Práctica 2:
Algoritmo de análisis sintáctico CYK

Grado en Ingeniería Informática

Teoría de la Computación

2017/2018

Dada una gramática independiente del contexto y dada una cadena de símbolos, a veces estaremos interesados en encontrar las estructuras sintácticas de dicha cadena de acuerdo con la gramática. Los algoritmos que realizan esta tarea se denominan analizadores sintácticos o *parsers*, y pueden clasificarse, básicamente, en dos grandes grupos:

- Los analizadores descendentes son aquéllos que parten del axioma de la gramática y exploran los posibles árboles de derivación que se pueden construir de acuerdo con las reglas de reescritura de la gramática, intentando obtener una secuencia de reglas que genere la cadena de símbolos a procesar.
- Los analizadores ascendentes utilizan la misma estrategia, pero en sentido contrario. Es decir, parten de la cadena de símbolos e intentan obtener el axioma de la gramática.

Respecto al primer grupo, el de los analizadores sintácticos descendentes, esbozamos las ideas generales de las dos aproximaciones principales que se han desarrollado, haciendo mención de sus principales ventajas e inconvenientes:

- La primera aproximación consiste en utilizar una estructura de cola para recorrer en anchura el espacio de posibles derivaciones. Si existe una derivación para la cadena de entrada, está garantizado que el algoritmo la encontrará. No obstante, a menudo dicho espacio de derivaciones puede crecer demasiado. En estos casos, este tipo de algoritmos consumirá una gran cantidad de recursos.
- La segunda aproximación consiste en utilizar una estructura de pila para recorrer el espacio de derivaciones en profundidad. Este tipo de algoritmos consume muy pocos recursos. Sin embargo, pueden no comportarse correctamente ante gramáticas recursivas. En ocasiones, dependiendo del orden en el que se apliquen las reglas de reescritura, la recursividad puede provocar que el algoritmo se quede procesando indefinidamente una cierta rama del árbol de derivaciones sin llegar a ninguna solución, aun cuando la solución está presente en otra rama del árbol.

Y respecto al segundo grupo, uno de los mecanismos de análisis sintáctico ascendente más sencillos e intuitivos es el tradicionalmente conocido como *algoritmo CYK* o *algoritmo Cocke-Younger-Kasami*, aunque realmente fue descubierto de manera independiente por distintas personas. La esencia del algoritmo consiste en la construcción de una tabla de análisis triangular, cuyas celdas se denotan por N_{ij}, para $1 \leq i \leq n - j + 1$ y $1 \leq j \leq n$, donde n es el número de símbolos de la cadena a analizar. Cada celda contendrá un subconjunto de los símbolos no terminales de la gramática. Un símbolo no terminal A estará en N_{ij} si y sólo si $A \stackrel{*}{\Rightarrow} w_i, w_{i+1}, \ldots, w_{i+j-1}$, es decir, si A deriva en un número finito de pasos la subcadena que comienza en la posición i y contiene j símbolos. La cadena pertenece al lenguaje generado por la gramática si el axioma se encuentra en la celda N_{1n}.

Descripción formal del algoritmo CYK

Considerando una cadena $w = w_1, w_2, \ldots, w_n$ y una gramática en forma normal de Chomsky, sin ϵ-producciones, $G = (N, T, P, S)$, la descripción formal del algoritmo es la siguiente:

1. Se inicializa la primera fila de la tabla de análisis, utilizando las reglas que generan directamente los símbolos terminales, como sigue:

$$N_{i1} = \{A \mid A \to w_{i1} \in P\},\ 1 \leq i \leq n$$

2. Para $j = 2, 3, \ldots, n$, hacer lo siguiente:
   - Para $i = 1, 2, \ldots, n - j + 1$, hacer lo siguiente:
     - Inicializar N_{ij} al conjunto vacío.
     - Para $k = 1, 2, \ldots, j - 1$, añadir a N_{ij} todos los símbolos no terminales A para los cuales $A \to BC \in P$, con $B \in N_{ik}$ y $C \in N_{(i+k)(j-k)}$.
3. La cadena w pertenece a $L(G)$ si y sólo si $S \in N_{1n}$.

La figura 1 muestra la tabla de análisis CYK para la gramática

$$S \to AB \mid BC \qquad A \to BA \mid a \qquad B \to CC \mid b \qquad C \to AB \mid a \tag{1}$$

y la cadena $w = bbab$. Dado que $S \in N_{14}$, la cadena w pertenece al lenguaje generado por dicha gramática.

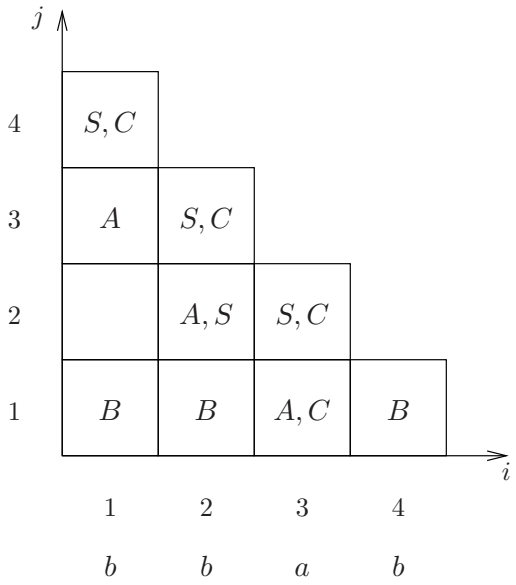

Figura 1: Tabla de análisis CYK para la cadena $w = bbab$

Ventajas e inconvenientes del algoritmo CYK

Aunque resulta de gran interés debido a su simplicidad, el algoritmo CYK presenta varios inconvenientes principales:

- El algoritmo original sólo trabaja con gramáticas en forma normal de Chomsky.

- La complejidad temporal es $\mathcal{O}(n^3)$, donde n es el número de símbolos de la cadena a analizar, independientemente de dicha cadena.
- La complejidad espacial es $\mathcal{O}(n^2)$, independientemente también de la cadena a analizar.

Aún con todo esto, se pueden señalar también las siguientes ventajas:

- El algoritmo CYK básico se puede extender fácilmente para su funcionamiento sobre gramáticas independientes del contexto arbitrarias.
- Presenta inherentemente un esquema natural de paralelización que puede permitir rebajar sus complejidades.
- En general, la presencia de un símbolo no terminal en una celda N_{ij} implica la existencia de un subárbol encabezado por dicho símbolo, que cubre la subcadena $w_i, w_{i+1}, \ldots, w_{i+j-1}$. Así pues, la **extracción de los árboles sintácticos** constituye una tarea realmente sencilla, no sólo en el caso de los árboles de cobertura total correspondientes a la celda superior de la tabla de análisis, sino también en el caso de cualquier otro subárbol correspondiente a cualquier otra celda.
- Si unimos a lo anterior la posibilidad de trabajar con **gramáticas probabilísticas**, nos encontramos con un marco especialmente bien adaptado para el procesamiento no sólo de lenguajes formales, sino también de muchos de los aspectos sintácticos que presentan las lenguas naturales.

Consideraciones estocásticas

El modelo probabilístico más sencillo y más natural para representar las estructuras anidadas y los comportamientos recursivos de los lenguajes es quizás el de las gramáticas independientes del contexto probabilísticas, también llamadas estocásticas. Una gramática de este tipo es simplemente una gramática independiente del contexto que incorpora una probabilidad asociada a cada regla de producción. El propósito de dichas probabilidades es indicar que algunas operaciones de reescritura son más probables que otras. La única restricción impuesta por este modelo es que las probabilidades de las reglas que comparten la misma parte izquierda, es decir, las reglas correspondientes a las distintas posibilidades de reescritura de un mismo símbolo no terminal, deben sumar 1.

Es decir, una gramática independiente del contexto estocástica se define como $G = (N, T, P, S)$, donde:

$$\sum_{\alpha} P(A \to \alpha) = 1, \qquad \forall A \in N. \tag{2}$$

El método para diseñar una gramática estocástica, es decir, la manera de identificar los símbolos, las producciones y las probabilidades, puede ser manual cuando la gramática es pequeña. Pero en la práctica, para las gramáticas de los lenguajes naturales, todos estos elementos se suelen extraer automáticamente de recursos lingüísticos especializados en forma de bancos de árboles o *treebanks*.

Por el momento, nos interesa centrarnos en el problema de cómo asignar una probabilidad a cada frase. La probabilidad de una frase s, de acuerdo con una gramática G, viene dada por:

$$P(s) = \sum_{t} P(s,t) = \sum_{t\,:\,\text{hojas}(t)=s} P(t),$$

donde la variable t recorre el espacio de todos los posibles árboles de análisis para los cuales la secuencia de nodos hoja, leída de izquierda a derecha, coincide con la frase s. Asumiendo la

hipótesis de que las reglas de una gramática estocástica G son independientes, la probabilidad de un nodo cualquiera de un árbol t se calcula recursivamente como el producto de las probabilidades de sus subárboles locales y de la probabilidad de la regla de producción de G que los une. La probabilidad de un árbol t viene dada, por tanto, por la probabilidad de su nodo raíz.

Por ejemplo, sea $G = (N, T, P, S)$ una gramática independiente del contexto estocástica, donde $N = \{S, A, B, C\}$, $T = \{a, b\}$, el conjunto de reglas P, con sus respectivas probabilidades entre paréntesis, viene dado por:

$$\begin{array}{llllllll} S \to A\,B & (0,25), & A \to B\,A & (0,5), & B \to C\,C & (0,1), & C \to A\,B & (0,2), \\ S \to B\,C & (0,75), & A \to a & (0,5), & B \to b & (0,9), & C \to a & (0,8), \end{array}$$

y $S = S$. La frase $s = b\,b\,a\,b$ tiene dos posibles árboles de análisis, tal y como se muestra en la figura 2. En esta figura, los símbolos no terminales de cada nodo llevan como subíndice la probabilidad de la regla mediante la cual generan los subárboles que encabezan. Así pues, la probabilidad de cada árbol es:

$$P(t_1) = 0,9 \times 0,9 \times 0,5 \times 0,5 \times 0,5 \times 0,9 \times 0,25 = 0,02278.$$
$$P(t_2) = 0,9 \times 0,9 \times 0,5 \times 0,5 \times 0,9 \times 0,2 \times 0,75 = 0,02733.$$

Y por tanto, $P(s) = P(t_1) + P(t_2) = 0,02278 + 0,02733 = 0,05011$.

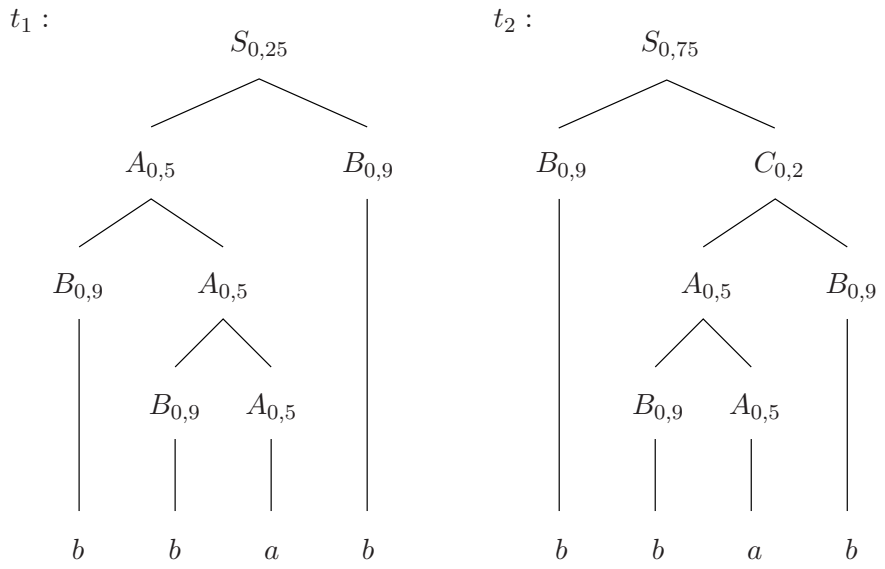

Figura 2: Árboles de análisis probabilístico para la frase $s = b\,b\,a\,b$

A medida que las gramáticas se expanden para conseguir la mayor cobertura posible sobre grandes colecciones de textos, la ambigüedad crece también con ellas. Este fenómeno puede dar como resultado la existencia de múltiples análisis estructurales diferentes para una misma secuencia de palabras. Con el uso de gramáticas estocásticas, se puede obtener una cierta idea de la plausibilidad de cada uno de esos análisis. Es decir, la gramática del ejemplo anterior, no sólo permite calcular la probabilidad de una frase, en este caso $s = b\,b\,a\,b$, sino que además nos indica cuál de los análisis es el más probable, en este caso t_2.

Sin embargo, las gramáticas estocásticas podrían presentar algunas limitaciones potenciales que describimos a continuación:

- Las gramáticas estocásticas no parecen ser del todo *imparciales* en algunos aspectos, lo cual puede resultar inadecuado para determinadas aplicaciones. Por ejemplo, en general, la probabilidad de un árbol pequeño es mayor que la de un árbol grande. Esto podría no ser demasiado importante, ya que las frases de un lenguaje tienden a tener una cierta

longitud media. Pero no cabe duda de que una gramática estocástica asigna demasiada masa de probabilidad a las frases más cortas. De igual manera, en los árboles de análisis, los símbolos no terminales con un número bajo de posibles reescrituras se ven favorecidos sobre los no terminales con muchas posibilidades, ya que las reglas individuales de estos últimos tendrán probabilidades mucho más bajas.

- Debido a que las probabilidades son números entre 0 y 1, si la profundidad media de los árboles de análisis es elevada, podríamos tener un problema de pérdida de precisión al implementar en el ordenador el algoritmo de análisis sintáctico probabilístico, ya que los productos de factores menores que 1 producen números cada vez más pequeños. Este problema puede solventarse fácilmente cambiando cada probabilidad p por su logaritmo y, al mismo tiempo, cambiando los productos por sumas, lo cual, por otra parte, hará que los cálculos sean más rápidos. Y además, por supuesto, la escala logarítmica es coherente en lo que se refiere a la comparación de valores. Es decir, dadas dos probabilidades p_1 y p_2, aunque sus logaritmos van a ser números negativos, si se cumple que $p_1 > p_2$, se verificará también que $log(p_1) > log(p_2)$.

- Las gramáticas estocásticas proporcionan modelos probabilísticos del lenguaje. En un primer momento, cabría esperar, por tanto, que si todas las reglas de producción verifican la restricción (2), entonces
$$\sum_{s \in L(G)} P(s) = \sum_{t} P(t) = 1.$$
Realmente, esto es cierto sólo si la masa de probabilidad de las reglas se acumula en un número finito de árboles de análisis. Así pues, consideremos el siguiente ejemplo.

  Sea G una gramática estocástica con un único símbolo no terminal S, un único símbolo terminal a, y un conjunto de reglas:
$$\begin{aligned} S &\rightarrow a \quad (\tfrac{1}{3}), \\ S &\rightarrow S\ S \quad (\tfrac{2}{3}). \end{aligned}$$
  Esta gramática genera frases de la forma a, aa, aaa, ... Sin embargo, las probabilidades de estas frases son de la forma:
$$\begin{aligned} P(a) &= \tfrac{1}{3} \\ P(aa) &= \tfrac{2}{3} \times \tfrac{1}{3} \times \tfrac{1}{3} = \tfrac{2}{27} \\ P(aaa) &= (\tfrac{2}{3})^2 \times (\tfrac{1}{3})^3 \times 2 = \tfrac{8}{243} \\ &\vdots \end{aligned}$$
  La probabilidad del lenguaje $L(G)$ es la suma de la serie infinita $\frac{1}{3} + \frac{2}{27} + \frac{8}{243} + \ldots$, la cual tiende a $\frac{1}{2}$. Por tanto, la mitad de la masa de probabilidad ha desaparecido en el conjunto infinito de árboles que no generan frases de este lenguaje.

  Distribuciones de probabilidad como la del ejemplo anterior se denominan normalmente *distribuciones inconsistentes*. En la práctica, el uso de distribuciones inconsistentes no presenta excesivos problemas. A menudo, ni siquiera importa si la distribución es consistente o no, especialmente cuando nuestro objetivo principal es la comparación de las magnitudes de probabilidad de los diferentes análisis. Además, se puede demostrar que si los parámetros de nuestras gramáticas estocásticas se estiman a partir de bancos de árboles, siempre podemos obtener distribuciones de probabilidad consistentes.

- Por último, es importante comentar que no está claro que la sintaxis de todos los lenguajes naturales encaje dentro del marco de las gramáticas independientes del contexto, estocásticas o no estocásticas. Incluso aunque lo hiciera, el formalismo se queda muy justo y presenta limitaciones. No obstante, a pesar de su simplicidad, las gramáticas independientes del contexto todavía permiten expresar una gran variedad de las construcciones sintácticas que aparecen en la mayoría de los idiomas, y llevan asociados algoritmos muy eficientes para múltiples tareas de comprensión del lenguaje, una de las cuales es precisamente la tarea del análisis sintáctico.

Consideraciones finales

La tarea del análisis sintáctico o *parsing* ha sido un área de investigación de gran actividad. Debido a ello, existen muchos otros algoritmos de *parsing*, tanto ascendentes como descendentes. No pretendemos aquí realizar una cobertura de la viabilidad y complejidad de todas y cada una de las aproximaciones que se han desarrollado en este terreno, por ser éste un objetivo que pertenece al ámbito de otras asignaturas. Así pues, nos hemos limitado sólo a indicar cuáles son los más importantes, y cuál es el tipo de tarea para el que han sido concebidos y al que mejor se adaptan.

Enunciado de la práctica

Esta práctica se centrará en la implementación del algoritmo CYK (Cocke-Younger-Kasami), el cual, dada una gramática independiente del contexto en forma normal de Chomsky, y una cadena de símbolos de entrada, nos dice si la cadena pertenece o no al lenguaje generado por dicha gramática. Más concretamente, se pide lo siguiente:

1. Implemente una función `es_fnc : Auto.gic -> bool` que indique si una gramática dada está o no en formal normal de Chomsky.

2. Implemente una función `cyk : Auto.simbolo list -> Auto.gic -> bool` que, dada una lista de símbolos de entrada y una gramática, indique si la cadena de entrada pertenece o no al lenguaje generado por la gramática.

   Lo primero que debe hacer esta función es comprobar que la cadena de entrada tiene al menos un símbolo, y que la gramática está en forma normal de Chomsky. Si no es así, la función activará una excepción.

3. Revise su implementación anterior del algoritmo CYK, para definir ahora una nueva función `cyk_plus : Auto.simbolo list -> Auto.gic -> bool * string list` que, o bien devuelve `(false,[])`, o bien devuelve `true` y la lista de todos los posibles árboles de derivación de la cadena de entrada, expresados en forma de secuencias parentizadas de símbolos.

   La realización de este apartado es opcional.

   Ejemplo de ejecución:

```
# let g = gic_of_string "S A B C; a b; S;
                         S -> A B | B C;
                         A -> B A | a;
                         B -> C C | b;
                         C -> A B | a";;
val g : Auto.gic =
```

```
  Gic
   (Conj.Conjunto
     [No_terminal "S"; No_terminal "A"; No_terminal "B"; No_terminal "C"],
    Conj.Conjunto [Terminal "a"; Terminal "b"],
    Conj.Conjunto
     [Regla_gic (No_terminal "S", [No_terminal "A"; No_terminal "B"]);
      Regla_gic (No_terminal "S", [No_terminal "B"; No_terminal "C"]);
      Regla_gic (No_terminal "A", [No_terminal "B"; No_terminal "A"]);
      Regla_gic (No_terminal "A", [Terminal "a"]);
      Regla_gic (No_terminal "B", [No_terminal "C"; No_terminal "C"]);
      Regla_gic (No_terminal "B", [Terminal "b"]);
      Regla_gic (No_terminal "C", [No_terminal "A"; No_terminal "B"]);
      Regla_gic (No_terminal "C", [Terminal "a"])],
    No_terminal "S")

# let c1 = cadena_of_string "b b a b";;
val c1 : Auto.simbolo list =
  [Terminal "b"; Terminal "b"; Terminal "a"; Terminal "b"]

# cyk_plus c1 g;;
- : bool * string list =
true,
["(S (A (B b) (A (B b) (A a))) (B b))";
 "(S (B b) (C (A (B b) (A a)) (B b)))"]

# let c2 = cadena_of_string "b b b b";;
val c1 : Auto.simbolo list =
  [Terminal "b"; Terminal "b"; Terminal "b"; Terminal "b"]

# cyk_plus c2 g;;
- : bool * string list = false, []
```

Sugerencias: Para la extracción de todos los posibles árboles de derivación de la cadena de entrada, o en general de cualquier subcadena de la misma, existen al menos tres posibles estrategias.

- La primera de ellas consiste en aplicar el algoritmo CYK básico, y no hacer nada más en este primer momento. Posteriormente, cuando nos piden todos los posibles árboles (o subárboles) de análisis de la cadena de entrada (o de cualquiera de sus subcadenas), localizamos la celda que da cobertura a esa secuencia de símbolos terminales, nos fijamos en los símbolos no terminales que contiene dicha celda, e intentamos deducir la razón por la cual aparecen ahí. Realmente, esta aproximación no tiene mucho interés, ya que es equivalente a volver a implementar el algoritmo CYK, pero esta vez en sentido inverso, es decir, desde los símbolos no terminales de la tabla hacia los símbolos terminales de la cadena.
- En contraposición a lo anterior, la segunda alternativa es la de ir calculando ya todos los posibles árboles y subárboles de análisis, a medida que el algoritmo CYK va rellenando las celdas de la tabla. Los árboles podrían almacenarse como secuencias parentizadas de símbolos. Y cada cruce que se realice entre los símbolos no terminales de dos celdas, para producir nuevos símbolos no terminales en otra celda superior, llevaría asociado su correspondiente cruce de árboles, para producir los nuevos árboles

de análisis de dicha celda superior. Con esta aproximación, las peticiones de los árboles de análisis de cualquier subcadena se resuelven de forma inmediata, porque ya han sido precalculados todos. La desventaja, sin embargo, es que muy probablemente estemos realizando numerosos cálculos innecesarios que luego no van a ser solicitados, lo cual además podría implicar unos requerimientos de almacenamiento muy elevados.

- Por tanto, quizás la mejor alternativa venga dada por una aproximación intermedia. El algoritmo CYK básico puede modificarse fácilmente para que, cada vez que añada un símbolo no terminal a una celda, incorpore una traza que especifique qué regla concreta lo genera y a qué celda pertenecen cada uno de los no terminales de la parte derecha de dicha regla. La figura 3 muestra la tabla que se obtiene al aplicar este método a la cadena de entrada *bbab*, con la gramática (1). En esta tabla podemos observar, por ejemplo, que la celda N_{23} incluye al símbolo C como consecuencia del cruce entre los símbolos A y B de las celdas N_{22} y N_{41}, respectivamente.

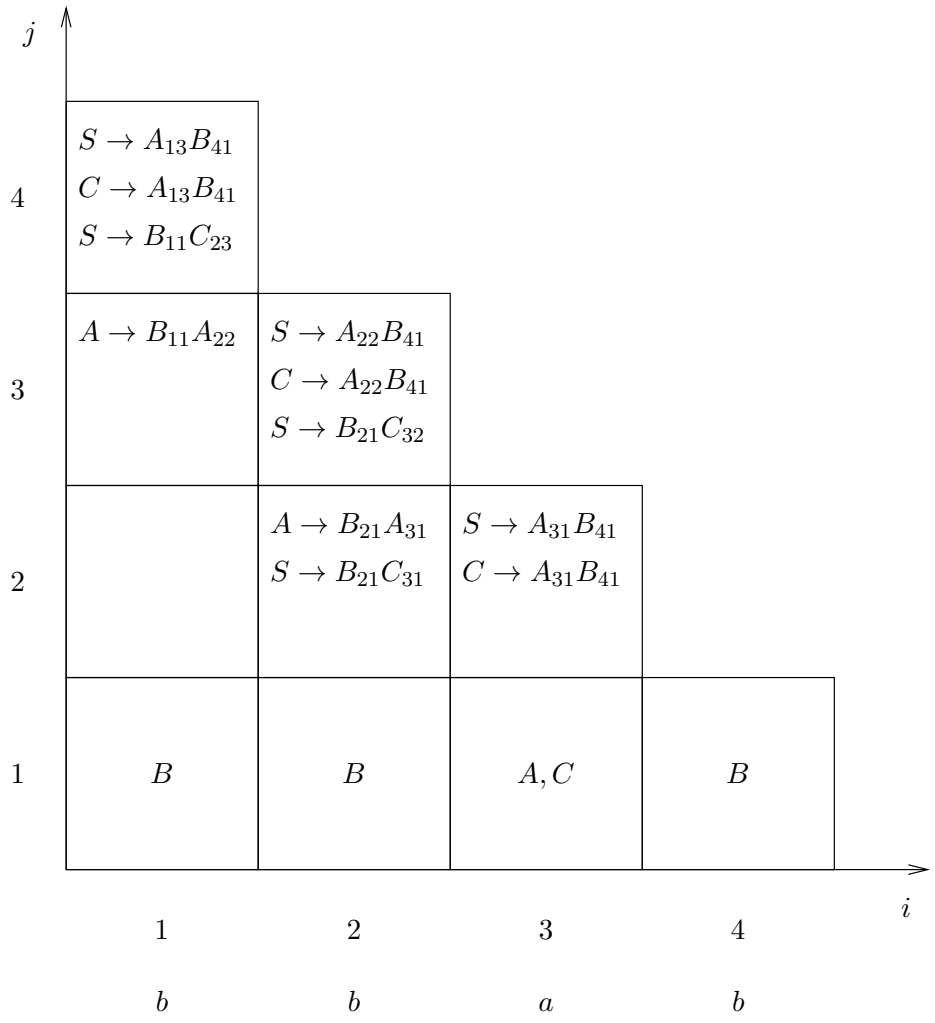

Figura 3: Tabla de análisis dinámico para la frase $s = b\,b\,a\,b$

Por supuesto, al igual que en la primera alternativa, la extracción de los árboles requerirá una navegación recursiva descendente a lo largo de la tabla. Pero dicha navegación estará guiada por las trazas y no repetirá todas los cálculos de todos los posibles cruces de celdas y símbolos que realiza el algoritmo CYK básico, sino sólo los involucrados en la obtención de los árboles solicitados. La figura 4 muestra los árboles de cobertura total para la cadena *bbab*, obtenidos mediante este método a

partir de todas las trazas del símbolo S presentes en la celda N_{14}.

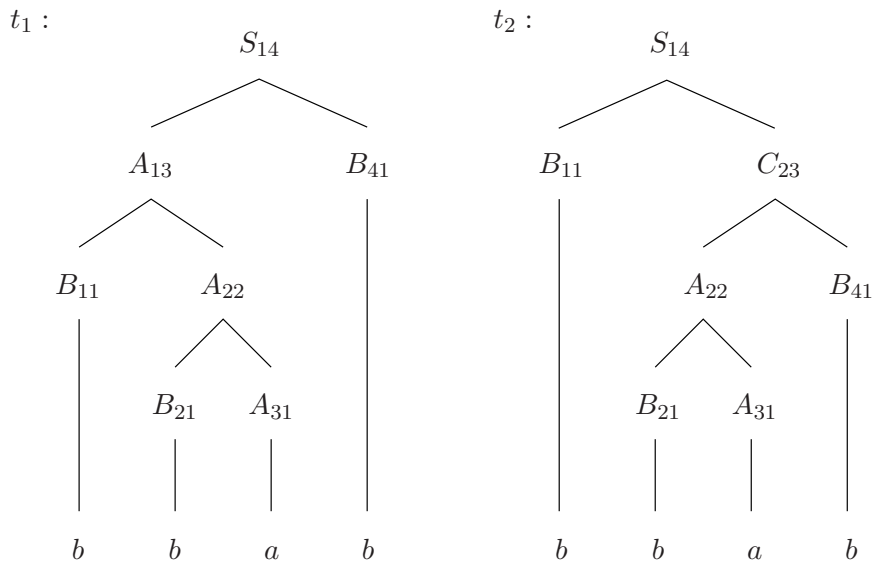

Figura 4: Árboles de análisis dinámico para la frase $s = b\,b\,a\,b$

La técnica a la que pertenecen este tipo de aproximaciones se denomina programación dinámica, y se caracteriza por presentar buenos compromisos entre las complejidades temporales y espaciales de los procesos que se resuelven con ella.

4. Revise su implementación anterior del algoritmo CYK, para definir una nueva función `cyk_prob : Auto.simbolo list -> gicp -> bool * (string * float) list` que, o bien devuelve `(false,[])`, o bien devuelve `true` y la lista de todos los posibles árboles de derivación de la cadena de entrada, expresados en forma de secuencias parentizadas de símbolos y acompañados de su correspondiente probabilidad de análisis.

   La realización de este apartado es opcional.

   Sugerencias: Implemente un nuevo tipo `gicp` para soportar las gramáticas independientes del contexto probabilísticas. Quizás lo mejor sea definir también un nuevo tipo `regla_gicp` que soporte la probabilidad de cada regla de escritura. Y, adicionalmente, para poder construir de manera cómoda valores del tipo `gicp` a partir de valores del tipo `gic`, quizás sea útil implementar una función `gicp_of_gic : Auto.gic -> float list -> gicp`, que a partir de una gramática convencional y de una lista de probabilidades, construya la correspondiente gramática probabilística asignando las probabilidades de la lista a cada regla de reescritura.

   Posible ejemplo de ejecución:

```
# let gp = gicp_of_gic g [0.25; 0.75; 0.5; 0.5; 0.1; 0.9; 0.2; 0.8];;
val gp : gicp =
  Gicp
   (Conj.Conjunto
     [No_terminal "S"; No_terminal "A"; No_terminal "B"; No_terminal "C"],
    Conj.Conjunto [Terminal "a"; Terminal "b"],
    Conj.Conjunto
     [Regla_gicp (No_terminal "S", [No_terminal "A"; No_terminal "B"], 0.25);
      Regla_gicp (No_terminal "S", [No_terminal "B"; No_terminal "C"], 0.75);
      Regla_gicp (No_terminal "A", [No_terminal "B"; No_terminal "A"], 0.5);
```

```
      Regla_gicp (No_terminal "A", [Terminal "a"], 0.5);
      Regla_gicp (No_terminal "B", [No_terminal "C"; No_terminal "C"], 0.1);
      Regla_gicp (No_terminal "B", [Terminal "b"], 0.9);
      Regla_gicp (No_terminal "C", [No_terminal "A"; No_terminal "B"], 0.2);
      Regla_gicp (No_terminal "C", [Terminal "a"], 0.8)],
    No_terminal "S")

# cyk_prob c1 gp;;
- : bool * (string * float) list =
true,
[("(S (A (B b) (A (B b) (A a))) (B b))", 0.02278);
 ("(S (B b) (C (A (B b) (A a)) (B b)))", 0.02733)]
```