Compilación eficiente de esquemas de análisis sintáctico*

Carlos Gómez Rodríguez Jesús Vilares Miguel A. Alonso

Depto. de Computación, Universidade da Coruña

Campus de Elviña, 5 - 15071 A Coruña

komoku@gmail.com jvilares@udc.es alonso@udc.es

Resumen

Presentamos un compilador capaz de generar analizadores sintácticos a partir de esquemas de análisis sintáctico. Dichos esquemas son representaciones de los analizadores en forma de sistemas deductivos, que abstraen los detalles de implementación y permiten definir y comparar fácilmente diferentes algoritmos. Nuestro compilador es capaz de generar, a partir de uno de estos esquemas, el código de una implementación del algoritmo correspondiente, incluyendo el código necesario para gestionar la indexación de resultados intermedios para que dicha implementación sea eficiente. El sistema acepta todo tipo de esquemas de analizadores para gramáticas independientes del contexto, y se puede generalizar fácilmente a otros tipos de formalismos gramaticales.

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 tanto en la compilación de programas como en el procesamiento del lenguaje natural. Desde los trabajos de Chomsky que formalizaron la noción de gramática independiente del contexto en los años 50, 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 [13], 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 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 nos permitirán generar nuevos ítems a partir de los iniciales, y cuáles serán los *ítems finales* que determinan que la frase ha sido analizada.

Así, por ejemplo, dada una gramática independiente del contexto $G = (N, \Sigma, P, S)$[1] en forma normal de Chomsky (es decir, con todas sus reglas de la forma $A \rightarrow BC$ y $A \rightarrow a$), y una oración de longitud n que

*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).

[1] Donde N denota el conjunto de símbolos no terminales, Σ el de símbolos terminales, P las producciones gramaticales y S el axioma.

denotamos $a_1\ a_2\ ...\ a_n$[2], un esquema para el algoritmo de análisis no determinista CYK [9, 14] sería el siguiente:

Conjunto de ítems:
$I = \{[A, i, j] \mid A \in N \wedge 0 \leq i < j\}$

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

Pasos deductivos:
$$D(1) : \frac{[a, i-1, i]}{[A, i-1, i]}\ A \to a \in P$$
$$D(2) : \frac{[B, i, j] \qquad [C, j, k]}{[A, i, k]}\ A \to BC \in P$$

Ítems finales:
$I_f = \{[S, 0, n]\}$

Una explicación detallada y formal de los esquemas de análisis sintáctico se puede encontrar en [13]. Aquí, por motivos de economía, nos limitaremos a una explicación informal basada en este ejemplo.

Los ítems en este algoritmo están compuestos por un símbolo, terminal o no, y dos números enteros que denotan posiciones de la cadena de entrada. El significado concreto de un ítem $[A, i, j]$ se puede interpretar como: "La gramática permite construir un árbol de raíz A que abarca la parte $a_{i+1} \ldots a_j$ de la cadena de entrada". De acuerdo con esta interpretación, los ítems iniciales del algoritmo denotan árboles triviales compuestos por un solo nodo, que es un símbolo terminal de la frase.

Los pasos deductivos $\frac{\eta_1 \ldots \eta_m}{\xi}\ \Phi$ 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 este ejemplo, los pasos deductivos $D(1)$ y $D(2)$ representan el proceso ascendente que lleva a cabo el algoritmo CYK. Así, el paso $D(1)$ permite aplicar una regla de la forma $A \to a$ para construir un árbol de dos nodos (una raíz no terminal y una hoja terminal). El paso $D(2)$ permite aplicar las reglas binarias de la forma $A \to BC$, uniendo dos árboles de raíces B y C en un nuevo árbol cuya raíz es A, y que abarca una porción de la frase unión de las dos anteriores. Si la frase pertenece al lenguaje generado por la gramática, mediante alguna secuencia de estas operaciones será posible generar el ítem $[S, 0, n]$, que denota la existencia de un árbol que abarca toda la frase y cuya raíz es el axioma de la gramática.

[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

Como se puede apreciar, aunque un esquema de análisis sintáctico como éste permite describir fácilmente un algoritmo, no es un algoritmo en sí mismo. El esquema especifica los pasos que permiten obtener el análisis sintáctico; pero no el orden en que se deben ejecutar, ni las estructuras de datos a utilizar para almacenar los ítems. Diferentes decisiones de diseño en lo que respecta a estos dos puntos darán lugar a distintos algoritmos que implementan el mismo esquema. Así pues, los esquemas de análisis sintáctico constituyen un nivel de abstracción superior al de los algoritmos correspondientes, y esta abstracción resulta útil para describir, diseñar, comprender y comparar entre sí los algoritmos.

Sin embargo, esa misma abstracción hace que los esquemas no se puedan ejecutar directamente sobre una máquina. Un sistema capaz de generar automáticamente una implementación algorítmica de un esquema de análisis sintáctico sería, pues, muy útil, al permitir comprobar inmediatamente en el computador el funcionamiento de los esquemas, facilitando el diseño y prototipado rápido de algoritmos.

El sistema que se describe en este trabajo cubre esta necesidad, pues permite compilar esquemas de análisis sintáctico, generando una implementación del analizador correspondiente al esquema en lenguaje Java.

La idea de proporcionar maneras más o menos directas de obtener implementaciones ejecutables a partir de esquemas de análisis u otros formalismos similares no es nueva. En [12] se propone una implementación en Prolog de un motor deductivo para el análisis sintáctico que se basa en un formalismo de deducción conceptualmente análogo a los esquemas de Sikkel. Esta implementación se compone de una serie de predicados de Prolog comunes que implementan el mecanismo general de deducción y otros que se deben crear específicamente para cada algoritmo de análisis, y que vienen a ser la transformación de los pasos de los esquemas en reglas Prolog, además de algún predicado auxiliar para manejar aspectos como la indexación de ítems.

Si bien esta implementación es interesante, tiene varios inconvenientes:

1. No es todo lo declarativa que cabría esperar, pues es preciso codificar los pasos deductivos en lenguaje Prolog.

2. Es ineficiente, ya que la capacidad de indexación de ítems es muy limitada, como queda patente en la sección de resultados experimentales de [2].

Nuestro sistema parte de una filosofía diferente: en lugar de ejecutar directamente el esquema previamente codificado en un lenguaje como Prolog, partimos de un esquema representado en una notación sencilla que el sistema compila, generando código en Java, que tras ser a su vez compilado permite obtener los *bytecodes* Java que ejecutan el algoritmo, incluyendo también aspectos como la indexación de los ítems (que en el enfoque de [12] debía ser programada por el usuario). Por ejemplo, el fichero de entrada para el esquema CYK sería:

```
@step D1
[ B , i , j ]
[ C , j , k ]
-------------------- A -> B C
[ A , i , k ]
@step D2
[ a , i , i+1 ]
-------------------- A -> a
[ A , i , i+1 ]
```

La filosofía y el modo de empleo del sistema son, por lo tanto, similares a los de compiladores de compiladores como yacc [7], bison [3] o javacc [6].

3. Esquema general del compilador

Nuestro compilador transformará los esquemas de análisis en una implementación ejecutable manteniendo la estructura subyacente al esquema. Para ello, generará código en un lenguaje orientado a objetos (Java), y dicho código será el resultado de modelar como objetos los componentes del esquema, teniendo en cuenta los siguientes principios:

• Cada uno de los pasos deductivos que componen el esquema dará lugar a una clase.

• Se creará un objeto de dicha clase por cada posible manera de satisfacer las condiciones laterales.

3.1. Aplicación de los pasos deductivos

La clase asociada a cada paso deductivo tendrá un método `paso` para tratar de aplicar dicho paso a un ítem determinado.

Si el antecedente del paso deductivo especifica un único ítem, entonces el método generado comprobará si hay alguna manera de instanciar las variables del antecedente dándoles valores que lo hagan igual al ítem. En tal caso, se genera un ítem consecuente que resulta de llevar a cabo esas mismas instanciaciones en las variables del consecuente, y que es el resultado que retorna el método.

Si el antecedente del paso especifica m ítems $(A_1, A_2, ..., A_m)$ entonces el método generado se implementa mediante el seudocódigo de la figura 1. Así pues, cuando tenemos m especificaciones de ítems en el antecedente, el ítem dado como parámetro se debe intentar emparejar con cada una de ellas tal y como se describió en el caso de antecedente simple. Para cada emparejamiento que tenga éxito hay que llevar a cabo una búsqueda de ítems que se correspondan con los restantes $m-1$ elementos. Dichas búsquedas dan lugar a un anidamiento de $m-1$ bucles, pues cada elemento del antecedente puede ajustarse a varios ítems, y cada uno de ellos implicará una instanciación distinta de las variables, condicionando las bús-

quedas posteriores. Esto implica que tras cada búsqueda de ítems que se ajusten a una especificación dada habrá que iniciar un nuevo bucle recorriendo los ítems encontrados, instanciando las variables a los valores obtenidos de esos ítems y llevando a cabo la siguiente búsqueda para cada una de esas instanciaciones. Si llegamos al bucle más interno, habremos encontrado ítems que emparejan con todas las especificaciones del antecedente, y todas las variables del paso estarán instanciadas a valores concretos. El ítem obtenido como resultado será entonces el consecuente con esos valores en sus variables. El resultado final de este proceso es independiente del orden en que se aniden los $m-1$ bucles, dado que lo que obtenemos es un conjunto de instanciaciones de las variables del antecedente que cumplen ciertas condiciones, y el orden de los bucles sólo afecta al orden en que se instancian las variables.

Las instanciaciones de variables que se llevan a cabo en el método `paso` se hacen en atributos del objeto paso deductivo. Se generará uno de estos atributos por cada elemento atómico del paso que no sea constante (por ejemplo, A, B, C, i, j y k en el caso del algoritmo CYK). Los atributos tomarán inicialmente valor nulo, al estar las variables sin instanciar. El constructor del paso deductivo inicializará aquéllos que se deriven de las condiciones laterales, y los restantes serán instanciados durante la ejecución del método `paso`, como hemos visto en el seudocódigo. Estas últimas instanciaciones se desharán cada vez que dicho método genere resultados, para poder aplicar el paso a nuevos ítems.[3]

3.2. La máquina deductiva de análisis sintáctico

Además de las clases que representan los pasos deductivos, hará falta un código que coordine la ejecución de todos ellos, proporcionando un método que los utilice para implementar completamente el algoritmo de análisis sintáctico correspondiente al esquema. Este código es la implementación de una "máquina deductiva de análisis sintáctico", y es un algoritmo genérico que no depende del esquema concreto que se esté compilando; sino que puede trabajar con cualquier conjunto de pasos deductivos instanciados de acuerdo con los principios mencionados anteriormente:

```
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(ítem vacío);
   ítems.añadir(resultados);
   agenda.añadirAlFinal(resultados);
   pasos.quitar(p);
}
Mientras agenda no vacía {
   ÍtemActual = agenda.quitarPrimero();
   Para cada paso deductivo aplicable a
            ÍtemActual (p) en pasos {
      resultados = p.paso(ÍtemActual);
      ítems.añadir(resultados);
      agenda.añadirAlFinal(resultados);
   }
}
Devolver ítems;
```

El funcionamiento de este algoritmo se basa en conocer en todo momento los ítems disponibles, ya sea como hipótesis iniciales del algoritmo de análisis sintáctico o como deducciones obtenidas tras la ejecución de pasos. Asimismo, es necesaria una *agenda* (implementada como una cola) para almacenar los ítems mientras no los hayamos utilizado para intentar ejecutar pasos deductivos.

La corrección y completitud del algoritmo se puede demostrar fácilmente por inducción.

4. Indexación

El algoritmo anterior, junto con el código asociado a los pasos deductivos, es suficiente para analizar frases de acuerdo con un esquema de análisis sintáctico; pero sólo será eficiente si el acceso a los ítems y a los pasos deductivos lo es. Cuando ejecutamos el código correspondiente a un paso, a menudo tenemos que buscar en el conjunto de ítems todos aquéllos que cumplan una especificación dada.

[3] En el seudocódigo se ha obviado la operación de deshacer instanciaciones por motivos de claridad; pero dicha operación se ubicaría, para cada instanciación, inmediatamente antes de la llave que cierra el bloque de código en que se encuentra.

```
paso(item) {
  resultado = [];
  for i=1..m {
    if (item empareja con A_i) {
      instanciar variables de A_i haciendo A_i=item;
      items_1 = buscar items que emparejan con A_1;
      para cada item en items_1 {
        instanciar variables de A_1 haciendo A_1=item;
        items_2 = buscar items que emparejan con A_2;
        para cada item en items_2 { //(...)
            instanciar variables de A_{i-2} haciendo A_{i-2}=item;
            items_{i-1} = buscar items que emparejan con A_{i-1};
            para cada item en items_{i-1} {
              instanciar variables de A_{i-1} haciendo A_{i-1}=item;
              items_{i+1} = buscar items que emparejan con A_{i+1};
              para cada item en items_{i+1} { //(...)
                  instanciar variables de A_{m-1} haciendo A_{m-1}=item;
                  items_m = buscar items que emparejan con A_m;
                  para cada item en items_m {
                    instanciar variables de A_m haciendo A_m=item;
                    resultado = resultado + estado actual del consecuente;
                  } //para cada item en items_m
              (...) //cierres de llaves
      } //para cada item en items(1)
    } //if
  } //for i=1..m
  return resultado;
} //paso
```

Figura 1: Código asociado a un paso deductivo de antecedente compuesto.

Estudiando casos como los algoritmos CYK o Earley [4], podemos comprobar que una implementación ineficiente de esta búsqueda (por ejemplo, ir tomando secuencialmente los ítems del conjunto y comprobar si cumplen la especificación) daría lugar a una implementación del algoritmo de análisis con una complejidad computacional más alta que la establecida teóricamente. Algo similar sucede con la operación que determina si un ítem ya existe en un conjunto, que no se menciona explícitamente en el seudocódigo pero deberá usarse cada vez que añadamos un ítem para evitar duplicados.

Por ejemplo, en el caso del algoritmo CYK, el paso deductivo $D(2)$ se ejecuta $O(n^3)$ veces para generar ítems (contando la generación de ítems repetidos), siendo n la longitud de la frase. Este paso, según lo que se explicó con anterioridad, dará lugar a este código:

```
paso(ítem) {
  result=[]
  if ( ∃ _i,_j | ítem = [B,_i,_j] ) {
    i=_i; j=_j;
    lista1 = buscarÍtems( [C,j,*] );
    para cada ítem [C,j,_k] en lista1 {
      k=_k;
      result = result ∪ [A,i,k];
    }
  }
  if ( ∃ _j,_k | ítem = [C,_j,_k] ) {
    j=_j; k=_k;
    lista2 = buscarÍtems( [B,*,j] );
    para cada ítem [B,_i,j] en lista2 {
      i=_i;
      result = result ∪ [A,i,k];
    }
  }
  devolver result;
}
```

Si suponemos que el método `buscarÍtems` y la comprobación de elementos repetidos que se producirá cuando añadamos los resultados al conjunto de ítems se ejecutan en tiempo constante, entonces cada ejecución del método `paso` es $O(n)$ y genera $O(n)$ ítems. Por lo tanto, la generación de un ítem (o ejecución elemental de un paso, que no coincide con la ejecución del método anterior que puede generar varios ítems) es en tiempo amortizado constante. Sin embargo, si el método `buscarÍtems` fuese una búsqueda secuencial, y teniendo en cuenta que la cantidad de ítems presentes en el conjunto para este algoritmo es $O(n^2)$, el método `paso` sería a su vez $O(n^2)$, con lo cual la implementación generada para el algoritmo sería $O(n^4)$.

Esto nos lleva a la necesidad de que las operaciones de acceso a ítems se lleven a cabo siempre en tiempo constante si queremos mantener la complejidad teórica de los algoritmos en las implementaciones que generemos. Para ello, además de las clases ya mencionadas, generaremos una clase que se encargará de gestionar los ítems, indexándolos de manera eficiente.

4.1. Gestión de los ítems

La generación de esta clase no es una tarea trivial pues, al contrario que en el caso de la máquina deductiva, un mismo código no podrá servir para cualquier algoritmo de análisis sintáctico. Por ejemplo, en el seudocódigo anterior podemos ver que para la implementación del CYK es necesario llevar a cabo dos tipos diferentes de búsquedas de ítems: una que busca los ítems con primera y segunda componente dadas (ítems de la forma $[C, j, *]$, donde $*$ podría tomar cualquier valor) y otra que busca los ítems con primera y tercera componente dadas (ítems de la forma $[B, *, j]$). Así pues, para conseguir que la búsqueda de un ítem se ejecute en tiempo constante, necesitaremos tener al menos dos índices, uno según la primera y segunda componente y otro según la primera y tercera. En la práctica, un solo índice por la primera componente basta para obtener resultados aceptables; pero en el caso general no será así.

En cada esquema de análisis sintáctico será necesario una indexación diferente, y lo normal en esquemas más complejos es que los tipos de búsqueda necesarios difieran tanto entre sí que hagan necesario tener los ítems indexados de varias maneras distintas. Este control de la indexación no se puede conseguir fácilmente desde la perspectiva de la programación lógica [12]; pero sí es factible en nuestro compilador.

Dado que nos interesa contar con un esquema de indexación genérico y adaptable a cualquier algoritmo, el interfaz de la clase responsable de la gestión de ítems será también genérico. La clase tendrá que proporcionar principalmente tres métodos: agregar un ítem al conjunto, introduciéndolo en los índices donde sea necesario, decir si un ítem determinado existe o no, y devolver todos los ítems que cumplan ciertas características. Tanto el segundo como el tercer método necesitarán técnicas de indexación para funcionar de manera eficiente.

Para llamar al tercer método necesitaremos una manera de especificar restricciones sobre ítems: a la hora de hacer búsquedas, necesitaremos pasarle un objeto de algún tipo que permita representar conceptos como "los ítems de la forma $[C, j, *]$", o "los ítems del algoritmo de Earley que tengan un símbolo S después del punto". Por simplicidad y eficiencia, se ha optado por representar estos criterios de búsqueda igual que los ítems normales; pero usando nulos para representar los elementos no restringidos. Así, para preguntar por los ítems de la forma $[A, i, *]$, el algoritmo CYK pasará a la clase gestora de ítems la especificación $[A, i, null]$, y dicha clase deberá devolver los ítems correspondientes, para lo cual utilizará el índice más adecuado de entre los que disponga, o una búsqueda secuencial si no hay ningún índice adecuado a la especificación.

4.2. Generación de índices para búsqueda

Para saber cuáles son los índices que conviene tener disponibles para un algoritmo dado, utilizamos una aproximación basada en recopilar información durante la generación del código de los pasos deductivos.

Como se vio anteriormente, el código del método que aplica un paso deductivo dado ha-

ce y deshace instanciaciones de variables, y llama a la operación de búsqueda de ítems. Cada llamada a la operación de búsqueda se hace para encontrar los ítems que se correspondan con una de las especificaciones del antecedente, partiendo de las instanciaciones que ya se han hecho y que, como comentábamos, se guardan en atributos del paso deductivo.

Si bien en el momento en que generamos el código nos es imposible saber a qué valores van a estar instanciadas las variables cuando se ejecute una búsqueda, pues dichos valores cambiarán en cada ejecución concreta del paso; sí que tenemos información suficiente para saber con certeza qué variables van a estar instanciadas y cuáles no en cada llamada a la búsqueda: estarán instanciadas sólo aquellas variables del paso que aparezcan en las condiciones laterales o en los elementos del antecedente que ya se han considerado en el momento en que se llama a la búsqueda. Así, por ejemplo, en el código asociado al paso deductivo

$$D(2) : \frac{A1 : [B, i, j] \qquad A2 : [C, j, k]}{B : [A, i, k]} \; A \rightarrow BC$$

sabemos que habrá dos operaciones de búsqueda:

• La búsqueda de ítems que se ajusten a A2 cuando el ítem con el que se ha ejecutado el paso se ha emparejado con A1. En este caso, estarán instanciadas A, B y C (por el constructor) e i, j (por el emparejamiento con A1). Por lo tanto, esta búsqueda preguntará por los ítems de la forma $[C, j, null]$, dado que k es la única variable no instanciada que aparece en A2.

• La búsqueda de ítems que se ajusten a A1 cuando el ítem con el que se ha ejecutado el paso se ha emparejado con A2. En este caso, estarán instanciadas A, B y C (por el constructor) y j, k (por el emparejamiento con A2). Por lo tanto, esta búsqueda preguntará por los ítems de la forma $[C, null, j]$, dado que i es la única variable no instanciada que aparece en A1.

Durante la generación del código del método `paso`, se llevará cuenta de qué variables están instanciadas en cada momento y cuáles no, y al generar una llamada a una operación de búsqueda se almacenará un descriptor del tipo de búsqueda que se ha llevado a cabo. Este descriptor reflejará la estructura de los ítems que se buscan, y las clases de los elementos instanciados que aparecen en la búsqueda. Por ejemplo, los descriptores de las dos búsquedas anteriormente descritas serían $[Symbol, Position, null]$ y $[Symbol, null, Position]$. El primero de ellos se podría leer como: "búsqueda de todos los ítems compuestos por tres elementos donde el primero es un símbolo dado y el segundo es una posición dada de la cadena de entrada". El descriptor no contiene información sobre la clase del tercer elemento, aunque al generar la búsqueda sabemos que nos interesa que sea una posición de cadena de entrada, porque esta información no es relevante de cara a la construcción de un índice.

Una vez que tengamos los descriptores de todas las búsquedas que realizan los pasos deductivos del algoritmo, podremos tomar la decisión sobre qué índices crear. Nos interesará generar índices sobre aquellos componentes de los descriptores de búsqueda que no sean nulos y que se presten a indexación. La manera más sencilla es crear un índice por cada descriptor de búsqueda, indexando por todos los componentes que cumplan estas condiciones: así, en este caso tendríamos un índice por la primera y la segunda componente y otro por la primera y tercera, que es lo que anteriormente comentamos que era lo óptimo para el algoritmo CYK. Con esta estrategia siempre obtendremos los índices necesarios para todas las búsquedas de ítems que se van a realizar; pero puede suceder que obtengamos índices redundantes. Así pues, tendremos que comparar los índices que vayamos obteniendo entre sí para eliminar duplicados.

La salida del método que toma la decisión sobre qué índices crear será un conjunto de especificaciones de índice, conteniendo toda la información necesaria para generar el código que manejará dichos índices. Dicha información consistirá en una lista con tantos elementos como componentes por las que se quiere indexar, de manera que en cada elemento se especificará la posición de la componente correspondiente y el tipo de indexación que

se utilizará para esa componente. La posición se especifica mediante un sencillo esquema de direccionamiento de los nodos del ítem, considerado como un árbol (en el algoritmo CYK los ítems son ternas de componentes atómicos; pero en otros las componentes pueden tener a su vez estructura). El tipo de índice se refiere a la estructura de datos que se utilizará, y el compilador escogerá una u otra dependiendo de las características del tipo de elemento por el que se indexa.

4.3. Generación de índices de existencia

Anteriormente mencionamos que además de los índices para búsquedas, en los que hemos centrado la explicación, también necesitaríamos utilizar indexación para el método que determina la existencia o no de un ítem en el conjunto de ítems. Estos índices se generan de manera análoga, para lo cual crearemos una especificación de búsqueda a partir del consecuente de cada paso deductivo. Análogamente al caso anterior, dicha especificación reflejará la estructura del consecuente y las clases de los elementos que aparecen en él y que están instanciados en el momento en que se lleva a cabo la llamada al método de existencia. En este caso todos los elementos del consecuente estarán instanciados, dado que para que un paso deductivo genere un ítem (situación en la cual es necesario comprobar su existencia en el conjunto de ítems) es necesario que se haya procesado todo el antecedente y todas las variables tengan un valor concreto. Por lo demás, la generación de estos índices de existencia seguirá el mismo camino que la generación de los índices para las búsquedas.

4.4. Generación de índices de pasos deductivos

Además de los índices sobre ítems ya comentados, se generará otro tipo de índices que contribuirán también a la eficiencia del algoritmo: son los índices de pasos deductivos, que utilizaremos para decidir rápidamente qué pasos resultarán útiles en cada momento del análisis.

La necesidad de este tipo de índice surge del hecho de que el código generado crea una instancia de cada clase *paso deductivo* por cada manera que haya de satisfacer las condiciones laterales. Las condiciones laterales de nuestros pasos son descripciones de reglas de la gramática, que pueden ser satisfechas por multitud de reglas: por ejemplo, la del paso $D(2)$ es válida para cualquier regla de la forma $A \rightarrow BC$, con lo cual se creará una instancia del paso por cada regla de esta forma que exista en la gramática.

La creación de todos esos pasos deductivos supone que en el bucle "Para cada paso deductivo aplicable a ÍtemActual (p) en pasos" (sección 3.2) tengamos que intentar aplicar muchos pasos, la mayoría de los cuales no producirán ningún resultado útil para el análisis al no tener éxito los emparejamientos con el antecedente que se hacen en el método `paso`. De cara a la eficiencia, es interesante reducir el conjunto de pasos que se intenta aplicar. Esto no nos permitirá reducir más la complejidad computacional del algoritmo en términos de tamaño de la cadena de entrada; pero sí reducirá el impacto del tamaño de la gramática en el rendimiento.

Para tratar de aproximar lo máximo posible nuestro conjunto de pasos deductivos aplicables a aquéllos que realmente van a formar parte del análisis, indexaremos los pasos deductivos en función de los elementos del antecedente que son constantes (están instanciados) tras la construcción de los pasos, es decir, aquellos elementos que aparecen en las condiciones laterales (o bien son directamente constantes en el propio paso) para cada una de las especificaciones de ítems del antecedente.

Así, por ejemplo, en el paso $D(2)$ el antecedente tiene dos especificaciones de ítems: $A1 = [B, i, j]$ y $A2 = [C, j, k]$. Considerando que sólo los símbolos B y C estarán instanciados tras la construcción de cada instancia del paso, procediendo de manera análoga a lo visto en los índices de búsqueda se crearán dos descriptores de búsqueda idénticos $[Symbol, null, null]$, dando lugar a un índice de los pasos deductivos por la primera componente de los ítems del antecedente. A la hora de introducir una instancia de un paso deduc-

tivo en el índice se tendrán en cuenta todas las especificaciones de ítems que aparezcan en el antecedente, es decir, la instancia concreta

$$\frac{[NP, i, j] \qquad [VP, j, k]}{[S, i, k]}$$

(que se ha instanciado a partir de la regla $S \rightarrow NP\ \ VP$) se introducirá en el índice dos veces, tomando como claves NP y VP, de manera que pueda ser localizada por cualquiera de ellos.

Este índice será el que se utilice cada vez que queremos saber qué pasos deductivos concretos (instancias) son aplicables a un ítem dado.

5. Otros elementos en los esquemas

Aparte de los elementos presentes en el esquema CYK tomado como ejemplo, en otros esquemas pueden aparecer otros tipos de elementos. Sin embargo, su tratamiento es siempre similar, ajustándose a alguno de estos cuatro tipos:

• *Simples*: son elementos que pueden estar instanciados o no, y cuando lo están toman un valor, posiblemente convertible a una clave para indexar. Ejemplos: símbolos gramaticales, posiciones de cadena de entrada, números enteros o probabilidades.

• *Expresiones*: representan expresiones sobre elementos simples, por ejemplo $i+1$ (sobre posiciones de cadena de entrada) o $tree[A, B]$ (sobre símbolos). Se considerarán constantes cuando todos los elementos raíz de la expresión lo sean, y en tal caso se tratarán de manera análoga a los elementos simples. Las expresiones tendrán asociada una expresión Java para obtener el resultado a partir de los operandos (por ejemplo, la suma de posiciones de la cadena de entrada estará asociada a una suma en Java).

• *Compuestos*: representan una secuencia de elementos de longitud finita y conocida. Los ítems del CYK son un elemento compuesto con tres componentes. Los ítems del algoritmo de Earley $[A \rightarrow \alpha.B\beta, i, j]$ son elementos compuestos cuya primera componente es un elemento compuesto. De cara a la indexación, como ya veíamos, considerábamos los ítems como estructuras arborescentes.

• *Secuencias*: representan una secuencia de elementos de un mismo tipo, de longitud finita pero desconocida mientras el elemento no esté instanciado. Por ejemplo, en el esquema para el algoritmo de Earley aparecen reglas como $A \rightarrow \alpha.B\beta$, y aquí α y β pueden corresponderse con cualquier número de símbolos. Este hecho deberá tenerse en cuenta a la hora de llevar a cabo los emparejamientos.

Todos los tipos de elementos concretos que manejará nuestro compilador deberán ser subclases de uno de estos cuatro tipos, y su manejo dependerá sólo de a cuál de ellos pertenecen, y no de su subclase concreta. Esto facilita la extensibilidad del sistema: dado que cada esquema de análisis sintáctico puede definir sus propios formatos de ítems, y los elementos notacionales de un esquema pueden hacer referencia a cualquier tipo de objeto matemático, no se puede esperar que el sistema sea capaz de compilar cualquier esquema posible. Optamos, pues, por proporcionar una serie de clases de elementos predeterminadas y dar la posibilidad al usuario de definir nuevos tipos de elemento mediante herencia, extendiendo uno de estos cuatro tipos. Si el usuario implementa correctamente el interfaz definido para uno de ellos y además define con una expresión regular la notación que se utilizará en el esquema para su nuevo tipo de elemento, el compilador será capaz de manejarlo sin problemas y generar el código adecuado. Esto proporciona un grado de extensibilidad muy grande, dado que lo normal es que cualquier tipo de elemento que aparezca en un esquema se pueda encuadrar de modo natural en uno de los tipos.

6. Conclusiones y trabajo futuro

En este trabajo se ha presentado un compilador que permite transformar esquemas de análisis sintáctico en una implementación, en forma de código Java directamente compilable, del algoritmo descrito por cada esquema.

El sistema implementado se ha probado con los esquemas correspondientes a los más conocidos algoritmos de análisis para gramáticas independientes del contexto, como CYK [9, 14], Earley [4] y Left-Corner [11], produ-

ciendo como salida implementaciones funcionales y eficientes de los mismos. En [5] se muestran resultados experimentales de todos ellos, incluyendo su aplicación a gramáticas cíclicas y ambiguas.

Este compilador proporciona la posibilidad de prototipar y validar rápidamente algoritmos de análisis sintáctico, dado que mediante un único comando podemos pasar de una representación del esquema prácticamente coincidente con la notación estándar a una implementación completa y autocontenida, directamente convertible en código ejecutable. Esto hace muy sencillo probar diferentes variantes de analizadores sintácticos y comprobar el efecto de los cambios, y proporciona un cómodo vínculo entre la teoría de los esquemas y su implementación.

Como trabajo futuro, queda pendiente generalizar el sistema a otros formalismos gramaticales más adecuados para el análisis de lenguajes de programación (como las gramáticas de atributos [10]) o lenguajes naturales (como las gramáticas de adjunción de árboles [8]); así como probarlo con algoritmos de análisis LR, que también se pueden expresar en términos de esquemas [1].

Referencias

[1] Miguel A. Alonso, David Cabrero, and Manuel Vilares. Construction of efficient generalized LR parsers. *Lecture Notes in Computer Science*, 1436:7–24. 1998.

[2] Miguel A. Alonso, and Víctor J. Díaz, Variants of mixed parsing of TAG and TIG, Traitement Automatique des Langues, 44(3):41-65, 2003.

[3] C Donnelly and R. M. Stallman. Bison, *Reference Manual, version 1.20*. Free Software Foundation, Inc., Cambridge, MA, USA, 1992.

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

[5] C. Gómez Rodríguez, J. Vilares, and M. A. Alonso. Generación automática de analizadores sintácticos a partir de esquemas de análisis. *Actas de SEPLN 2005*, Granada, España, 2005.

[6] JavaCC Project Home: `https://javacc.dev.java.net/`

[7] S. C. Johnson. YACC: Yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, USA, 1975.

[8] Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol 3: Beyond Words*, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York, 1997.

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

[10] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[11] D. J. Rosenkrantz and P. M. Lewis II. Deterministic Left Corner parsing. In *Conference Record of 1970 11th Annual Meeting on Switching and Automata Theory*, pages 139–152, Santa Monica, 1970.

[12] S. M. Shieber, Y. Schabes, and F. C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.

[13] Klaas Sikkel. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Springer-Verlag, Berlin/Heidelberg/New York, 1997.

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