Universidad de Zaragoza

Departamento de Informática e Ingeniería de Sistemas

Aplicación de las 2-estructuras a las gramáticas del lenguaje humano y representación gráfica de ambas

Proyecto Fin de Carrera

Ingeniería en Informática

Autor: Daniel Larraz Hurtado

Directora: Elvira Mayodormo Cámara

Centro Politécnico Superior

Agosto de 2010

Curso 2009-2010

Índice

A Teoría de las 2-estructuras

En este anexo se introduce las nociones fundamentales de las 2-estructuras. Por cuestiones de claridad y brevedad muchas ideas son explicadas 'informalmente' y por consiguiente se remite a [1,2,4] para una descripción más rigurosa y extensa.

A.1. Conceptos básicos

Una *2-estructura* es una tupla (D, R) formada por un dominio finito D junto con una relación de equivalencia R sobre los pares ordenados $(x, y) \in D \times D$ con $x \neq y$. Por tanto, una 2-estructura puede ser considerada como un grafo dirigido completo con un 'coloreado abstracto' de las aristas.

Un coloreado de las aristas puede hacerse concreto si se añade una función de etiquetado a las aristas de forma que dos aristas tienen la misma etiqueta si pertenecen a la misma clase de equivalencia. Una 2-estructura asociada a dicha función se llama una 2-estructura *etiquetada*.

Ejemplo A.1. En la parte izquierda de la ilustración A.1 vemos una representación gráfica de una 2-estructura. En ella, las etiquetas fijadas son arbitrarias y sólo sirven para distinguir a qué clase de equivalencia pertenece cada arista. Si consideremos que las etiquetas forman parte de la definición, ese mismo grafo representaría una 2-estructura etiquetada.

Si observamos el ejemplo, podemos ver que para cada arista (x, y) etiquetada con I la arista (y, x) está etiquetada con H. Lo mismo ocurre con las etiquetadas con D y E respectivamente. Cuando esto ocurre la 2-estructura admite una representación más compacta donde una de las dos aristas es omitida. En la parte derecha de la ilustración A.1 está dibujada la representación compacta del ejemplo.

Una 2-estructura puede descomponerse en 2-estructuras más simples. Esta descomposición, que es *única* para las 2-estructuras etiquetadas, es representada con una estructura en forma de árbol. En dicho árbol cada nodo interno está etiquetado con una 2-estructura cuyos elementos 'representan' o bien un elemento del dominio o bien un subconjunto.

La idea en la que se basa la descomposición consiste en hallar subconjuntos del dominio, llamados *clanes*, en los que los elementos contenidos se relacionen de la misma forma con todos aquellos elementos de fuera del subconjunto.

Ejemplo A.2. En la ilustración A.2 puede verse la descomposición, también llamada la forma (*shape*), de la 2-estructura etiquetada del ejemplo A.1[1].

[1] Las 2-estructuras de cada nodo interno están representadas de forma compacta.

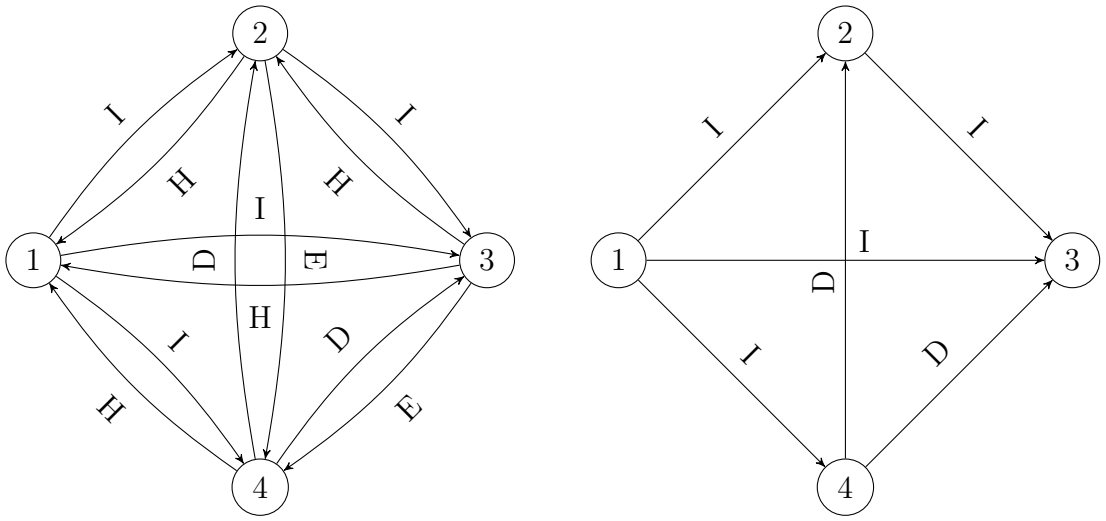

Ilustración A.1 Ejemplo de una 2-estructura junto a una representación compacta de la misma.

Notar que en el ejemplo A.1, todas las aristas que parten del elemento 1 están etiquetadas con I. Esto queda reflejado en la 2-estructura 'contenida' en la raíz del árbol de la descomposición. En ella uno de los dos elementos (nodos) representa al 1 mientras que el otro representa al resto de elementos (2, 3 y 4) con los que el 1 se relaciona de la misma forma. Estos elementos pueden encontrarse en las hojas del subárbol al que queda conectado el elemento representante.

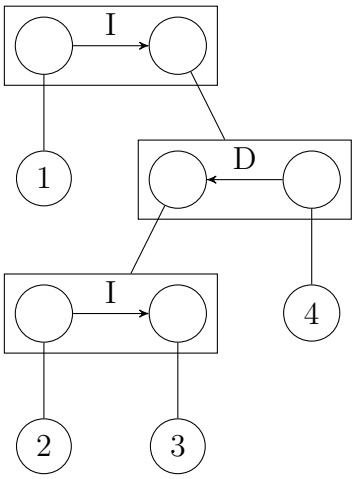

Ilustración A.2 La forma de la 2-estructura etiquetada del ejemplo A.1.

Una propiedad que se cumple en la descomposición es que las 2-estructuras en las que se factoriza la 2-estructura original tienen que pertenecer al menos a alguna de

tres subclases *especiales*. Estas subclases son las 2-estructuras completas, las lineales y las primitivas.

Una 2-estructura es *completa* cuando sólo tiene definida una clase de equivalencia. Es *lineal* cuando tiene dos clases y las aristas contenidas en cada una de ellas establecen un orden total sobre los elementos del dominio. Y es *primitiva* cuando, ni siendo completa ni siendo lineal, no puede descomponerse en otras 2-estructuras.

Antes de pasar a la siguiente sección vamos a dar una última definición en relación a las aristas de una 2-estructura que será necesario conocer.

Dada una arista (x, y) se dice que es *símetrica* si y sólo si la arista (y, x) pertenece a la misma clase de equivalencia. En caso contrario se dice que es *antisimétrica*. Una 2-estructura es simétrica (antisimétrica) si todas sus aristas son símetricas (antisimétricas). Por ejemplo, la 2-estructura mostrada en la ilustración A.1 es antisimétrica.

A.2. T-estructuras y textos alternantes

Además de las 2-estructuras especiales mencionadas previamente hay otra subclase de gran importancia, las T-estructuras. Una *T-estructura* es una 2-estructura antisimétrica que cumple la propiedad *angular*. Dicha propiedad viene a decir, informalmente, que si se escoge tres elementos cualesquiera del dominio, la subestructura inducida por las tres aristas que los unen no es una 2-estructura primitiva.

Las T-estructuras están relacionadas con la generalización del concepto de función y de una palabra en un lenguaje formal. Para verlo se van a presentar una serie de definiciones.

Una *función estructurada* f es un par ordenado (λ, g) tal que λ es una función, y g es una 2-estructura etiquetada con el mismo dominio que λ. Si g es una T-estructura entonces f es una *T-función*.

Una *palabra* es un par ordenado (λ, ρ) donde λ es una función que asigna a los elementos de un conjunto (a los que llamaremos carácteres) un símbolo de un alfabeto y ρ es un orden lineal sobre dicho conjunto de carácteres. Notar que el orden lineal establece una secuencia de los elementos. Por tanto, esta definición coincide prácticamente con la noción que solemos tener de una palabra en un lenguaje formal. La generalización viene a continuación.

Un *texto* es una tupla $(\lambda, \rho_1, \rho_2)$ donde λ es una función, y ρ_1, ρ_2 son dos ordenes lineales sobre el dominio de λ. Luego, un texto puede verse como una palabra junto a un orden lineal que como veremos asociará una estructura a la palabra.

Dado un texto, podemos definir una función estructurada que represente mediante las clases de equivalencia de la 2-estructura los dos ordenes lineales de los

que se compone el texto. Está demostrado que dicha 2-estructura es siempre una T-estructura y, por tanto, la función estructurada una T-función.

En concreto, en este trabajo, nos interesa que las T-estructuras que quedan definidas se puedan descomponer solamente en 2-estructuras lineales. Esto se debe a que en esos casos, el árbol en el que se descompone la T-estructura tiene una correspondencia con un árbol ordenado con las hojas etiquetadas. Es decir, los textos con los que trabajamos quedan asociados inequívocamente a un árbol ordenado.

En la siguiente sección se explicará intuitivamente la correspondencia entre los mencionados textos, llamados textos *alternantes*, y los árboles ordenados y se verá el por qué de su nombre.

A.3. Textos alternantes y árboles ordenados

Dada un orden total ρ sobre un dominio y s la secuencia asociada a dicho orden, $rev(\rho)$ es el orden total que tiene como secuencia asociada el reverso de s.

Dado un árbol ordenado (el orden de los nodos importa) que no contenga cadenas (nodos intermedios con un solo hijo) y en el que sólo los nodos hoja estén etiquetados, podemos especificar dos textos $(\lambda, \rho_1, \rho_2)$ y $(\lambda, \rho_1, rev(\rho_2))$ que lo identifiquen inequívocamente cumpliendo las siguientes propiedades: λ es una función finita que etiqueta cada nodo hoja, ρ_1 es un orden total sobre las hojas cuya secuencia asociada coincide con la frontera del árbol y ρ_2 es un orden total sobre las hojas resultado de recorrer el árbol en profundidad pero alternando la dirección (de izquierda a derecha o viceversa) en cada nodo interno. Según se elija una dirección u otra en el nodo raíz, esto da lugar a la representación del árbol con una u otra tupla.

Ejemplo A.3. El árbol más a la izquierda de la ilustración A.3 puede ser representado por los textos $(\lambda, \rho, (1, 5, 2, 3, 4))$ y $(\lambda, \rho, (4, 3, 2, 5, 1))$ donde $\lambda = \{(1, a), (2, a), (3, e), (4, b), (5, b)\}$ y $\rho_1 = (1, 2, 3, 4, 5)$. Los otros dos árboles de la ilustración A.3 representan gráficamente las dos alternativas.

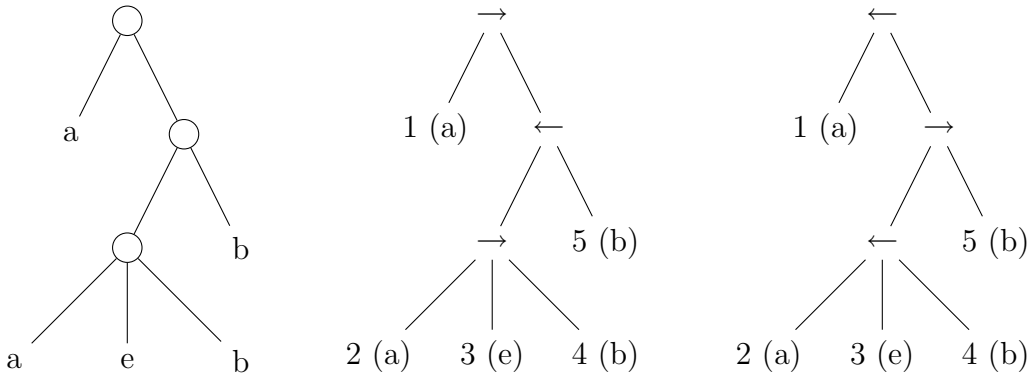

Ilustración A.3 Un árbol ordenado junto a sus dos representaciones.

B El lenguaje natural y las MCSG

En este anexo se describe parte de la problemática del procesamiento del lenguaje natural y se presenta dos formalismos, las gramáticas de adjunción de árboles y las gramáticas de núcleo, pertenecientes al conjunto de gramáticas suavemente sensibles al contexto.

B.1. El procesamiento del lenguaje natural

El lenguaje natural, término técnico con el que se engloba al conjunto de lenguas habladas, es casi en cualquier aspecto más complejo de lo esperado [6]. A diferencia de los lenguajes artificiales, como los lenguajes de programación, que son concebidos con un propósito concreto y a los que se les han impuesto una estructura fija, los lenguajes naturales han sido 'diseñados' a lo largo de siglos de evolución por la gente que los habla.

Esta diferencia en su origen tiene una implicación importante. Mientras que el lenguaje natural es inherentemente ambiguo y contiene construcciones típicamente no-independientes del contexto que hacen que su procesado sea en general inviable, los lenguajes artificiales pueden ser procesados eficientemente al ser descritos a priori con gramáticas independientes del contexto exentas de cualquier ambigüedad.

Las gramáticas suavemente sensibles al contexto (*Mildly Context Sensitive Grammars*, MCSG) pretenden capturar la sintaxis del lenguaje natural y conseguir su procesado eficiente [7,9].

Entre los lenguajes que describen estas gramáticas encontramos una subclase de gran interés debido a tres motivos. Los lenguajes que contiene capturan un amplio espectro de las dependencias del contexto del lenguaje natural, son reconocibles en tiempo polinómico y existen cuatro formalismos independientes entre sí que los generan [8]. Son los lenguajes descritos por las grámaticas de adjunción de árboles (*Tree Adjoining Grammars*, TAG), las gramáticas de núcleo (*Head Grammars*, HG), las gramáticas lineales de índices (*Linear Indexed Grammars*, LIG) y las gramáticas categoriales combinatorias (*Combinatory Categorial Grammars*, CCG). En este anexo se introducirá las dos primeras que son utilizadas en el trabajo principal.

B.2. El modelado de las dependencias

Al modelar el lenguaje natural con lenguajes formales, se suele hacer corresponder las frases y las palabras del primero con las cadenas y símbolos del alfabeto del segundo. De esta forma fenómenos sintácticos del lenguaje natural como la replicación (encontradas en ciertas variantes del alemán), las concordancias cruzadas (presentes en el holandés) y las concordancias múltiples, pueden ser representadas mediante

los lenguajes $\{ww \mid w \in \{a,b\}^*\}$, $\{a^n b^m c^n d^m \mid n,m \geq 1\}$ y $\{a^n b^n c^n d^n \mid n \geq 1\}$ respectivamente.

Todos estos lenguajes dan lugar a palabras con dependencias anidadas y cruzadas que pueden relacionar diferentes símbolos dependiendo de la gramática utilizada. En la parte izquierda de la ilustración B.1 se muestra una palabra con dependencias cruzadas, y en la derecha una palabra con dependencias tanto anidadas como cruzadas.

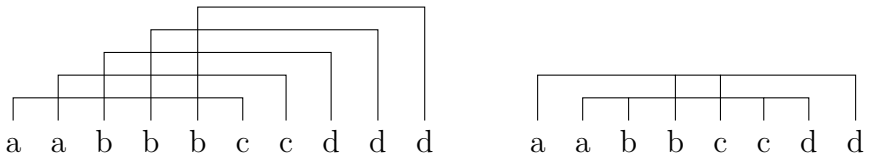

Ilustración B.1 Algunos ejemplos de palabras junto a sus dependencias.

Las gramáticas independientes del contexto asignan a las palabras generadas sólo dependencias anidadas. No obstante, podemos plantearnos asignar dependencias cruzadas a lenguajes propiamente independientes del contexto. Logicamente, para ello se necesita hacer uso de formalismos más potentes como las gramáticas de adjunción de árboles.

Por ejemplo, con una grámatica independiente del contexto se podría generar la palabra *aaaebbb*, pertenciente al lenguaje $\{a^n e b^n \mid n \geq 0\}$, con las dependencias mostradas a la izquierda de la ilustración B.2. Sin embargo no sería posible generarla con las dependencias mostradas a la derecha de esa misma ilustración.

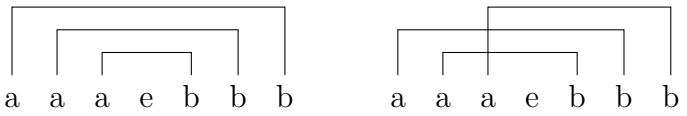

Ilustración B.2 Palabra *aaaebbb* con dependencias sólo anidadas y con dependencias anidadas y cruzadas.

B.3. Gramáticas de adjunción de árboles (TAG)

Las gramáticas de adjunción de árboles (*Tree Adjoining Grammars*, TAG) son una extensión de las gramáticas independientes del contexto que en vez de utilizar producciones utiliza árboles elementales para derivar las palabras. A continuación se presenta su definición extraída de [7].

Formalmente una gramática de adjunción de árboles es una tupla (V_T, V_N, I, A, S) donde:

- V_T es un conjunto finito de símbolos *terminales*.
- V_N es un conjunto finito de símbolos *no-terminales*.
- I es un conjunto finito de *árboles iniciales*.
- A es un conjunto finito de *árboles auxiliares*.
- $S \in V_N$ es el axioma de la gramática.

Los árboles en $I \cup A$ se denominan *árboles elementales* de la gramática.

Los árboles iniciales se caracterizan porque su raíz está etiquetada por el axioma de la gramática, sus nodos interiores están etiquetados por no-terminales y sus nodos hoja están etiquetados por terminales o por la palabra vacía, denotada por ϵ.

Los árboles auxiliares son como los árboles iniciales con la excepción de que la etiqueta de su raíz puede ser un no-terminal arbitrario y porque uno de sus nodos hoja, que recibe el nombre de *pie* está etiquetado por el mismo no-terminal que etiqueta su raíz.

Los árboles elementales se pueden combinar entre sí para crear *árboles derivados*, los cuales a su vez se pueden combinar con otros árboles para formar árboles derivados más grandes. La operación mediante la cual se combinan los árboles se denomina *adjunción*. Mediante una adjunción se construye un nuevo árbol a partir de un árbol auxiliar β y de otro árbol γ, que puede ser un árbol inicial, auxiliar o derivado de adjunciones realizadas previamente. En su forma más simple, una adjunción puede tener lugar si la etiqueta de un nodo del árbol γ (denominado *nodo de adjunción*) coincide con la etiqueta del nodo raíz de un árbol auxiliar β. En tal caso, el árbol derivado resultante se construye como sigue:

1. El subárbol de γ dominado por el nodo de adjunción se escinde de γ, aunque se deja una copia del nodo de adjunción en γ.
2. El árbol auxiliar β se pega a la copia del nodo de adjunción de tal forma que la raíz del árbol auxiliar se identifica con dicha copia.
3. El subárbol escindido de γ se pega al nodo pie del árbol auxiliar β de tal modo que la raíz del subárbol escindido (el nodo de adjunción) se identifica con el nodo pie de β.

La aplicabilidad de una adjunción tal y como ha sido descrita sólo depende de las etiquetas de los nodos. Sin embargo, por conveniencia se puede especificar para cada nodo un conjunto de restricciones que permite indicar con más precisión los árboles auxiliares que pueden ser adjuntados. Las restricciones asociadas a un nodo, que se denominan restricciones de adjunción, pueden ser de los tipos siguientes:

– Restricciones de *adjunción selectiva* (SA) que especifican el subconjunto de árboles auxiliares que pueden participar en una operación de adjunción. En todo caso, no es obligatorio realizar una adjunción.

– Restricciones de *adjunción nula* (NA) que impiden la realización de adjunciones.

– Restricciones de *adjunción obligatoria* (OA) que especifica un subconjunto de árboles auxiliares uno de los cuales ha de ser utilizado obligatoriamente en una operación de adjunción.

El lenguaje definido por una gramática de adjunción de árboles es el conjunto de cadenas $w \in V_T$ tal que w constituye la frontera de un árbol derivado a partir de un árbol inicial.

Ejemplo B.1. La gramática $G = (\{a, b, c, d\}, \{S, T\}, \{\alpha_1, \alpha_2\}, \{\beta_1, \beta_2, \beta_3\}, S)$ representada en la ilustración B.3, que genera el lenguaje $\{a^n b^m c^n d^m \mid n, m \geq 1\}$, es capaz de generar la palabra *aabbbccddd* respetando las dependencias mostradas a la izquierda de la ilustración B.1.

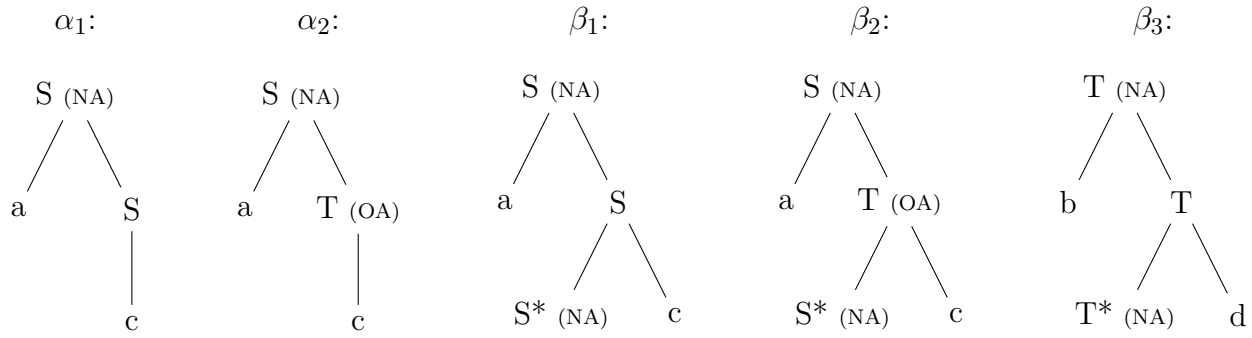

Ilustración B.3 Gramática de adjunción de árboles que genera el lenguaje $a^n b^m c^n d^m$

En la parte izquierda de la ilustración B.4 se indica el árbol de derivación de la palabra *aabbbccddd*. Éste tiene etiquetada la raíz con el árbol inicial y los demás nodos con un árbol auxiliar y el nodo en el que se realizó la adjunción[2]. En la parte derecha se muestra el resultado de realizar la primera adjunción de β_2 en α_1 al derivar.

En la ilustración B.5 está dibujado el árbol derivado final donde se puede comprobar que los terminales dependientes entre sí (ver la ilustración B.1) han sido generados del mismo árbol elemental.

[2] Se emplea el direccionamiento de Gorn que utiliza 0 para referirse al nodo raíz, k para referirse al k-ésimo hijo del nodo raíz y $p.q$ para referirse al q-ésimo hijo del nodo con dirección p.

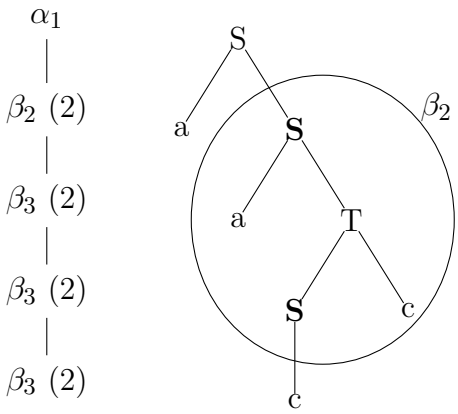

Ilustración B.4 Árbol de derivación de la palabra *aabbbccddd* y el resultado de la primera adjunción.

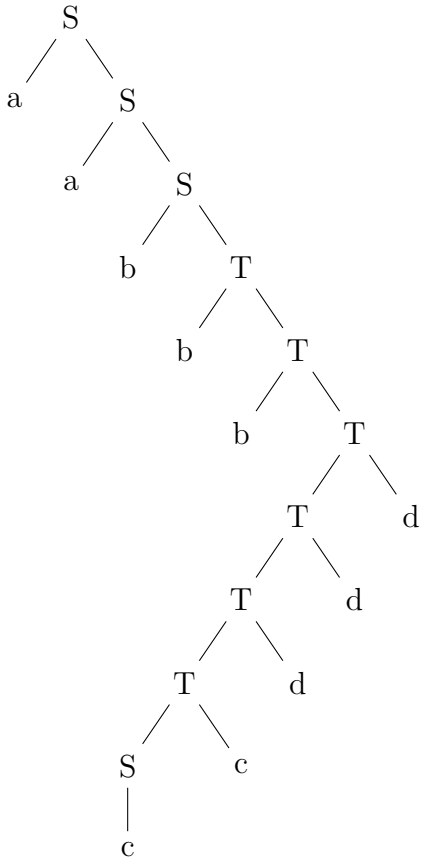

Ilustración B.5 Árbol derivado de la palabra *aabbbccddd*.

B.4. Gramáticas de núcleo (HG)

Las gramáticas de núcleo (*Head Grammars*, HG) pueden considerarse una generalización de las gramáticas independientes del contexto (CFG) que, además de la operación de sustitución, utilizan una nueva operación denominada de *wrapping*. Mientras que los no-terminales de las CFG derivan cadenas de terminales, los no-terminales de las grámaticas de núcleo derivan cadenas con núcleo. Siguiendo la definición dada por Vijay-Shanker y Weir en [8], las cadenas con núcleo son un par de cadenas de terminales (u, v); a este par lo denotaremos con $u{\uparrow}v$. Esta definición es equivalente a la que Pollard dio originalmente pero resuelve ciertos problemas notacionales.

En [8] se demuestra que las gramáticas con núcleo son débilmente equivalentes a otros formalismos como las gramáticas de adjunción de árboles. Por tanto, ambas generan la misma clase de lenguajes.

Sin embargo, las gramáticas de núcleo no proporcionan una árbol derivado [11,12] a las cadenas de un lenguaje. Aunque en [11] se muestra que es posible construir un árbol de derivación en el que cada nodo interno es anotado por un no-terminal y por la operación utilizada para combinar los pares de cadenas con núcleo que son derivados por los nodos hijo.

Formalmente una gramática de núcleo es una 4-tupla (V_N, V_T, S, P) donde:

- V_N es un conjunto finito de símbolos *no-terminales*.
- V_T es un conjunto finito de símbolos *terminales*.
- $S \in V_N$ es el axioma de la gramática.
- P es un conjunto finito de producciones de la forma $A \to f(\sigma_1, \ldots, \sigma_n)$, con $A \in V_N$, $n \geq 1$, $f \in \{W, C_{1,n}, C_{2,n}, \ldots, C_{n,n}\}$, $\sigma_1, \ldots, \sigma_n \in V_N \cup (V_T^* \times V_T^*)$ y si $f = W$ entonces $n = 2$.

$C_{i,n} : (V_T^* \times V_T^*)^n \to (V_T^* \times V_T^*)$ es la operación de concatenación:

$$C_{i,n}(u_1{\uparrow}v_1, \ldots, u_i{\uparrow}v_i, \ldots, u_n{\uparrow}v_n) = u_1 v_1 \ldots u_i{\uparrow}v_i \ldots u_n v_n$$

$W : (V_T^* \times V_T^*)^2 \to (V_T^* \times V_T^*)$ es la operación de wrapping:

$$W(u_1{\uparrow}v_1, u_2{\uparrow}v_2) = u_1 u_2{\uparrow}v_2 v_1$$

La relación de derivación $\Rightarrow$ se define como sigue:

- $u{\uparrow}v \overset{0}{\Rightarrow} u{\uparrow}v$ para todo $u{\uparrow}v \in V_T^* \times V_T^*$.
- Si $A \to f(\sigma_1, \ldots, \sigma_n) \in P$ entonces

$$A \overset{k}{\Rightarrow} f(u_1{\uparrow}v_1, \ldots, u_n{\uparrow}v_n)$$

donde $\sigma_i \overset{k_i}{\Rightarrow} u_i{\uparrow}v_i$ con $(1 \leq i \leq n)$ y $k = 1 + \sum_{1 \leq i \leq n} k_i$.

El lenguaje generado por una gramática de núcleo queda definido por todos los $uv \in V_T^*$ tales que $S \overset{*}{\Rightarrow} u{\uparrow}v$, donde $\overset{*}{\Rightarrow} = \cup_{k \geq 0} \overset{k}{\Rightarrow}$.

Ejemplo B.2. La gramática de núcleo $G = (\{S, T\}, \{a, b, c, d\}, S, P)$ genera el lenguaje $\{a^n b^n c^n d^n \mid n \geq 0\}$ donde P es como sigue.

$$P = \{S \to C_{1,1}(\epsilon{\uparrow}\epsilon), \quad S \to C_{2,3}(a{\uparrow}\epsilon, T, d{\uparrow}\epsilon), \quad T \to W(S, b{\uparrow}c)\}$$

Una derivación de la cadena *aabbccdd* viene dada por los siguientes pasos.

$$S \overset{1}{\Rightarrow} C_{1,1}(\epsilon{\uparrow}\epsilon) = \epsilon{\uparrow}\epsilon$$

de aquí $T \overset{2}{\Rightarrow} W(\epsilon{\uparrow}\epsilon, b{\uparrow}c) = b{\uparrow}c$

de aquí $S \overset{3}{\Rightarrow} C_{2,3}(a{\uparrow}\epsilon, b{\uparrow}c, d{\uparrow}\epsilon) = ab{\uparrow}cd$

de aquí $T \overset{4}{\Rightarrow} W(ab{\uparrow}cd, b{\uparrow}c) = abb{\uparrow}ccd$

de aquí $S \overset{5}{\Rightarrow} C_{2,3}(a{\uparrow}\epsilon, abb{\uparrow}ccd, d{\uparrow}\epsilon) = aabb{\uparrow}ccdd$

.

C Generación de una gramática TAG a partir de un árbol de dependencias

En este anexo se explica cómo se puede hacer corresponder un árbol de dependencias con una gramática TAG. Primero se presenta el caso en el que el árbol de dependencias sólo se compone de una rama principal o espina (todo nodo interno tiene como máximo un nodo no hoja como hijo directo) y después se tiene en consideración las peculiaridades del caso en el que el árbol de dependencias tiene varias ramas independientes.

C.1. Generación de una TAG desde un árbol de dependencias con una sola espina

Una gramática TAG quedará especificada si se construye una quíntupla (V_T, V_N, I, A, S) tal como está definida en el anexo B.3. A continuación se detalla cómo construir cada componente de dicha tupla.

El conjunto de terminales viene determinado por las etiquetas distintas de las hojas del árbol.

Para cada dependencia D_i construimos un árbol elemental que se obtiene al eliminar del árbol de dependencias los nodos etiquetados con una dependencia distinta D_j junto a sus hijos hoja, todos excepto aquel que es hijo del nodo interno etiquetado con D_i más lejano de la raíz del árbol (si es que existe tal hijo). En caso de estar presente, el árbol elemental será un árbol auxiliar y dicho nodo será su pie. En caso contrario el árbol elemental será el árbol inicial a partir del cual se inició la derivación.

Además cada nodo interno de cada árbol elemental deberá ser sustituido por un no-terminal tal que permita mediante la operación de adjunción concatenarse a los subárboles adyacentes a los que está conectado en el árbol de dependencias, y de forma que el árbol inicial tengan como raíz el axioma de la gramática.

Ejemplo C.1. Consideremos el árbol de dependencias mostrado en la ilustración C.1. Si nos fijamos en las hojas tenemos que $V_T = \{a, b, c, d\}$. Por otra parte, si eliminamos del árbol todos los nodos internos que no estén etiquetados con D_0 y sus nodos hijo hoja, obtenemos el árbol inicial α_1 de la ilustración C.2. En él se ha utilizado dos no-terminales arbitrarios en el que S es el axioma de la gramática por estar en la raíz del árbol inicial. Sabemos que es el árbol inicial porque el nodo etiquetado con A no tiene hijos no hoja.

Repitiendo el proceso para la dependencia D_1 obtenemos el árbol auxiliar β_1 de la ilustración C.2. En este caso, el no-terminal A viene impuesto por el elegido para el árbol α_1 ya que debe ser posible la adjunción. Para las dependencias D_2, D_3 y

D_4 se tiene los árboles auxiliares γ_1, γ_2 y γ_3 (tal como se describen en la ilustración C.2) donde $T_1 = B$, $R_1 = C$, $T_2 = C$, $R_2 = D$, $T_3 = D$ y $R_3 = E$.

Luego la gramática final es $(\{a, b, c, d\}, \{S, A, B, C, D, E\}, \{\alpha_1\}, \{\beta_1, \gamma_1, \gamma_2, \gamma_3\}, S)$.

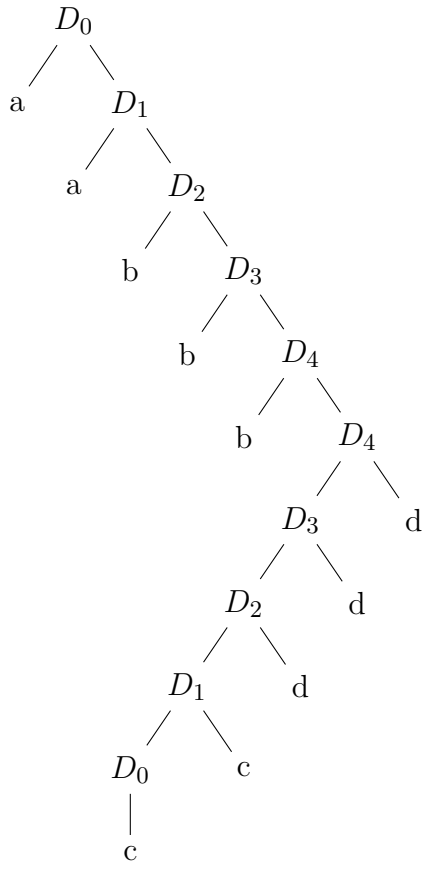

Ilustración C.1 Árbol de dependencias de la palabra *aabbbccddd*.

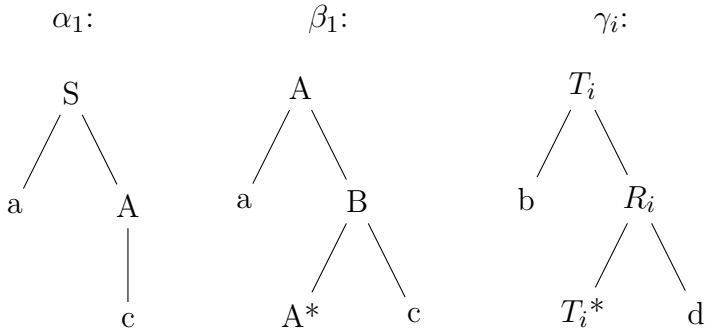

Ilustración C.2 Árboles elementales de la nueva gramática.

Una vez construida la nueva gramática se pueden intentar mininizar el número de árboles auxiliares buscando un isomorfismo que haga equivalentes cada par de árboles. En el ejemplo anterior, los árboles γ_1, γ_2 y γ_3 pueden reducirse a un único árbol haciendo que $R_1 = T_2 = R_2 = T_3 = R_3 = B$.

También es interesante la imposición de restricciones a los nodos de adjunción. Por ejemplo, todo nodo interno que no sea adjuntado puede asociarse a la restricción de adjunción nula (SA).

C.2. Generación de una TAG desde un árbol de dependencias con dos o más espinas

Para construir una gramática TAG a partir de un árbol con varias espinas se debe aplicar los mismos pasos que en la sección anterior pero teniendo en cuenta las siguientes consideraciones en los nodos internos desde los que parten dos ramas independientes.

Dado un nodo padre etiquetado con D_k que tiene al menos dos hijos no hojas tales que en el subárbol de uno existe una hoja hija de un nodo etiquetado con D_k y en el otro no, el primero se interpretará como el resultado de una adjunción que ha divido en dos mitades el árbol elemental asociado a D_k mientras que el segundo deberá ser interpretado como consecuencia de una adjunción en el nodo pie del árbol asociado a D_k.

En el caso de que ninguno de los hijos tengan en su subárbol una hoja hija de un nodo etiquetado con D_k, uno de ellos deberá ser interpretado tal como antes y el otro deberá añadir en algún punto del subárbol una hoja etiquetada con ϵ que sea hija de un nodo etiquetado con D_k.

Ejemplo C.2. Consideremos el árbol de dependencias mostrado en la ilustración C.3. Los nodos etiquetados con D_1 y D_2 junto al nodo raíz (su padre) se corresponden al primer caso expuesto. Mientras que los nodos etiquetados con D_2 y D_3 junto al nodo raíz se corresponden al segundo caso.

En la ilustración C.4 se puede ver los árboles elementales construidos para cada dependencia según lo explicado previamente.

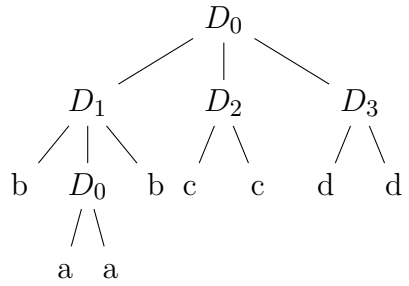

Ilustración C.3 Árbol de dependencias con varias ramas.

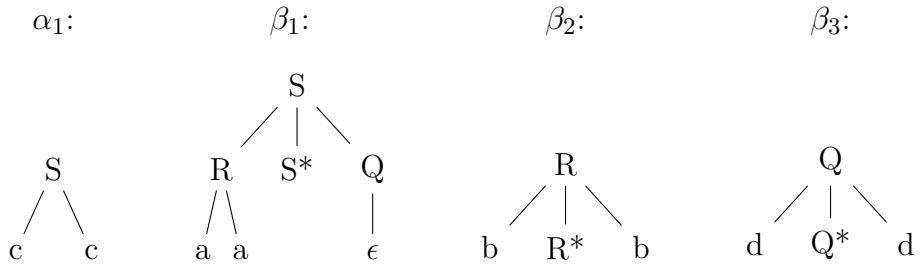

Ilustración C.4 Árboles elementales construidos a partir del árbol de dependencias de la ilustración C.3.

D Pseudocódigo de los algoritmos para generar árboles de dependencias

En este anexo se da una implementación en pseudocódigo de los algoritmos para generar un árbol de dependencias simple y un árbol de dependencias complejo a partir de una cadena con dependencias.

D.1. Generación de un árbol de dependencias simple

En el algoritmo se utiliza los enteros del 0 al m para identificar las dependencias D_i. Para no enmarañar el código supondremos que en la secuencia de dependencias no existen dos dependencias iguales contiguas. En caso contrario habría que agrupar dichas dependencias en una sola de forma que en vez de estar asociada ésta a un único terminal estaría asociada a una lista.

En las siguientes páginas se lista el código y a continuación viene una explicación del mismo.

El algoritmo `crearArbolDeDependenciasSimple` es básicamente el esquema de una búsqueda de la solución con mínimo coste, en este caso el mínimo número de inserciones ficticias, dejando que toda la lógica recaiga sobre la clase `Nodo`.

Puesto que las subsecuencias que se quieren hallar son símetricas (una el reverso de la otra), basta con almacenar una de ellas y el punto de corte en la secuencia original. En `Nodo` estos dos datos se obtienen de `subsecuenciaIzq` e `izq`. En realidad, `izq` es un puntero que se utiliza junto a `dch` para recorrer simultáneamente a izquierda y derecha la secuencia de dependencias intentando hacer coincidir las dependencias. Lo que ocurre es que el valor final de `izq` (menos uno) coincide con el punto de corte.

La idea del método consiste en avanzar los punteros mientras que las dependencias en ambos extremos coincidan. LLegados a un punto en el que no lo hagan, se puede insertar dependencias ficticias o bien a izquierda o bien a derecha hasta hacerlas coincidir. Esto da lugar a la creación desde un nodo en el árbol de búsqueda a dos nodos hijos. Los nodos vivos cuya subsecuencia generada sea menor serán los elegidos hasta que uno de ellos sea solución.

Como ya se mencionó anteriormente no todas las inserciones son posibles. Por tanto, cada vez que se intenta insertar una dependencia en `añadirDependencia` se comprueba que sea posible. Para ello se emplea los atributos `pila`, `leidas`, `encerradas` y `esValido` de `Nodo`. Si se detecta que el nodo ya no puede llevar a una solución válida, `esValido` pasa a valer `falso`. El dato `pila` va almacenando las dependencias leídas hasta encontrar una que ya fue leída (ya fue almacenada en `leidas`). En ese caso se marca añadiendo a `encerradas` todas las dependencias que están en la cima de `pila` hasta encontrar la última vez que se leyó la dependencia.

A partir de `palabra`, `subsecuenciaIzq`, `izq` y `ligaduras` ya se puede construir el árbol de dependencias.

```
algoritmo crearArbolDeDependenciasSimple(
                                  E palabra: vector[0..n-1] de caracteres;
                                  E ligaduras: vector[0..n-1] de enteros;
                                  -- Ligaduras es sinómino de dependencias
                                  S arbolDep: arbol de dependencias)
  clase Nodo { ... }
principio
 Nodo nodo = crearNodo();
 ColaPrioridades cp = crearColaPrioridades();
 cp.añadir(nodo);

 booleano exito = falso;
 mq (not exito and not cp.isVacia()) hacer
   nodo = cp.min();
   cp.elminarMin();

   si (nodo.esSolucion()) entonces
     exito = cierto;
   sino
     Nodo hijo = nodo.generarHijoDch();
     si (hijo != nulo) cp.añadir(hijo);
     hijo = nodo.generarHijoIzq();
     si (hijo != nulo) cp.añadir(hijo);
   fsi;
 fmq;
 si exito entonces
   construirArbol(palabra, ligaduras, nodo.subsecuenciaIzq,
                  nodo.izq-1, arbolDep);
 sino
   lanzar ErrorNoExisteSubsecuecia();
 fsi;
fin
```

```
...
 clase Nodo
 {
   Lista<entero> subsecuenciaIzq;
   Pila<entero> pila;
   Conjunto<entero> leidas;
   Conjunto<entero> encerradas;
   entero izq, dch;
   booleano esValido;

   constructor Nodo()
   {
     subsecuenciaIzq = crearLista();
     pila = crearPila();
     leidas = crearConjunto();
     encerradas = crearConjunto();
     izq = 0; dch = n-1;
     esValido = cierto;
     avanzar();
   }

   metodo avanzar()
   {
     mq (esValido and (izq<=dch) and (ligaduras[izq]==ligaduras[dch])) hacer
       añadirDependencia(ligadura[izq]);
       izq++; dch++;
     fmq;
   }

   metodo compararNodo(Nodo otro) devuelve entero
   {
     dev (este.subsecuenciaIzq.longitud()-otro.subsecuenciaIzq.longitud());
   }

   metodo esSolucion() devuelve booleano
   {
     dev (esValido and izq>dch);
   }
   ...
 }
...
```

```
...
 clase Nodo
 {
   ...
   metodo añadirDependencia(entero dep)
   {
     si (not encerradas.contiene(dep))
       subsecuenciaIzq.añadir(dep);
       si (not leidas.contiene(dep))
         mq (pila.cima() != dep) hacer
           encerradas.añadir(pila.cima());
           pila.pop();
         fmq;
       sino
         leidas.añadir(dep);
         pila.push(dep);
       fsi;
     sino
       esValido = falso;
     fi;
   }
 }

 metodo generarHijoDch() devuelve Nodo
 {
   si (esValido)
     Nodo nodo = este.crearCopia();
     nodo.añadir(ligaduras[izq]);
     nodo.izq++;
     mq (nodo.esValido and (ligaduras[nodo.izq] != ligaduras[nodo.dch])) hacer
       nodo.añadir(ligaduras[izq]);
       nodo.izq++;
     fmq;
     nodo.avanzar();
     dev (nodo.esValido) ? nodo : nulo;
   sino
     dev nulo;
   fsi;
 }
 metodo generarHijoDch() devuelve Nodo { // Simétrico al Izq }
...
```

```
algoritmo construirArbol(E palabra: vector[0..n-1] de caracteres;
                         E ligaduras: vector[0..n-1] de enteros;
                         E subsecuenciaIzq: Lista<entero>;
                         E fin; S arbolDep: arbol de dependencias)
principio
  arbolDep = crearArbolDep();
  Nodo padre = arbolDep.raiz();

  entero indice = 0;
  Iterador<entero> iter = subsecuenciaIzq.iteradorDesdeInicio();
  mq (iter.haySiguiente()) hacer
    entero dep = iter.siguiente();
    Nodo nodo = padre.añadirHijo(dep);
    si (indice <= fin and dep == ligaduras[indice]) entonces
      nodo.añadirHijo(palabra[indice]);
      indice++;
    fsi
    padre = nodo;
  fmq

  iter = subsecuenciaIzq.iteradorDesdeFinal();
  mq (iter.hayAnterior()) hacer
    entero dep = iter.anterior();
    Nodo nodo = padre.añadirHijo(dep);
    si (indice<n and dep == ligaduras[indice]) entonces
      padre.añadirHijo(palabra[indice]);
      indice++;
    fsi
    padre = padre.padre();
  fmq
fin
```

D.2. Generación de un árbol de dependencias complejo

Se van a definir las subclases `SimboloTerminal` y `SimboloNoTerminal` de la superclase `Simbolo`. Ambos tienen en común que están asociados a una dependencia (el entero que la identifica). En cuanto a sus diferencias se tiene que `SimboloTerminal` denota una posición (un carácter) de la palabra original mientras que `SimboloNoTerminal` representa una secuencia (agrupación) de símbolos. Además `SimboloNoTerminal` se especializa en `SimboloNoTerminalHomogeneo`, que contiene secuencias

de símbolos asociados a una misma dependencia. También se empleará una clase `Rango` para representar intervalos.

```
// Los constructores se omiten por simplicidad
clase Simbolo
{
  constante SUPERDEPENDENCIA := nulo;
  entero depId; // A la superdependencia se le asocia el valor nulo
}
clase SimboloTerminal extiende Simbolo
{
  int posicion;
}
clase SimboloNoTerminal extiende Simbolo
{
  Lista<Simbolo> simbolos;
}
clase SimboloNoTerminalHomogeneo extiende SimboloNoTerminal {}
clase Rango
{
  entero primero;
  entero ultimo;
}
```

Por otra parte, se hará uso de una clase llamada `PalabraLigada` que representará la entrada del algoritmo principal. Ésta contendrá una secuencia de símbolos y un diccionario que a cada dependencia usada asocia el intervalo en el que se encuentran sus símbolos. En esta estructura quedará representada la descomposición en grupos de la secuencia de dependencias original. La clase dispone de un constructor que adapta la entrada original al formato de la clase para iniciar la descomposición.

```
clase PalabraLigada
{
  Lista<Simbolo> simbolos;
  Diccionario<entero> rangos;

  constructor Palabra(ligaduras: vector[0..n-1] de enteros)
  {
    para i:=0..n-1 hacer
      añadirSimbolo(crearSimboloTerminal(ligaduras[i]), i));
    fpara;
  }
  metodo añadirSimbolo(Simbolo simbolo)
  {
    entero ultimoIndice = simbolos.tamaño()-1;
    si (ultimoIndice>=0 and (simbolos[ultimoIndice].depId == simbolo.depId)
      SimboloNoTerminalEmpaquetado sntE;
      Simbolo ultSimbolo = simbolos[ultimoIndice];
      si (ultSimbolo esInstanciaDe SimboloNoTerminalEmpaquetado)
       sntE = ultimoSimbolo;
      sino
       sntE = crearSimboloNoTerminalEmpaquetado(simbolo.depId);
       sntE.simbolos.añadir(ultSimbolo);
       simbolos[ultimoIndice] = sntE;
      fsi;
    sino
      simbolos.añadir(simbolo);
      añadirRango(simbolo.depId, ultimoIndice+1);
    fsi;
  }
  metodo añadirRango(entero depId, entero pos)
  {
    si (not rangos.contiene(depId))
      rangos[depId] = crearRango(pos, pos);
    sino
      rangos[depId].ultimo = pos;
    fsi;
  }
  ... // otros métodos que simplifican el acceso (se omiten)
}
```

El algoritmo que construye el árbol de dependencias es el siguiente:

```
algoritmo crearArbolDeDependencias(E palabra: vector[0..n-1] de caracteres,
                                   E ligaduras: vector[0..n-1] de enteros,
                                   S arbolDep: arbol de dependencias)
principio
  si n==0 entonces // Palabra vacía
    arbolDep = crearArbol("S");
    arbolDep.raiz.añadir("epsilon");
  sino
    PalabraLigada pl = descomponerPalabraLigada(crearPalabraLigada(ligaduras));
    // La palabra ligada descompuesta solo contiene un símbolo
    arbolDep = construirArbolComplejo(pl.primerSimbolo());
  fsi;
fin;
```

Como ya se comentó, hay algunos casos en los que un grupo debe ser sustituido por una dependencia igual a la última que abrió su intervalo. Para llevar dicho control se utiliza la clase `MonitorRangos`. En él se van registrando los símbolos leídos y permite consultar cúal es la última dependencia abierta.

```
clase MonitorRangos
{
  PalabraLigada palabraLigada;
  Pila<entero> abiertas;
  Conjunto<entero> cerradas;
  constructor MonitorRangos(PalabraLigada pl) { palabraLigada=pl; }

  metodo registrarSimbolo(entero pos)
  {
    entero depId = palabraLigada.simboloPosicion(pos).depId;
    si (palabraLigada.esPrimerSimboloDependencia(pos))
      abiertas.push(depId);
    fsi;
    si (palabraLigada.esUltimoSimboloDependencia(pos))
      cerradas.añadir(depId);
      mq (not abiertas.estaVacia() and cerradas.contiene(abiertas.cima()))
        abiertas.pop();
      fmq;
    fsi;
  }
  metodo ultimaDependencia() devuelve entero
  {
    dev (not abiertas.esVacia()) ? abiertas.cima() : Simbolo.SUPERDEPENDENCIA;
  }
}
```

El algoritmo `descomponerPalabraLigada` va leyendo subsecuencias definidas por los puntos de corte, aquellas posiciones donde se abrió una dependencia después de cerrarse otra. Al finalizar vuelve a llamarse recursivamente si aún quedan símbolos por agrupar, es decir, hay más de un símbolo en la secuencia generada.

```
funcion descomponerPalabraLigada(PalabraLigada entradaPL) devuelve PalabraLigada
principio
  entero ultimoIndice = entradaPL.tamaño()-1;
  PalabraLigada salidaPL = crearPalabraLigada();
  MonitorRangos monitorRangos = crearMonitorRangos(entradaPL);

  entero pos = leerSubsecuencia(entradaPL, salidaPL, monitorRangos,
                                ultimoIndice, 0);
  mq (pos < ultimoIndice) hacer
    pos = leerSubsecuencia(entradaPL, salidaPL, monitorRangos,
                           ultimoIndice, pos+1);
  fmq;
  si (salidaPL.numSimbolos()==1) entonces
    dev salidaPL;
  sino
    dev descomponerPalabraLigada(salidaPL);
  fsi;
fin;
```

La función `leerSubsecuencia` se encarga de encontrar el siguiente punto de corte y de precalcular en `subsecRango` mediante llamadas a `actualizarSubsecRango` el rango más grande que incluye a dependencias de tipo A en la subsecuencia que se está generando. Después en `procesarSubsec` se comprobará si ese rango o un subrango más pequeño es realmente válido como grupo.

```
funcion leerSubsecuencia(PalabraLigada entradaPL, PalabraLigada salidaPL,
                         MonitorRangos monitorRangos,
                         entero ultIndice, entero pos)
principio
  Rango subsecRango = nulo;
  si (pos < ultIndice) entonces
    booleano cerrada = entradaPL.esUltimoSimboloDependencia(pos);
    actualizarSubsecRango(subsecRango, entradaPL, pos, pos);
    entero indice = pos + 1;
    booleano nuevoSalto = cerrada and entradaPL.esPrimerSimboloDependencia(pos);
    mq (not nuevoSalto and indice<ultIndice) hacer
      cerrada = cerrada or entradaPL.esUltimoSimboloDependencia(pos);
      actualizarSubsecRango(subsecRango, entradaPL, pos, indice);
      indice++;
      nuevoSalto = cerrada and entradaPL.esPrimerSimboloDependencia(pos);
    fmq;
    entero fin = (nuevoSalto) ? indice-1 : indice;
    si (not nuevoSalto) entonces
      actualizarSubsecRango(subsecRango, entradaPL, pos, indice);
    fsi;
    procesarSubsec(entradaPL, salidaPL, monitorRangos, pos, fin, subsecRango);
    dev fin;
  sino
    actualizarSubsecRango(subsecRango, entradaPL, pos, pos);
    procesarSubsec(entradaPL, salidaPL, monitorRangos, pos, pos, subsecRango);
    dev pos;
  fsi;
fin;
```

```
funcion actualizarSubsecRango(ES Rango subsecRango, PalabraLigada entradaPL
                              entero inicio, entero indice)
principio
  Rango rango = entradaPL.rangoDependenciaEnPosicion(indice);
  si (rango.ultimo == indice and rango.primero >= inicio) entonces
    si (subsecRango == nulo) entonces
      subsecRango = crearRango(rango.primero, rango.ultimo);
    sino
      subsecRango.primero = Min(subsecRango.primero, rango.primero);
      subsecRango.ultimo = rango.ultimo;
    fsi;
  fsi;
fin;
```

En `procesarSubsecuencia` se busca primero una posible grupo entre el rango precalculado. Si no existe se copian todos los símbolos de la subsecuencia original. Si existe se copian todos los símbolos situados antes del grupo, se genera un no-terminal con el grupo que se añade a la secuencia de salida y se copian todos los símbolos situados después del grupo. Si el grupo incluye un símbolo asociado a una dependencia de tipo B, se asocia dicha dependencia con el representante. Sino se consulta con `monitorRangos` la última dependencia abierta.

```
algoritmo procesarSubsecuencia(PalabraLigada entradaPL,
                               PalabraLigada salidaPL,
                               MonitorRangos monitorRangos,
                               entero inicio, entero fin, Rango subsecRango)
principio
  Rango rango = buscarGrupo(entradaPL, subsecRango);
  si (rango == null) entonces
    copiarSubsecuencia(entradaPL, salidaPL, monitorRangos, inicio, fin);
  sino
    copiarSubsecuencia(entradaPL, salidaPL, monitorRangos,
                       inicio, rango.primero-1);

    SimboloNoTerminal noTerminal = crearSimboloNoTerminal();
    para i:=rango.primero...rango.ultimo hacer
      monitorRangos.registrarSimbolo(i);
      Simbolo simbolo = enradaPL.simboloPosicion(i);
      si (entradaPL.estaEnRango(simbolo, inicio, fin)) entonces
        noTerminal.depId = simbolo.depId;
      fsi;
      noTerminal.añadir(simbolo);
    fpara;
    si (noTerminal.depId != nulo) entonces
      noTerminal.depId = monitorRangos.ultimaDependencia();
    fsi;
    salidaPL.añadirSimbolo(noTerminal);

    copiarSubsecuencia(entradaPL, salidaPL, monitorRangos,
                       rango.ultimo+1, fin);
  fsi;
fin;
```

La búsqueda de un grupo consiste en tener en cuenta todas aquellas dependencias de tipo A que sólo contienen dependencias de este tipo en su rango o como máximo una de tipo B. Para ello se usa una pila, `abiertas`, donde se van registrando las dependencias abiertas por primera vez y un conjunto, `cerradas`, donde se marcan las dependencias cerradas. Una dependencia de la pila sólo se decarta de ella si está en la cima y fue cerrada. De esta forma aseguramos que todas las dependencias entre el principio y el final de su intervalo también fueron descartadas. Además, comprobaremos que haya como mucho una dependencia de tipo B. Para ello se

almacena y se usa el valor de `marca` que nos indica la última dependencia de tipo B leída.

```
funcion buscarGrupo(PalabraLigada entradaPL, Rango subsecRango)
principio
 si (subsecRango == nulo) dev nulo;
 entero inicio = subsecRango.primero, fin = subsecRango.ultimo;
 Rango rangoGrupo = nulo;

 Pila<Simbolo> abiertas = crearPila();
 Conjunto<entero> cerradas = crearConjunto();

 entero siguienteMarca = nulo;
 entero marca = inicio-1;

 para i:=inicio..fin hacer
   Simbolo simbolo = entradaPL.simboloPosicion(i);
   si (entradaPL.estaEnRango(simbolo, inicio, fin)) entonces
     registrarDependenciaIncluida(entradaPL, abiertas, cerradas, simbolo,
                                  rangoGrupo, marca, pos);
   sino
     // Se ha detectado en la subsecuencia
     // una dependencia que no cumple la condición
     nuevaMarca(siguienteMarca, marca, i);
   fsi;
 fpara;

 dev rangoGrupo;
fin;
```

```
algoritmo nuevaMarca(ES entero siguienteMarca, ES entero marca, entero pos)
principio
  si (siguienteMarca != nulo) entonces
    marca = siguienteMarca;
  fsi;
  siguienteMarca = pos;
fin;
```

```
algoritmo registrarDependenciaIncluida(PalabraLigada entradaPL,
                                       Pila<Simbolo> abiertas;
                                       Conjunto<entero> cerradas;
                                       Simbolo simbolo;
                                       ES Rango rangoGrupo;
                                       entero marca, entero pos)
principio
  si (entradaPL.esPrimerSimboloDependencia(pos)) entonces
    abiertas.añadir(simbolo);
  fsi;
  si (entradaPL.esUltimoSimboloDependencia(pos)) entonces
    cerradas.añadir(simbolo);
    mq (not abiertas.esVacia() and cerradas.contiene(abiertas.cima().depId))
      Rango rango = entradaPl.rangoDependenciaSimbolo(abiertas.cima());
      abiertas.pop();
      si (rango.primero > marca) entonces
        si (rangoGrupo == nulo) entonces
          rangoGrupo = crearRango(rango.primero, rango.ultimo);
        sino
          rangoGrupo.primero = Min(rangoGrupo.primero, rango.primero);
          rangoGrupo.ultimo = Max(rangoGrupo.ultimo, rango.ultimo);
        fsi;
      fsi;
    fmq;
  fsi;
fin;
```

```
algoritmo copiarSubsecuencia(PalabraLigada entradaPL, PalabraLigada salidaPL,
                             MonitorRangos monitorRangos,
                             entero inicio, entero fin)
principio
  para i:=inicio..fin hacer
    monitorRangos.registrarSimbolo(i);
    salidaPL.añadirSimbolo(entradaPL.simboloPosicion(i));
  fpara;
fin;
```

Una vez obtenida la descomposión jerárquica de la entrada en los diferentes grupos, se va construyendo el árbol de dependencias mediante el procesado de cada grupo mediante el primer algoritmo visto.

.

E Manual de usuario del Entorno de Análisis y Visualización

Este anexo contiene una descripción de los pasos necesarios para ejecutar el programa interactivo destinado al análisis y la visualización de las 2-estructuras (y sus estructuras relacionadas) y una guía de uso de la aplicación.

E.1. Requisitos e instalación

Para poder ejecutar el programa sólo es necesario disponer de la máquina virtual de Java 6.0 [17]. Ésta es gratuita y está disponible para múltiples sistemas operativos.

La aplicación se distribuye en un JAR ejecutable y no necesita de ninguna configuración previa. En un sistema Windows basta con hacer doble click sobre el JAR. En un sistema GNU/Linux o derivado se puede ejecutar desde una consola con el siguiente comando (estando situado en el directorio del programa):

```
java -jar TwoStructures.jar
```

E.2. La ventana principal

En la ilustración E.1 se muestra una captura de pantalla del programa. Como se puede observar la ventana está dividida en tres áreas.

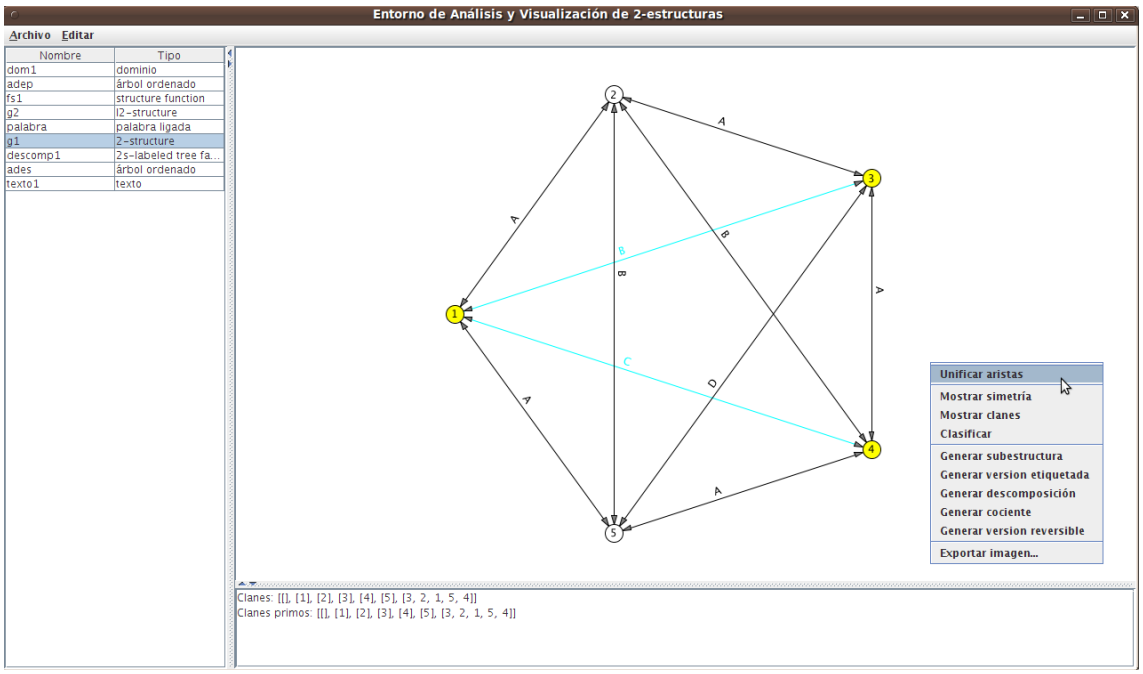

Ilustración E.1 Ventana principal de la aplicación

En el lateral izquierdo está el *espacio de trabajo* donde se listan el nombre, junto al tipo, de las estructuras creadas durante la sesión. Al seleccionar una estructura en concreto las otras dos áreas se actualizan mostrando la información asociada a la estructura.

En la parte superior del lateral derecho se encuentra el visor y editor de la estructura. Y en la parte inferior de ese mismo lateral está situado un cuadro de texto donde se muestran propiedades y otros datos sobre la estructura.

Cada una de las áreas puede ocupar más o menos espacio. Para cambiar el tamaño de un área hay que mantener presionado el botón izquierdo sobre una de las fronteras y arrastrar hasta la anchura o la altura deseada. Si en algún momento se quiere ocultar momentaneamente un área se puede conseguir pulsando sobre las flechas situadas sobre la frontera de cada área.

E.3. Creación, duplicación y borrado de estructuras

Para crear una nueva estructura se debe abrir el menú `Archivo`, `Nuevo` y seleccionar el tipo de estructura deseada. Se abrirá un diálogo donde se pedirán los datos necesarios para crear la estructura. Además se tendrá que introducir un nombre que permitirá distinguir la estructura creada del resto de estructuras del espacio de trabajo.

Una vez creada, la nueva estructura aparecerá en el lateral izquierdo de la ventana donde se podrá seleccionar para visualizarla y/o editarla.

Si se quiere borrar una estructura del espacio de trabajo se debe seleccionar y acceder al menú `Editar`, `Borrar`. De manera similar se puede duplicar una estructura, aunque en este caso se pedirá un nombre que sirva para identificar a la nueva copia.

E.4. Creación y visualización de una 2-estructura

Para crear una 2-estructura lo primero que hay que hacer es definir un dominio. Al crear un nuevo dominio se debe especificar una lista de elementos separados por espacios o comas. Tras crearlo, se debe seleccionar y proceder a la creación de la 2-estructura.

Una vez creada, se puede visualizar en el visor de la derecha con sólo seleccionarla en el espacio de trabajo. Con el visor se puede definir que aristas son equivalentes entre sí (por defecto se crea una clase de equivalencia para cada arista). Para ello, se debe seleccionar las aristas que se desea hacer equivalentes con el botón izquierdo, abrir el menú contextual disponible al apretar el botón derecho y seleccionar la opción `Unificar aristas` (ver la ilustración E.1).

El menú contextual proporciona otras opciones de utilidad en relación con la 2-estructura. Se puede pedir calcular los clanes de la 2-estructura, clasificarla o analizar las posibles simetrías. También es posible generar otras estructuras derivadas a partir de la 2-estructura entre las que se encuentra su *descomposición* (también llamada la *forma* de la 2-estructura), su versión etiquetada o su versión reversible. Además se puede exportar a un fichero la imagen actualmente mostrada.

E.5. Visualización de la forma de una 2-estructura

Al final de la sección E.4 se explica cómo crear la descomposición de una 2-estructura. Una vez creada, puede ser visualizada como el resto de estructuras con el visor del lateral derecho. Este visor se compone, para esta estructura, de dos paneles que permiten verla comodamente (ver la ilustación E.2). El primero muestra su representación en forma de árbol y dibuja las 2-estructuras de menos de cuatro elementos que etiquetan cada uno de sus nodos. El segundo muestra las 2-estructuras con más de tres elementos de aquellos nodos que son seleccionados.

Como el árbol puede tener muchos nodos, el visor permite hacer hacer *zoom* con la ruleta del ratón para reducir o ampliar la imagen.

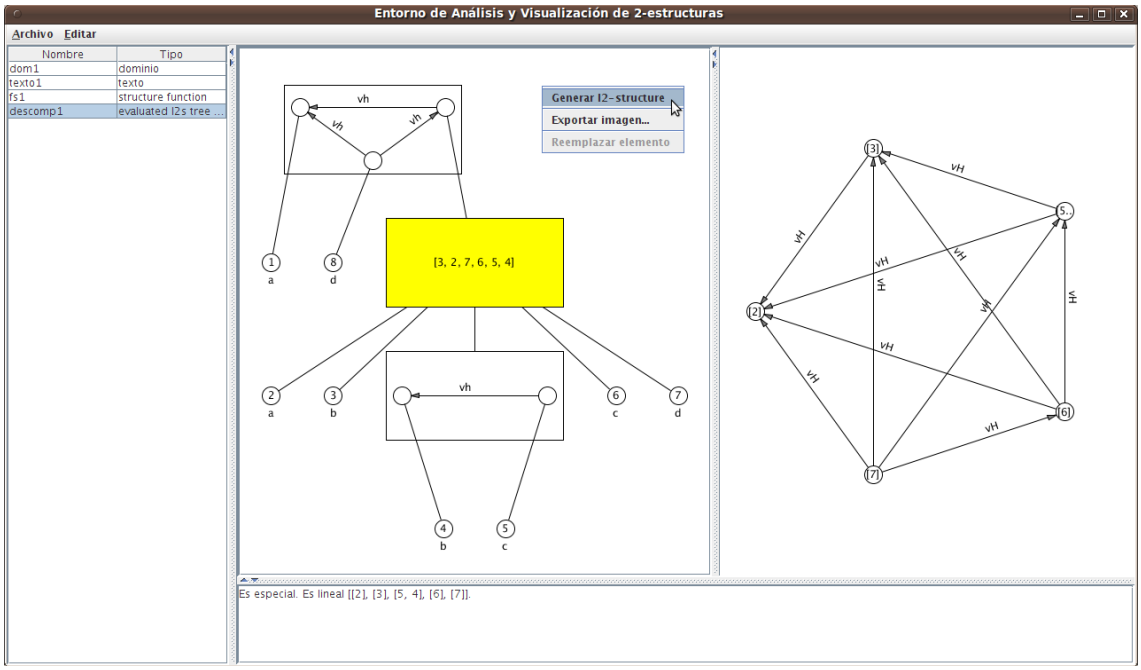

Ilustración E.2 Visualización de la forma de una 2-estructura.

E.6. Creación de una subestructura a partir de una 2-estructura

Para crear una subestructura a partir de una 2-estructura, se debe visualizar esta última en el visor y seleccionar con el botón izquierdo aquellos elementos (nodos) que formarán parte de la subestructura. Después usando el menú contextual, accesible mediante el botón derecho, se debe seleccionar la opción `Generar subestructura`.

E.7. Creación de un texto y de su función estructurada asociada

Para crear un texto lo primero que hay que hacer es definir un dominio tal como se hizo en la sección E.4 para una 2-estructura. Después se puede proceder a crear el texto, para lo cual se deberá indicar la etiqueta asociada a cada elemento y la posición que ocupa cada elemento en cada uno de los dos ordenes lineales que componen el texto. Ver la ilustración E.3.

Una vez creado un texto, se puede seleccionar y generar la función estructurada asociada desde el menú `Archivo`, `Nuevo`. Como una función estructurada es una 2-estructura con una función de etiquetado de sus elementos, ésta puede descomponerse y visualizarse como se explicó en la secciones anteriores.

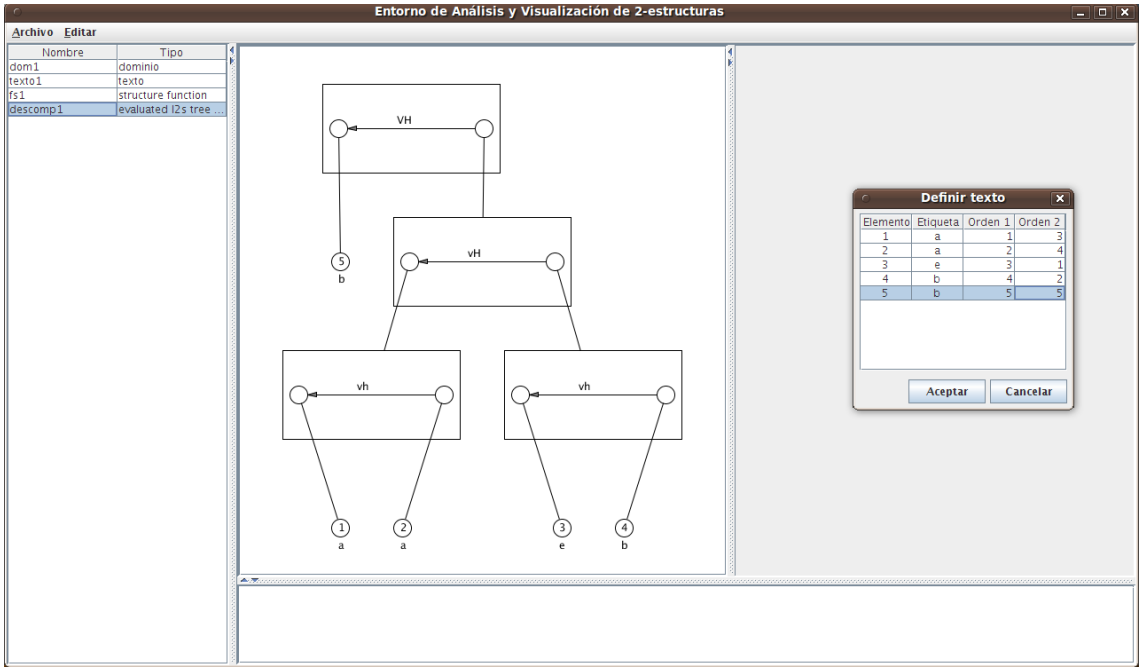

Ilustración E.3 Visualización de la forma de una 2-estructura.

E.8. Creación de una palabra ligada y generación de su árbol de dependencias

El algoritmo explicado en el capítulo 4 sobre la generación de un árbol de dependencias a partir de una cadena con dependencias está disponible a través del programa interactivo.

Para crear una palabra con dependencias hay que seleccionar la opción `Palabra Ligada` del menú `Archivo`, `Nuevo`. A continuación hay que introducir una cadena y después indicar qué símbolos dependen entre sí. Ver la ilustración E.4.

Una vez creada, se puede seleccionar y generar el árbol de dependencias asociado desde el menú `Archivo`, `Nuevo`. También es posible visualizar la jerarquía en la que se agrupa las dependencias antes de calcular el árbol de dependencias final.

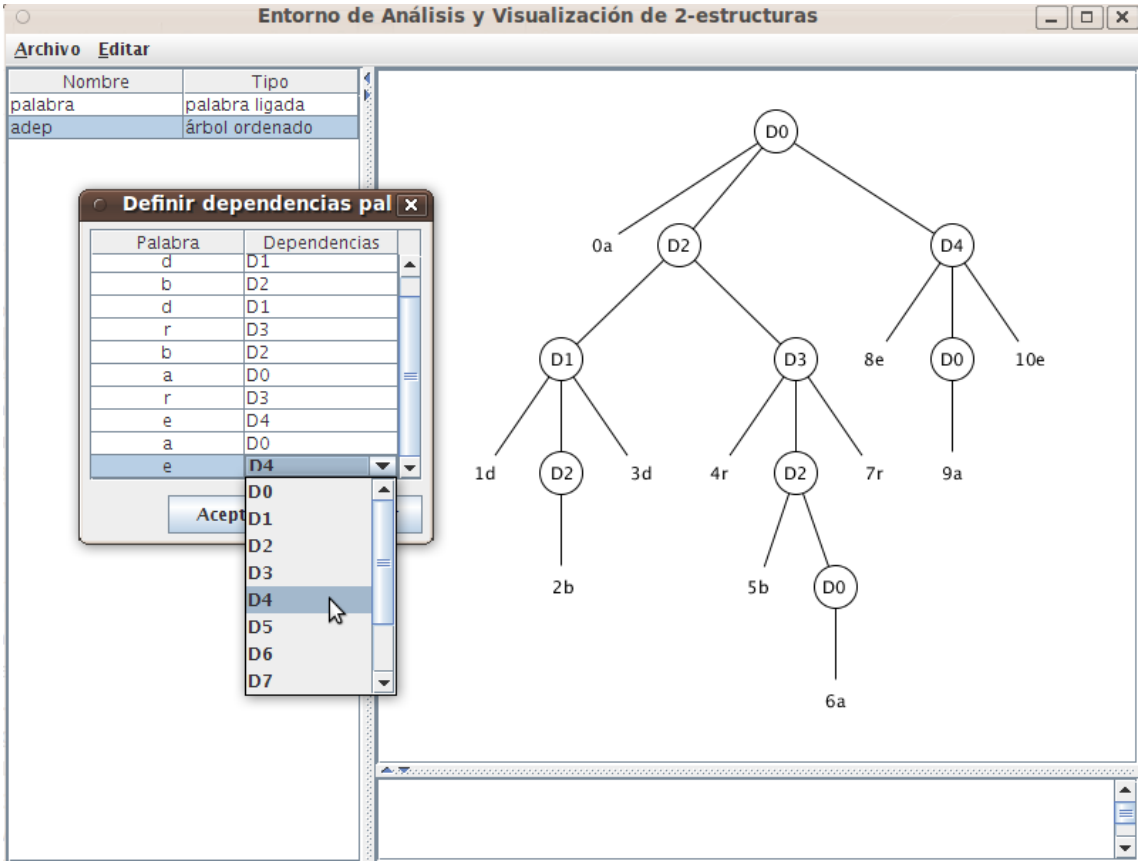

Ilustración E.4 Árbol de dependencias junto a diálogo de creación de una nueva palabra ligada.

E.9. Almacenamiento permanente de las estructuras

El espacio de trabajo de una sesión se puede almacenar en un fichero XML usando el menú `Archivo`, `Guardar como`. Después se puede cargar su contenido de nuevo con el menú `Archivo`, `Abrir`. Si se quiere empezar con un espacio de trabajo vacío se debe seleccionar la opción `Espacio de trabajo` del menú `Archivo`, `Nuevo`,

El formato del fichero XML es bastante intuitivo. A modo de muestra se lista a continuación la definición de una función estructurada.

```
<?xml version="1.0" encoding="UTF-8"?>
<workspace>
  <variables>
    <structured_function name="sf1">
      <elements>
        <e value="a">1</e>
        <e value="b">2</e>
        <e value="c">3</e>
      </elements>
      <labeled_two_structure>
        <domain>
          <e>1</e><e>2</e><e>3</e>
        </domain>
        <class label="A">
          <edge src="1" dst="2" />
          <edge src="1" dst="3" />
          <edge src="3" dst="1" />
        </class>
        <class label="B">
          <edge src="2" dst="1" />
          <edge src="2" dst="3" />
          <edge src="3" dst="2" />
        </class>
      </labeled_two_structure>
    </structured_function>
  </variables>
</workspace>
```

Bibliografía

[1] A. Ehrenfeucht and G. Rozenberg. "Theory of 2-structures. Part I: Clans, basic subclasses, and morphisms". Theor. Comput. Sci., 70:277–303, 1990.

[2] A. Ehrenfeucht and G. Rozenberg. "Theory of 2-structures. Part II: Representation through labeled tree families". Theor. Comput. Sci., 70:305–342, 1990.

[3] A. Ehrenfeucht and G. Rozenberg. "Angular 2-structures". Theor. Comput. Sci., 92:227–248, 1992.

[4] A. Ehrenfeucht and G. Rozenberg. "T-Structures, T-functions, and texts". Comput. Sci., 116:227–290, 1993.

[5] A. Ehrenfeucht, T. Harju and G. Rozenberg. "The theory of 2-estructures. A Framework for Decomposition and Transformation of Graphs". World Scientific, 1999.

[6] D. Jurafsky, J.H. Martin: "Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition". Prentice Hall, 2009.

[7] M. A. Alonso. "Interpretación tabular de autómatas para lenguajes de adjunción de árboles", Tesis doctoral, Universidad de La Coruña, España, 2000.

[8] K. Vijay-Shanker, D.J. Weir. "The Equivalence Of Four Extensions Of Context-Free Grammars". Mathematical Systems Theory, 27:511–546, 1994.

[9] J.M. Vergés. "La no-independencia del contexto de los lenguajes naturales". Lenguajes naturales y lenguajes formales: actas del XII congreso de los lenguajes naturales y lenguajes formales, pp. 87—102, 1996.

[10] C. Pollard. "Generalized Phrase Structure Grammars, Head Grammars and Natural Language". PhD thesis, Stanford Ubiversity, 1984.

[11] D.J. Weir. "Characterizing Mildly Context-Sensitive Grammar Formalisms". PhD thesis, University of Pennsylvania, 1988. Available as Technical Report MS-CIS-88-74 of the Department of Computer and Information Sciences, University of Pennsylvania.

[12] D.J. Weir, K. Vijay-Shanker and A.K. Joshi. "The relantionship Between Tree Adjoining Grammars And Head Grammars". Proceedings of the 24th annual meeting on Association for Computational Linguistics, 67–74, 1986.

[13] J.Q. Walker II. "A node-positioning algorithm for general trees". Softw. Pract. Exper., 20:685–705, 1990.

[14] C. Buchheim, M. Jünger and S. Leipert. “Improving Walker’s Algorithm to Run in Linear Time”, Graph Drawing. Lecture Notes in Computer Science, 2528:347–364, 2002.

[15] A.K. Joshi. “Processing Crossed and Nested Dependencies: An automaton Perspective on the Psycholinguistic Results”. Language and Cognitive Processes, 1989.

[16] Bernard A. Galler and Michael J. Fischer. “An improved equivalence algorithm”. Communications of the ACM, 7:301–303, 1964.

[17] Plataforma Java SE. http://www.oracle.com/technetwork/java/javase/ (Último acceso: 24/08/2010).

[18] JGraph. http://www.jgraph.com (Último acceso: 24/08/2010).

[19] JUNG. http://jung.sourceforge.net/ (Último acceso: 24/08/2010).

[20] Eclipse. http://www.eclipse.org/ (Último acceso: 24/08/2010).

[21] Subversion. http://subversion.apache.org/ (Último acceso: 24/08/2010).