparar programas escritos en un cierto lenguaje (denominado fuente, como Pascal, C, C++, Java...) para su ejecución por parte de la computadora. Ello se consigue traduciendo el programa fuente al lenguaje propio de la computadora, el llamado lenguaje máquina, obteniendo un programa objeto que luego se ejecuta.

Si este es efectivamente el interés primigenio de los compiladores (que han posibilitado la existencia de los lenguajes de programación de alto nivel, y por ende, el espectacular desarrollo de la informática desde finales de los 50, sin menospreciar otros aspectos igualmente importantes), no queremos comenzar este artículo sin romper una lanza a favor de la posibilidad de aplicar las técnicas de compilación que se conocen hay en día en otra clase de problemas, incluso ni siquiera relacionados con la programación.

Un ejemplo de actualidad (más si estás leyendo estas páginas en la computadora) es un cliente de páginas telaraña (web dicen en inglés). Conceptualmente (aunque por supuesto intervienen otros detalles nada desdeñables) un programa de esta clase ha de resolver dos necesidades: un servicio de telecomunicaciones (como es obvio) y una presentación de información. Es en esta segunda necesidad donde el conocimiento de las técnicas de compilación es necesario para quien diseña tal programa: como se sabe, las páginas telaraña han de escribirse observando las normas de un cierto lenguaje, llamado HTML, y el cliente telaraña ha de analizar las páginas (del mismo modo que un compilador de C analiza la información contenida en un programa C), para después ofrecer una visualización de las mismas (tarea equiparable a la generación del programa ejecutable que ha de realizar el compilador de C). Este último aspecto tiene características ciertamente diferentes en uno y otro ejemplo, pero la tarea de analizar y verificar la información de entrada (página telaraña o programa C) se resuelve aplicando exactamente la misma clase de técnicas y metodologías.

En definitiva, la clave está en la palabra *lenguaje*. Siempre que exista una información «codificada» mediante una notación particular, con unas características «complejas» en cuanto a léxico (unidades mínimas de información), sintaxis (estructura de los elementos que se concatenan) y semántica (relaciones adicionales entre unos elementos y otros), hay muchas probabilidades de que las técnicas de compilación puedan aplicarse para la solución de problemas en los que interviene esa información como entrada. Por ello, los términos *traductor* o *procesador de lenguaje* son muy adecuados como generalización del concepto de compilador (si bien es cierto que no hasta el punto de aplicarse a los lenguajes llamados naturales, como el castellano, inglés, etc.: son «demasiado complejos»). Y lo cierto es que en numerosos campos (Internet es uno especialmente relevante) están surgiendo notaciones, protocolos, formalizaciones... que al fin y al cabo, no son otra cosa que lenguajes en el sentido explicado, y que plantean problemas que muy bien podrían resolverse con algunos de los métodos de construcción de compiladores.

El punto de partida del proyecto es crear un sistema basado en páginas telaraña como apoyo o complemento de las llamadas 'herramientas de generación de compiladores'

El problema que particularmente planteamos en este artículo trata de una cuestión técnica relacionada con la construcción de compiladores (ya sea para traducir un lenguaje de programción, o para aplicar a uno de esos problemas más generales que hemos comentado), y es el punto de partida de un proyecto más ambicioso: crear un sistema basado en páginas telaraña (con programas Java) como apoyo o complemento de las llamadas *herramientas de generación de compiladores*, que permiten la construcción automática de ciertas partes de un compilador a partir de descripciones formales del lenguaje fuente (para los ya iniciados, a partir de expresiones regulares, gramáticas independientes del contexto, etc.). Justamente estas partes son las que han de construirse en la solución de esos problemas más generales comentados en los párrafos anteriores (de ahí que nos hayamos extendido tanto en ello,

y una de las razones por las que nos interesa este proyecto).

Una de las fases fundamentales de un compilador o traductor es el 'análisis sintáctico': el compilador ha de construir el árbol que refleja la estructura jerárquica de los diferentes elementos del programa fuente

Más en concreto, entendemos que el sistema que pretendemos construir cumple varios objetivos:

* Primero, servir de apoyo en la docencia de las asignaturas de compilación: la universalización del sistema telaraña (junto con el lenguaje Java) posibilita también la seguridad de que el material que aporta el profesor va a llegar a todos sus alumnos; antes, diversos detalles «técnicos» podían llegar a impedir seriamente esa comunicación (por ejemplo, para programas como las herramientas mencionadas). Por otro lado, se hace posible la presentación y uso del material de un modo dinámico, en el lugar y momento elegido por el usuario, sin necesidad de depender de otros aspectos (como la existencia de un laboratorio específico).

* Segundo, resultar de utilidad a los estudiantes y profesionales que trabajan con dichas herramientas de generación de compiladores. No deseamos construir un sustituto para las ya existentes, sino un complemento para las mismas. Por ejemplo, estas herramientas pueden verse como metalenguajes, pero su uso es similar al de un lenguaje de programación: se ha de editar un programa, luego se compila, se evalúa, etc., con lo que se aplica prácticamente todo el ciclo de la programación. Sin embargo, no existen «entornos integrados» orientados al desarrollo, como se encuentran actualmente para la mayoría de lenguajes y plataformas. En ese sentido, entendemos el sistema como una especie de entorno de tal clase, que permita la rápida solución de problemas ofreciendo el máximo de información, y cuyo paso final sea la aplicación de la herramienta clásica, pero con la seguridad de la corrección del resultado obtenido. Y una vez más, al estar construido sobre el sistema telaraña asegurará una disponibilidad prácticamente universal.

La faceta elegida para comprobar la factibilidad del proyecto y estudiar cuáles han de ser las características del sistema a construir ha sido la implementación del algoritmo LR(1). Quizá, para los profanos, el tema a partir de aquí se vuelva especializado en demasía (si no lo ha sido ya, lo cual lamentaríamos), pero intentaremos describir someramente las ideas principales en la siguiente sección de este artículo. Después se darán unas explicaciones sobre los detalles de la implementación, y finalmente, las direcciones en que puede utilizarse.

2. EL ALGORITMO LR(1)

Una de las fases fundamentales de un compilador o traductor es el *análisis sintáctico:* el compilador ha de construir el árbol que refleja la estructura jerárquica de los diferentes elementos del programa fuente. El formalismo matemático que describe la sintaxis se llama *gramática independiente del contexto* (a partir de los estudios de Chomsky; ver [1]), que básicamente consta de *reglas sintácticas.* Una de las técnicas principales de diseño de un analizador sintáctico (o en inglés, *parser*) se conoce como *técnica ascendente*, porque construye el árbol desde las hojas (la entrada o programa fuente) hacia la raíz (el objetivo). Los detalles se encuentran en cualquier texto de compiladores, entre los que citamos [1], [5] y [6].

En la técnica ascendente, se va leyendo el programa o texto fuente símbolo a símbolo (operación que se denomina *desplazamiento*), y se van tratando de aplicar las reglas sintácticas según es posible (tarea que se llama *reducción*). El analizador sintáctico tiene que disponer del «conocimiento» necesario para poder decidir en cada momento cuándo debe realizarse un desplazamiento y cuándo una reducción. Este conocimiento es en la práctica una tabla, denominada *tabla de acciones*, que recoge todas las situaciones posibles según el *estado* de análisis sintáctico y el símbolo de la entrada.

Existen diversos algoritmos para el cálculo de tal tabla a partir de la gramática que describe la estructura sintáctica de un cierto lenguaje. Históricamente, el primero es justamente el algoritmo LR(1), obtenido por Knuth (1965) [8]. A su vez, DeRemer (1971) [2] propuso variaciones de este algoritmo, llamadas SLR(1) y LALR(1), pero con pasos realmente muy similares (esto es, una vez implementado uno de ellos, la consecución de los demás resulta más sencilla). Se considera que es el algoritmo LALR(1) el más adecuado, pues lleva a unas tablas que pueden ser significativamente menores que con la técnica LR(1), aunque también es cierto que esporádicamente puede conducir a problemas que con LR(1) no se producen. Por ser el que más cálculo precisa de los tres mencionados (y debido a que nuestro objetivo principal es evaluar, como se ha dicho, el interés y factibilidad del proyecto), se ha elegido el algoritmo original de Knuth.

Por supuesto, el hecho de que la tabla de acciones pueda obtenerse algorítmicamente a partir de la gramática es el que posibilita la existencia de las herramientas de generación de compiladores mencionadas en la introducción y también de nuestro sistema, obviamente. Las más utilizadas son `yacc` (Johnson 1975) [7] y `bison` (Donelly y Stallman 1995) [4]. Por otro lado, es solo gracias a ellas que pueden obtenerse los analizadores sintácticos descritos: téngase en cuenta que una tabla de acciones para un lenguaje como Pascal puede llegar a tener del orden de 300

estados distintos (según las técnicas óptimas a este respecto), siendo el número de símbolos de alrededor de 30. Ni que decir tiene que es imposible en la práctica llevar a cabo la construcción de la tabla «con lápiz y papel», por mucho que el algoritmo no sea, incluso podríamos decir, demasiado complejo.

3. UN ENTORNO DE DESARROLLO PARA LAS HERRAMIENTAS DE GENERACIÓN DE COMPILADORES

Un usuario de herramientas como `yacc o bison` escribe un «programa», que no es otra cosa que la gramática que describe el lenguaje a procesar por el futuro compilador o traductor. Como salida, obtendrá el analizador sintáctico: en una herramienta clásica como las mencionadas, la forma de la salida es un programa escrito en C (ambas utilidades se originaron alrededor del sistema UNIX), y esa es precisamente una de sus mayores ventajas. En efecto, un compilador o traductor, al fin y al cabo, debe ser también un programa, que alguien ha de escribir. Pues bien, las utilidades mencionadas no solo permiten calcular propiamente las tablas que gobernarán el análisis sintáctico, sino que les resulta igual de sencillo crear estructuras de datos para dichas tablas y generar los programas (hemos dicho C como podríamos haber dicho cualquier otro lenguaje, lo que ocurre que la mayoría de versiones usa tal lenguaje) que las manejan y realizan en definitiva el análisis sintáctico.

En nuestro sistema, pese a lo dicho, de momento nos interesa más la faceta de obtención de las tablas que su implementación final en este o aquel lenguaje (para ello, confíamos plenamente en `yacc o bison`, que serían el punto final de uso de nuestra herramienta). Pretendemos ofrecer toda la información posible al usuario, en su forma matemática o formal, para que pueda examinarla de un modo más adecuado, sacando partido de las posibilidades de visualización existentes en la actualidad. Entonces, podrá detectar posibles puntos de mejora, resolver situaciones problemáticas (el algoritmo puede aplicarse a cualquier gramática independiente del contexto, pero no siempre se obtiene una solución determinista: en tal caso es cuando el diseñador del compilador ha de echar mano de todo su conocimiento y experiencia para intentar resolver el problema sin modificar la gramática, tarea para la que, de momento, no parece existir un algoritmo, dicho sea más bien jocosamente), etc. En definitiva, nuestra pretensión es aplicar los modernos sistemas de comunicación entre la persona y la máquina (para los cuales el sistema telaraña resulta ser una infraestructura inmejorable de puesta en práctica), con la creencia de que redundará en una mayor facilidad y seguridad en la solución de la clase de problemas expuestos.

Después de haber construido un primer (aunque probablemente demasiado simple) boceto de cómo ha de ser una utilidad como la que proponemos (que se describe en la siguiente sección), nos encontramos en estos momentos en la fase de especificación definitiva de la misma. La experiencia nos ha servido principalmente para entrar en contacto con los detalles de uso de la plataforma telaraña (limitaciones del lenguaje HTML y de la interfaz de usuario, adecuación de Java para la programación, etc.), detalles que son lógicamente de gran importancia para la resolución práctica de las especificaciones que se propongan.

4. IMPLEMENTACIÓN DEL ALGORITMO LR(1)

Si en determinada ocasión el lector (por razones de estudio o de ampliación de conocimientos) se ha visto involucrado en la generación «manual» de la tabla de acciones producida por el algoritmo LR(1), es casi seguro que a su finalización haya percibido cierta sensación de inseguridad en el resultado obtenido: el algoritmo es más bien simple, pero también es muy fácil equivocarse en algún símbolo. Afortunadamente la próxima vez que quiera analizar la tabla que genera su gramática favorita ya no deberá coger un lápiz y uno (o muchos) papeles sino que abrirá su paginador favorito para examinar:

http://www-eleves.enst-bretagne.fr/~jldiez/lr1-1.html ó
http://www.deusto.es/~josuka/lr1-1.html

La idea es sencilla: crear una base de clases Java para poder producir un complejo sistema de ayuda a la generación de compiladores. El algoritmo LR(1) ha sido elegido como punto de partida, no por la complejidad de su aplicación, sino debido a la gran cantidad de información que produce (y maneja internamente). JavaParser está listo para generar la colección LR(1) para cualquier gramática introducida por el usuario, concentrándose en dos aspectos fundamentales: realizar un informe detallado del resultado del algoritmo, y por otro lado detectar e informar de todos los errores que se produzcan al definir la gramática por parte del usuario.

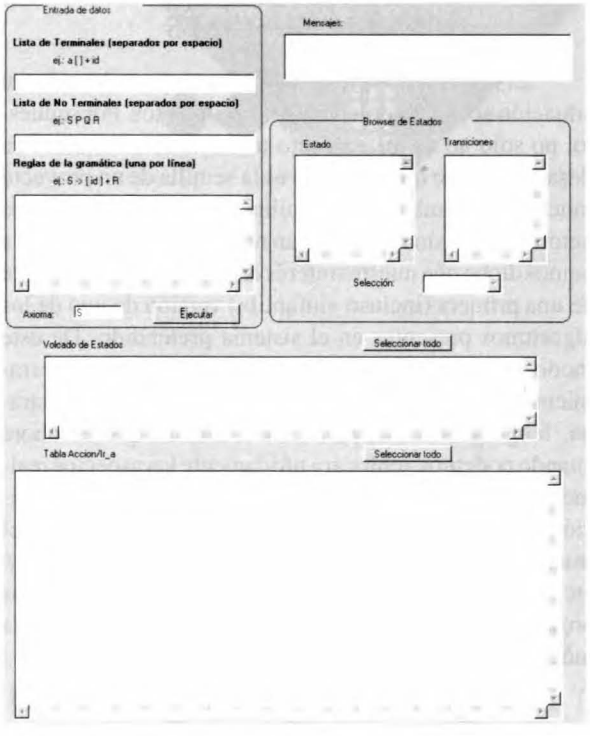
Existen diversos algoritmos para el cálculo de tal tabla a partir de la gramática que describe la estructura sintáctica de un cierto lenguaje. Históricamente, el primero es justamente el algoritmo LR(1)

A la pregunta ¿por qué Java? se nos ocurren dos respuestas. La primera es clara: no puede haber forma más cómoda de ejecutar esta aplicación que mediante una página telaraña (la alternativa de los programas CGI no

tiene la universalidad que proporciona Java en estos momentos); cualquier alumno, por ejemplo, podrá tener disponible el generador para consultar cualquier gramática en cualquier momento. Por otro lado, al ser Java un lenguaje orientado a objetos puro, se nos antoja como la plataforma ideal de programación donde diseñar objetos que se asemejen a los que existen en la misma Teoría de Compiladores.

4.1 El interface

El *applet* (o subprograma Java) ante el que nos encontramos presenta tres zonas muy determinadas. Su interfaz presenta el siguiente aspecto:



La zona de «Entrada de datos» dispone de cuatro áreas de texto para introducir la definición de reglas. El botón «Ejecutar» provoca la ejecución del algoritmo, que proporciona diversa información en otras zonas. La de nombre «Mensajes» nos dará información detallada de cómo se han realizado las fases de la ejecución del método LR(1). Se nos informará de todos los errores que haya cometido el usuario, así como si ha sido correcta la introducción de los datos, la generación de la colección y la tabla.

El resultado que propiamente interesa al diseñador del analizador sintáctico será visualizado de diversas formas para que el usuario lo pueda consultar de la manera que prefiera. Por un lado está el denominado «Browser de estados» con el que se podrán consultar de manera cómoda todos los estados generados, y las relaciones que hay entre los mismos (las llamadas «Transiciones», que conforman realmente una tabla que debe incluirse en el *parser* final). Por otro lado, las zonas de «Volcado»: una de estados (donde en realidad se recoge toda la información del browser de estados, en forma continua), y otra con la «Tabla de acciones y transiciones», que es el «motor» verdadero del *parser* que se intenta conseguir (objetivo último del algoritmo, en definitiva). Estas zonas están especialmente diseñadas para poder ser cortadas y pegadas en un editor de textos convencional.

4.2 Introduciendo una gramática

Para introducir una gramática correctamente hay que rellenar las cuatro zonas dispuestas para ello:

Zona de entrada de Terminales: en esta zona se deben introducir todos los terminales que la gramática use para que el programa los identifique como tales. El modo de inserción de esta lista es introduciendo toda la lista de no terminales *separadas por al menos un espacio*. Posteriormente el analizador se encargará de desechar (generando un aviso) aquellos caracteres no permitidos o incluso detectar si se ha introducido por error varias veces el mismo terminal. No se considerará error la introducción de los caracteres «#» (que representa la llamada cadena vacía) o «$» (que se usa internamente para el concepto de «*fin de programa*»), pero se aconseja que no se introduzcan debido a que la gramática subsiguiente carecería del sentido pretendido; tales símbolos deben tomarse entonces como una especie de «palabras reservadas».

Ejemplo: int id - + [ ]

Zona de entrada de No Terminales: en esta zona se deben introducir de forma análoga a la anterior la lista de No Terminales de la gramática. También se desecharán aquellos símbolos insertados varias veces pero se producirá un error si introducimos un símbolo que estaba definido previamente como Terminal.

Ejemplo: S E G Variable

Zona de entrada de Reglas: esta es sin duda la parte más crítica de la introducción de datos. Se deberá introducir cada regla de la gramática en una línea independiente (de momento, no se permite dar varias partes derechas para una misma parte izquierda, como en BNF). En cuanto al formato, es de la forma:

E -> E + R

Además debemos tener en cuenta que todos los símbolos deben estar separados por espacios y por supuesto, que dichos símbolos deben haber sido definidos previamente o el analizador se quejará. Pero no todo es malo; podemos al igual que antes dejar cualquier número de espacios entre símbolos e incluso dejar líneas en blanco entre reglas. El analizador, por otra parte, numerará las reglas *exactamente* en el mismo orden en el que están insertadas. Por último, el analizador se encargará de evitar errores y la inserción de una misma regla repetida hará que este la pase por alto.

Ejemplos:

S -> R = R
R -> a
R -> a [R]
R -> # (para la regla vacía)

Zona de entrada del axioma: en el cuadro dispuesto para esto se debe introducir el axioma, que por supuesto, habrá tenido que ser declarado como No Terminal previamente. El nombre de axioma por defecto es S.

4.3 Analizando la gramática y detectando errores

El proceso de lectura de gramáticas posee un mini-analizador léxico y sintáctico que se encarga de cuatro aspectos fundamentalmente:

1. Definir claramente la manera de introducir la gramática.
2. Evitar que el usuario cometa errores en la introducción de la misma
3. Informar de todos los errores en la gramática descrita.
4. Corregir inteligentemente los errores no fatales.

Como ya hemos visto previamente, el analizador genera avisos o errores fatales dependiendo del error en la inserción de la gramática por parte del usuario. Una vez que se acepte la gramática se enumeran los estados por el orden de introducción.

El proceso de análisis desemboca en el informe del resultado del proceso dentro de la ventana de mensajes del interface. Si el analizador no encuentra errores fatales en la inserción de la gramática se nos informará dentro del cuadro de mensajes y se procederá en este instante a la ejecución del método con los datos insertados. Si por el contrario se han encontrado errores de este tipo, el analizador informará de los mismos y detendrá su proceso en la línea que los haya producido.

4.4 Ejecución del algoritmo y salida de resultados

Si la gramática que se ha introducido es correcta, el programa generará la colección LR(1) que ella produce. La salida se centra en dos partes:

Browser de Estados. Mediante esta parte del interface podremos navegar por todos los estados que el algoritmo LR(1) haya generado. Para ello debemos elegir el estado que queremos visualizar en la casilla de selección. Podemos elegir un estado de los que se nos muestran al desplegar dicha casilla o si lo preferimos podemos utilizar los cursores para ir visitando los estados secuencialmente.

En la parte izquierda del browser podremos ver los elementos con el «punto» que representa la situación de análisis, y en la parte derecha veremos las transiciones que desde dicho estado salen hacia otro estado generado.

Zona de volcados. El programa genera dos volcados completos con toda la información de la colección generada. Estos dos cuadros poseen la cualidad de poderse *cortar y pegar* en cualquier editor de textos como el propio bloc de notas de Windows; para ello, solo ha de pulsarse el botón que aparece junto a las ventanas, que selecciona todo el texto que contengan. Esta cualidad lo hacen especialmente atractivos para poder llevar los resultados a otro documento y poderlos archivar o imprimir, no teniendo que visitar la página telaraña cada vez que queramos hacer una consulta para la misma gramática.

5. CONCLUSIONES

El sistema que presentamos es muy modesto en su situación actual, y adolece de graves defectos. Por supuesto, no solo no es un producto terminado, ni siquiera en desarrollo, sino que más bien es la semilla de un proyecto mucho más ambicioso, semilla en la que a propósito hemos dejado sin resolver gran número de cuestiones. Ya hemos dicho que nuestro interés era disponer rápidamente de una primera (incluso «infantil») versión de uno de los algoritmos presentes en el sistema pretendido. De este modo, tenemos más claras las ideas acerca de las herramientas con que vamos a trabajar (esto es, páginas telaraña, lenguaje Java, *applets*, etc.), con lo que es ahora cuando podemos «enfocar» nítidamente los aspectos realmente de interés en el problema: las técnicas de compilación, la especificación de la interface, la interacción con el ususario, el comportamiento temporal y espacial del *applet*, etc. Esperamos que nuestro trabajo fructifique en los próximos meses y podamos ofreceros una herramienta mucho más completa.

6. BIBLIOGRAFÍA

[1] AHO, A.V.; SETHI, R.; ULLMAN, J.D. [1986] Compilers: Principles, Techniques, and Tools, Adisson-Wesley, Reading, MA.

[2] DEREMER, F. [1971] «Simple LR(k) grammars», Communications of the ACM, 14, 7, 453-460.

[3] DÍAZ, JOSUKA [1994] Teoría de autómatas y lenguajes formales, Universidad de Deusto.

[4] DONELLY, C.; STALLMAN, R. [1995] Bison: the YACC-compatible Parser Generator, Free Software Foundation, Boston, MA.

[5] EGUÍLUZ, ANDONI; DÍAZ, JOSUKA [1995] Compiladores, Universidad de Deusto.

[6] FISCHER, C.N.; LEBLANC, R.J. [1988] Crafting a Compiler, Benjamin/Cummings, Redwood City, CA.

[7] JOHNSON, S.C. [1975] «Yacc - yet another compiler compiler», Comp. Sci. Tech. Report 32, AT&T Bell Lab., Murray Hill, NJ.

[8] KNUTH, D. [1965] «On the translation of languages from left to right», Information and Control, 8, 6, 607-639.