Diagramática: Una Formalización para la Construcción de Lenguajes Visuales

Federico C. Repond

Director:
Pablo E. Martínez López

Codirector:
Gabriel Baum

Universidad Nacional de La Plata

28 de febrero de 2006

Índice general

a mi familia y amigos...

Agradecimientos

Afortunadamente, el desarrollo de esta tesis ha llegado a su punto de culminación. El deseo implícito de finalizar la tesis no implica el deseo de abandonar el hábito de la investigación y la búsqueda constante de perfeccionamiento, sino que abre nuevos horizontes, nuevas metas y nuevos desafíos. Por este motivo quiero agradecer a dos personas (quienes fueron primero profesores y luego amigos) que han sido fundamentales para despertar en mí este incesante apetito por conocimiento. Guido, quién me ayudó a dar mis primeros pasos en esta fascinante ciencia y me alentó a tomar el camino que hoy llega a un punto de profunda satisfacción, y Fidel, mi mentor a lo largo de estos años de estudios en la UNLP, quién fue fuente incesante de conocimiento, apoyo y por sobre todas las cosas, amistad. También quiero agradecer a toda mi familia Ana, Eduardo, Verónica, Otilia, Nélida, Pedro y Héctor por el apoyo que me brindaron a lo largo de estos años junto con mis amigos. Ellos saben que no fue fácil tantos años de exilio de mi querida Rosario. Finalmente y más importante a mi esposa Carina, que me conoció durante mis años de estudio y me brindó el soporte necesario para poder llegar a este punto. Ella más que nadie sufrió y se alegró junto a mí a lo largo de mi carrera compartiendo logros y fallos.

También es necesario hacer un apartado especial para los muchos compañeros y amigos que me acompañaron a lo largo de este proceso; para aquellos docentes y no docentes que hacen de su trabajo una forma de vida y con dedicación ayudan a limpiar el ambiente decadente que hoy reina fuera de la

Capítulo 1

Introducción

Las interfaces gráficas son el modelo estándar de interacción hombre–máquina. Hoy en día cualquier tarea imaginable que requiera este tipo de interacción puede ser desarrollada mediante la ayuda de una herramienta visual adecuada. Uno de los puntos claves en el cual reside la importancia de los lenguajes visuales es que permiten a expertos en distintas áreas, en general escasos y de costo elevado, expresar sus conocimientos en su propio lenguaje de forma natural, obteniendo una respuesta sobre su trabajo en forma instantánea y en muchos casos también código ejecutable o algún tipo de estructura intermedia en forma automática, maximizando de esta forma su productividad. Sólo restringiéndonos al campo de las ciencias de la computación, existen cientos de diagramas diferentes (E-R [Che76b, Che76a], UML [RJB99], PERT [Ste71], Redes de Petri [Rei85, MMS92], etc.), y análogamente distintos editores para éstos. Si se hace extensivo este análisis a otras áreas, el número es más impresionante aún. Existe una gran diversidad de editores de diagramas para distintos propósitos. Ejemplos claros son el *Visio* (VT [1]) o el *Rational Rose* (RR [2]). El primero es un editor de propósito general que permite crear una gran variedad de diagramas y ser extendido por

[1] `http://www.microsoft.com/office/visio/`

[2] `http:// www.rational.com/products/rose/`

el usuario (experto en programación) para la construcción de nuevos diagramas. El segundo es una muestra clara de editores de diagramas especializados con la posibilidad de generar un modelo subyacente – en este caso un modelo que permite la generación de código en forma automática a partir del mismo. Ambos productos son potentes herramientas en su área de aplicación, pero pueden ser mejorados notablemente utilizando las herramientas teóricas adecuadas.

El VT permite la creación diagramas complejos y la posibilidad de agregar nuevos tipos de diagramas; sin embargo carece de la posibilidad de adosar un modelo al diagrama o validar los mismos en forma sencilla y transparente; requiere un conocimiento avanzado de detalles internos del producto y una API propietaria que no siempre está abierta al usuario o está acompañada por una documentación apropiada, sin mencionar además que el diseñador del lenguaje visual debe poseer conocimientos avanzados de programación. Estas falencias se deben a que generalmente este tipo de herramientas no proveen un marco adecuado que estandarice y motive la asignación de semántica a los diagramas. En este tipo de herramientas u otras de propósito específico, la asignación de semántica al diagrama es programada de manera ad-hoc dependiendo del requerimiento específico.

Los del tipo del RR, por otro lado, permiten la creación de diagramas que al mismo tiempo generan un modelo subyacente y permiten generar, por ejemplo, código Java; sin embargo están orientados a un tipo específico de diagramas. Más aún, distintas versiones del mismo producto eran necesarias para trabajar con C++ [Str95], Smalltalk [GR83], etc. Además probablemente no encontraremos una versión adecuada si nuestro lenguaje no es considerado lo suficientemente popular para la compañía que desarrolla el producto.

Otra característica importante de la cual carecen la mayoría de los editores de diagramas es la posibilidad de validar los diagramas construidos, ya sea una vez terminada la edición o durante el proceso de construcción. Tam-

poco ninguna de las herramientas actuales permiten que dichas reglas de validación y asignación de semántica sean definidas o modificadas en forma sencilla y natural por el usuario, si acaso lo permiten. Estas carencias en las herramientas actuales se deben en parte a la falta de un formalismo que permita la especificación en forma clara y precisa de la estructura y semántica de los lenguajes visuales. A pesar de la diversidad de diagramas existentes, todos comparten ciertas características comunes: directa o indirectamente todos los diagramas presentan cierta estructura. Con esto se vislumbra que esta propiedad puede ser explotada para la formalización de la estructura de los diagramas, y la construcción de un editor genérico para la manipulación de lenguajes visuales basado en dicha formalización.

En este trabajo propongo una formalización basada en gramáticas de atributos [Pos99] que permite capturar en forma precisa la estructura de los diagramas y su simultánea evaluación (p. ej., construir/evaluar automáticamente el modelo semántico de los diagramas durante la construcción de los mismos). Comencé este trabajo en el marco de un proyecto colaborativo entre la UNLP y la PUC (Pontificia Universidad Católica do Rio de Janeiro) y reporté algunos resultados preliminares en [MLPR97] y [PRML99]. En dicha instancia introduje la idea básica de utilizar gramáticas para capturar la estructura de los diagramas y estudié además distintas alternativas para la formalización. Además, ese proyecto culminó con la construcción de un prototipo funcional que mostraba la importancia práctica de la investigación previamente mencionada.

En esta tesis reporto el trabajo mencionado y brindo un análisis más comprensivo y extenso del uso de gramáticas para la descripción de los lenguajes visuales. Por otro lado, y a fin de demostrar la importancia práctica, desarrollo un editor que utiliza los conceptos expresados, mostrando de esta forma cómo el diseño y prototipación de lenguajes visuales se vé simplificada en forma significativa. Además el trabajo está acompañado de una serie de ejemplos prácticos que apoyan el desarrollo.

1.1. Introducción a los Lenguajes Visuales

Es sabido que el sistema visual humano está preparado para procesar información presentada en varias dimensiones. Las representaciones textuales, predominantes en el área de la tecnología de la información hasta hace unos años, no hacen uso extensivo de este hecho. Ciertos problemas complejos son expresados en forma más simple con la ayuda de gráficos o diagramas, esto es, con información en más de una dimensión [PB99]. Una de las mayores ventajas (y quizás la más importante) que introdujeron los lenguajes de alto nivel fue la posibilidad de utilizar una herramienta más afín al tipo de problema que enfrentaban los desarrolladores de software. De esta manera el programador podía enfocarse en mayor medida a la resolución de su problema específico y no en los detalles internos de la máquina que los ejecutaba [Che71] . De esta misma manera los lenguajes visuales pueden ser aún más expresivos que los lenguajes textuales. Mediante la utilización de un lenguaje visual adecuado un desarrollador puede expresar su comprensión del problema en un nivel de abstracción mayor. Más aún, éstos permiten, en algunos casos, a expertos en determinadas áreas totalmente ajenas a la tecnología de la información expresar su conocimiento sobre el problema que intentan resolver sin necesidad de un programador como intermediario. Aún programadores expertos se ven beneficiados con el uso de herramientas visuales.

El éxito de los lenguajes visuales se vé con mayor énfasis en el área de la enseñanza, donde los entornos gráficos han ayudado al aprendizaje de la programación. Esto se debe a que si bien los lenguajes textuales son potentes y expresivos, también son difíciles de aprender y dominar. El uso de lenguajes visuales que utilicen construcciones conocidas y luego sean transformadas a código en algún lenguaje textual ayudan enormemente al aprendizaje de los mismos. También, como hemos mencionado, acercan la programación a personas con amplio conocimiento en determinados dominios pero sin la experiencia necesaria en el arte de la misma.

Hoy en día existen numerosas herramientas visuales para la construcción

de interfaces del usuario para distintos lenguajes. También podemos contar un sinfín de herramientas que asisten al diseño de alto nivel (diagramas E-R, UML, redes de Petri), tareas que son en algunos casos difíciles de desarrollar sin la asistencia de una herramienta adecuada, y en otros casos, tediosas.

Los lenguajes visuales pueden ser "ejecutados" de diferentes maneras. Estos pueden ser compilados a código nativo utilizando la sintaxis (visual) y semántica del lenguaje, pueden ser interpretados, o pueden producir código para un lenguaje textual determinado. La utilización de lenguajes visuales no intenta sustituir a los lenguajes textuales existentes sino actuar en combinación con éstos y potenciarlos. Sin embargo a pesar de la importancia y ventajas de los lenguajes visuales que hemos visto, existen aún varios problemas abiertos. Los mismos presentan serias restricciones en cuanto a su escalabilidad y a su nivel de abstracción. Generalmente están enfocados a un propósito específico, en contrapartida a los textuales que son en, muchos casos, de propósito general. Esto se evidencia en la falta de herramientas adecuadas para muchos de los problemas existentes. Por ejemplo, programadores OCaml [RV98], O'Haskell [Nor99] o Mondrian [MC97] encuentran una variedad de herramientas para la construcción de diagramas UML pero ninguno de éstos genera código para su lenguaje. Es más, muchos de los editores gráficos existentes no brindan mayor valor agregado que programas de dibujo simples como el *Paintbrush* para los usuarios de Windows o el *Kontur* para los de Linux. Editores más sofisticados que los nombrados generalmente sólo son más convenientes porque facilitan el proceso de edición con una mayor cantidad de construcciones visuales y herramientas, pero no suelen brindar más que esto.

Estas deficiencias mencionadas se deben en mayor medida a la falta de editores visuales que se valgan de los formalismos existentes para su construcción. Otra de las carencias encontradas es la falta de herramientas que asistan en la implementación de estos editores visuales facilitando la definición de la sintaxis y semántica de los lenguajes visuales diseñados.

1.2. Trabajos Relacionados

Hoy en día no existen muchas herramientas que asistan al diseñador de lenguajes visuales en su tarea. Los lenguajes de programación textuales se beneficiaron de meta-lenguajes formales para su definición y esto permitió su proliferación. Ejemplos comunes de estas herramientas son, por ejemplo, la EBNF (forma normal Backus Naur extendida) que permite al diseñador de un lenguaje textual especificar su sintaxis sin ambigüedades, o las gramáticas de atributos que permiten asignar semántica a los mismos. Estas herramientas teóricas fueron la base para la construcción de diversas herramientas prácticas que permitieron en gran medida automatizar parte de la construcción de compiladores e intérpretes.

En los últimos años se han presentado una gran cantidad de formalismos con la meta de permitir la descripción y *parsing* de lenguajes visuales. En general, estos formalismos utilizan distintos tipos de gramáticas para dicho propósito. Actualmente sobresalen dos enfoques: el primero se basa en la utilización de relaciones que describen una sentencia visual como un conjunto de objetos gráficos y un conjunto de relaciones entre ellos [FPS+96, RS96, TVC94]; el segundo, basado en atributos, describe una sentencia como un conjunto de objetos gráficos adornados con atributos [CLOT97, Mar94, Gol91, Wit92]. La mayoría de los formalismos basados en gramáticas proveen soporte para parsear objetos gráficos sin orden preestablecido, esto es, donde los mismos son agregados libremente al diagrama (edición libre) y luego son procesados sin ningún orden en particular. Ejemplo de estas gramáticas son las Picture Layout Grammars [Gol91], Relation Grammars [CGN+91], Constraint Multiset Grammars [Mar94] y Positional Grammars [CLOT97]. Generalmente las herramientas basadas en estos formalismos realizan, en el mejor de los casos, una enumeración bottom-up de los objetos del diagrama durante el proceso de parsing. Para reducir el espacio de búsqueda y brindar métodos más eficientes se han definido subclases de estos formalismos que permiten construir variantes predictivas de los parsers.

1.3. Contribución

Este trabajo estudia y explora la formalización de los lenguajes visuales; con este propósito brinda una formalización alternativa a las existentes para la descripción de los mismos (Cap. 2). Además del estudio, esta herramienta teórica fué utilizada para el desarrollo de su aplicación práctica: la rápida prototipación de lenguajes visuales. El resultado del desarrollo culminó con la construcción de un prototipo de editor parametrizable que facilita notablemente la experimentación con lenguajes visuales (Cap. 3).

1.4. Organización de la Tesis

A esta altura ya se ha establecido la importancia de los diagramas y lenguajes visuales en general y la motivación del desarrollo de un formalismo para éstos. El trabajo estará organizado de la siguientes manera: en el Capítulo 2 se comienza con una introducción a los lenguajes formales que mostrará conceptos básicos que luego serán utilizados en la definición del formalismo presentado en este trabajo. Las personas ya familiarizadas con estos temas podrán pasar directamente a la Sección 2.2 donde se presentan las *gramáticas de diagramas*. Dicha sección comienza con una breve introducción y definición informal para luego pasar a la definición formal. En el Capítulo 3 se presenta un editor parametrizable mediante archivos XML basado en la herramienta teórica presentada. Dicho editor intentará mostrar la importancia práctica de los resultados alcanzados. Luego, en el Capítulo 4 se definirán a manera de ejemplo dos gramáticas para distintos lenguajes visuales. Durante este desarrollo se verá el proceso completo involucrado en el diseño de un lenguaje visual para un domino específico. Por último se brindarán las conclusiones y los interrogantes que quedaron abiertos para continuar la investigación.

Capítulo 2

Gramáticas de Diagramas

El estudio formal de lenguajes se vió ampliamente favorecido por del desarrollo de herramientas que permitiesen su descripción de manera precisa. Las gramáticas fueron la herramienta utilizada desde los orígenes del estudio de los lenguajes formales para este propósito. Los *lenguajes libres de contexto* (LLCs) son de gran importancia en el área de las ciencias de la computación porque pueden ser utilizados para la definición de lenguajes de programación. Las gramáticas libres de contexto (GLCs) [HU80] son utilizadas para describir a los LLCs. Una GLC está compuesta por un conjunto de variables llamadas *noterminales* que se definen recursivamente mediante reglas llamadas *producciones* a partir de otros noterminales y de símbolos constantes denominados *terminales*. Al definir un lenguaje, las GLC permiten determinar que "palabras" (combinaciones de símbolos terminales) son válidas para cierta categoría sintáctica. Las gramáticas de atributos (GAs) [Pos99, VSK89, SSK00] son una extensión de las GLCs que agregan un conjunto de atributos a cada símbolo de la gramática y un conjunto de ecuaciones que determinan el valor de estos atributos a cada producción, conservando de esta forma la simplicidad de las GLCs pero con la ventaja de poder capturar información contextual. Se ha probado extensivamente el poder expresivo de las GAs para la especificación de lenguajes de programación [ASU85]. A lo

largo de este capítulo y el resto de este trabajo se presentarán las *gramáticas de diagramas* (GDs) – una variación de las GAs – y se mostrará cómo las mismas son utilizadas para la especificación de lenguajes visuales de la misma manera que sus precursoras son utilizadas para la especificación de lenguajes textuales. En la siguiente sección se brindará una introducción informal a las gramáticas y lenguajes formales. Esto nos brindará el marco adecuado para introducir formalmente las GDs en la siguiente sección. Finalizaremos el capítulo con dos ejemplos reducidos de lenguajes visuales, uno para expresiones regulares y otro para diagramas de Entidad-Relación. Esto permitirá clarificar y asentar, mediante ejemplos concretos, los conceptos introducidos a lo largo del capítulo.

2.1. Introducción a los Lenguajes Formales

Como primer paso brindaremos en esta sección una introducción a las gramáticas y lenguajes formales. Esta introducción no pretende ser completa ni abarcativa de todos los conceptos relacionados. Existen excelentes fuentes introductorias al tema como [Lin01] y [HU80], entre otras, para aquellos lectores interesados en una introducción extensiva y completa del tema.

El lenguaje natural – Castellano en nuestro caso – es impreciso y lleno de ambigüedades; las descripciones informales en castellano son generalmente vagas y carentes de precisión, y la estructura de las oraciones extremadamente compleja. Esto hace que un estudio riguroso de los lenguajes naturales sea complicado. Por este motivo se elije trabajar con una estructura mucho más simple de lenguaje conocida como lenguaje formal, cuya estructura puede estudiarse con precisión. Considerando, a modo de ejemplo, una versión muy simplificada del lenguaje castellano como un lenguaje formal, podemos dar una gramática que expresa la forma de las oraciones (si bien usaremos el lenguaje natural para ejemplificar la noción de lenguaje formal por la familiaridad de cualquier lector con el mismo, el estudio del lenguaje natural

requiere lemas más poderosos). Una gramática dice cuándo una oración particular está bien formada o no. Por ejemplo, una típica regla gramatical es "una oración está compuesta por un sujeto seguido de un predicado". En forma más concisa

$$\langle oración \rangle \quad \rightarrow \quad \langle sujeto \rangle \ \langle predicado \rangle$$

Por supuesto aquí no finaliza la definición de la gramática de oraciones del lenguaje Castellano. Se debe proveer definiciones para las nuevas construcciones (categorías sintacticas) introducidas $\langle sujeto \rangle$ y $\langle predicado \rangle$

$$\begin{array}{rcl} \langle sujeto \rangle & \rightarrow & \langle artículo \rangle \ \langle sustantivo \rangle \\ \langle predicado \rangle & \rightarrow & \langle verbo \rangle \end{array}$$

Si se asocian las palabras "la" y "el" con $\langle artículo \rangle$, "mujer" y "avión" con $\langle sustantivo \rangle$, y "camina" y "vuela" con $\langle verbo \rangle$, las oraciones "la mujer camina" y "el avión vuela" están bien formadas de acuerdo a la gramática definida. Este ejemplo ilustra claramente cómo se puede definir un concepto general a partir de otros más simples. A partir de un concepto de alto nivel $\langle oración \rangle$ se fueron definiendo sucesivamente nuevas construcciones del lenguaje hasta llegar a términos irreducibles. La generalización de estas ideas llevan a la definición formal de la gramática.

2.1.1. Definición de Gramática

Una gramática G se define como una tupla con 4 componentes

$$G = (V, T, S, P)$$

donde

- V es un conjunto finito de variables llamadas *noterminales*,

- T es un conjunto finito de símbolos *terminales*,
- $S \in V$ es un símbolo especial llamado variable *inicial*,
- P es un conjunto finito de *producciones*.

Se asumen V y T no vacíos y disjuntos. Las producciones (reglas de reescritura) son el centro de una gramática; especifican cómo se transforma (reescribe) una secuencia de palabras en otra, definiendo de esta forma el lenguaje asociado a la misma. Las producciones de una gramática tienen la forma $x \rightarrow y$, donde x es un elemento de $(V \cup T)^+$ e y uno de $(V \cup T)^*$, significando * 0 o más repeticiones y $^+$ 1 o más. Luego, dada una palabra w con $w = uxv$, decimos que la producción $x \rightarrow y$ es aplicable y podemos utilizarla para reescribir w como un nueva palabra $z = uyv$. Esto se escribe como $w \Rightarrow z$ y se interpreta como z *es derivado de* w. La clausura transitiva y reflexiva de $\Rightarrow$, esto es la aplicación de sucesivas derivaciones $w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$, se escribe como $w_1 \stackrel{*}{\Rightarrow} w_n$. Luego se define al conjunto $L(G) = \{w \in T^* : S \stackrel{*}{\Rightarrow} w\}$ como el lenguaje generado por G.

Gramáticas Libres de Contexto

Para facilitar su estudio, las gramáticas fueron categorizadas por Noam Chomsky [Lin01], por lo que se conoce a esta jerarquía como jerarquía de Chomsky. En esta jerarquía se definen restricciones sobre las producciones de la gramática, por lo cual se puede definir una relación de inclusión entre éstas, y por ende una relación de inclusión entre los lenguajes generados por las mismas.

En este trabajo se limitará el repaso sólo a una clase especial de gramáticas, las *gramáticas libres de contexto* (GLC), debido a su relevancia en el mismo. Se dice que una gramática $G = (V, T, S, P)$ es libre de contexto si todas las produciones de P tienen la forma $A \rightarrow x$, donde $A \in V$ y $x \in (V \cup T)^*$. Luego, se dice que un lenguaje L es libre de contexto sii existe una gramática

libre de contexto G tal que $L = L(G)$. El nombre "libre de contexto" se deriva del hecho de que la sustitución de una variable del lado izquierdo de una producción se puede realizar siempre que ésta aparezca en una frase, o sea, no depende de los símbolos que aparecen en el contexto. La siguiente gramática es un ejemplo de una GLC para describir expresiones aritméticas:

$G = ([E], [+, *, (,)], E, P)$ donde P está compuesto por

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

Luego $2 * (1 + 1)$ pertenece a $L(G)$ ya que se puede derivar de la siguiente forma:

$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \stackrel{*}{\Rightarrow} 2 * (1 + 1)$

Como puede intuirse, las producciones pueden aplicarse en diferente orden produciendo el mismo resultado final. Una forma alternativa de mostrar derivaciones, sin importar el orden en el cual son aplicadas las producciones, es por medio de un *árbol de derivación*.

Un árbol de derivación es un árbol ordenado, donde los labels de los nodos son los noterminales del lado izquierdo de la producción y sus hijos son los terminales/noterminales del lado derecho de la misma. En la Fig. 2.1 se observa el árbol de derivación para $2 * (1 + 1)$ donde a las hojas se les asignaron los valores numéricos 2, 1 y 1 respectivamente y a los nodos los operadores habituales de multiplicación y suma. Al proceso que determina el árbol de derivación correspondiente a un string del lenguaje se lo denomina *parsing*.

Formalmente, se define que un árbol ordenado (con respecto a los nodos en cada nivel) es un árbol de derivación para una gramática libre de contexto $G = (V, T, S, P)$ sii cumple las siguientes propiedades:

1. La raíz tiene como label a S.

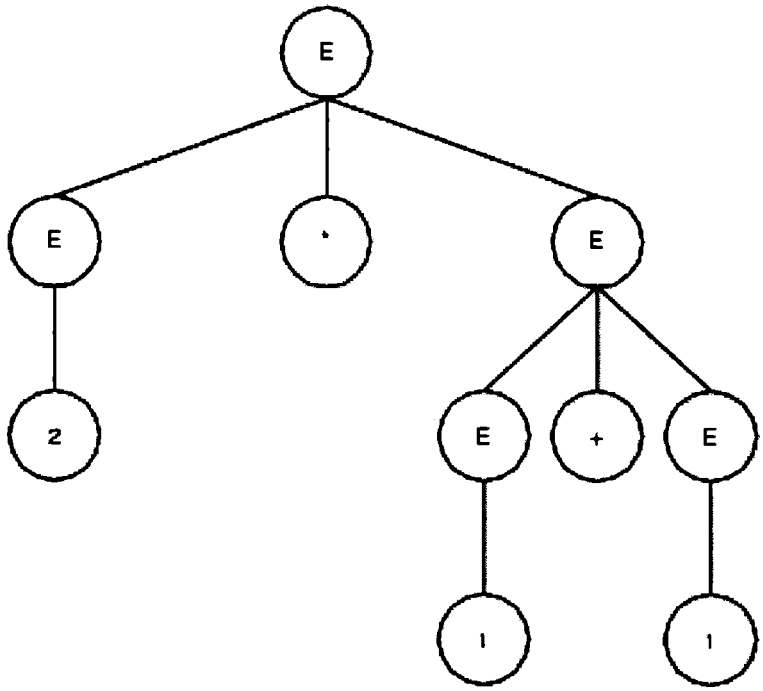

Figura 2.1: Arbol de derivación para $2 * (1 + 1)$.

2. Cada hoja tiene un label de $T \cup \{\lambda\}$.

3. Cada nodo interior tiene un label de V.

4. Si un nodo tiene un label $A \in V$, y sus hijos son (de izquierda a derecha) $a_1, a_2, ..., a_n$, entonces P contiene una producción $A \rightarrow a_1 a_2 ... a_n$.

5. Una hoja con nombre λ no tiene hermanos, esto es, un nodo con un hijo nombrado λ no tiene otros hijos.

Un árbol *parcial* de derivación es uno donde se cumplen las propiedades 3, 4 y 5, pero 1 no necesariamente se cumple y en 2, cada hoja tiene un label de $V \cup T \cup \{\lambda\}$. La concatenación de símbolos que se encuentran en las hojas del árbol de derivación, de izquierda a derecha (habitualmente nombrado como "frontera del árbol"), es el resultado del árbol de derivación. El concepto de árbol de derivación y árbol parcial es súmamente importante en este trabajo ya que es la estructura intermedia utilizada en la representación de diagramas.

Hasta el momento nos hemos referido a los aspectos sintácticos de los lenguajes. Un segundo rol de las gramáticas tiene que ver con el significado de los lenguajes, por ejemplo determinando el significado de los strings. En particular la forma en que los strings del lenguaje son derivados utilizando la gramática es utilizado por el compilador de lenguajes de programación para determinar el significado de los programas. Los árboles de derivación nos permiten relacionar los strings reconocidos por una gramática con su significado. El significado de los strings del lenguaje puede ser explicado asociando significado a cada nodo del árbol de derivación.

Existen casos donde este proceso de parsing genera distintos árboles de derivación para una misma palabra; esto se conoce como *ambigüedad* y se dice que la gramática es *ambigüa*. En la Fig. 2.2 se pueden observar dos árboles de derivación para la expresión $1 - 2 - 3$. Claramente, si se usa el árbol de derivación para interpretar el significado de las expresiones (de

hecho los compiladores lo utilizan) ambos árboles tienen significado diferente: $(1 - 2) - 3 = -4$ y $1 - (2 - 3) = 2$. Las gramáticas ambigüas suelen ser problemáticas para la construcción de compiladores, aunque en la mayoría de los casos éstas pueden transformarse en una gramática sin ambigüedades que genere el mismo lenguaje, o bien utilizar técnicas como la asignación de precedencia a los operadores.

Gramáticas de Atributos

Una GA consiste de una GLC aumentada con reglas semánticas para la definición de atributos. Una gramática de atributos (GA) permite agregar "significado" a una *palabra* en un lenguaje libre de contexto, asignándole valores a los atributos de los nodos del árbol de derivación de dicha palabra. Las gramáticas de atributos son un formalismo comúnmente utilizado para asignar la semántica estática de los lenguajes textuales de programación; por ejemplo, la semántica que puede ser evaluada durante el proceso de compilación. Este formalismo es utilizado por herramientas para la generación automática de código para parsers LALR(1) [ASU85], como el *yacc*.

Una GA es una GLC con las siguientes extensiones:

- cada símbolo de la gramática (terminal o noterminal) tiene asignado un conjunto de atributos.
- toda producción posee un conjunto de ecuaciones de atributos que describen las dependencias y las reglas de computación de los atributos asignado al símbolo de la gramática.

Cada ecuación define los valores de los atributos de una producción como una función aplicada a otros atributos de la producción. Estas funciones que asignan valores a los atributos se denominan funciones semánticas y suelen ser especificadas mediante alguna variante de lenguaje funcional sin efectos laterales. Si bien puede permitirse que estas funciones tengan efectos laterales,

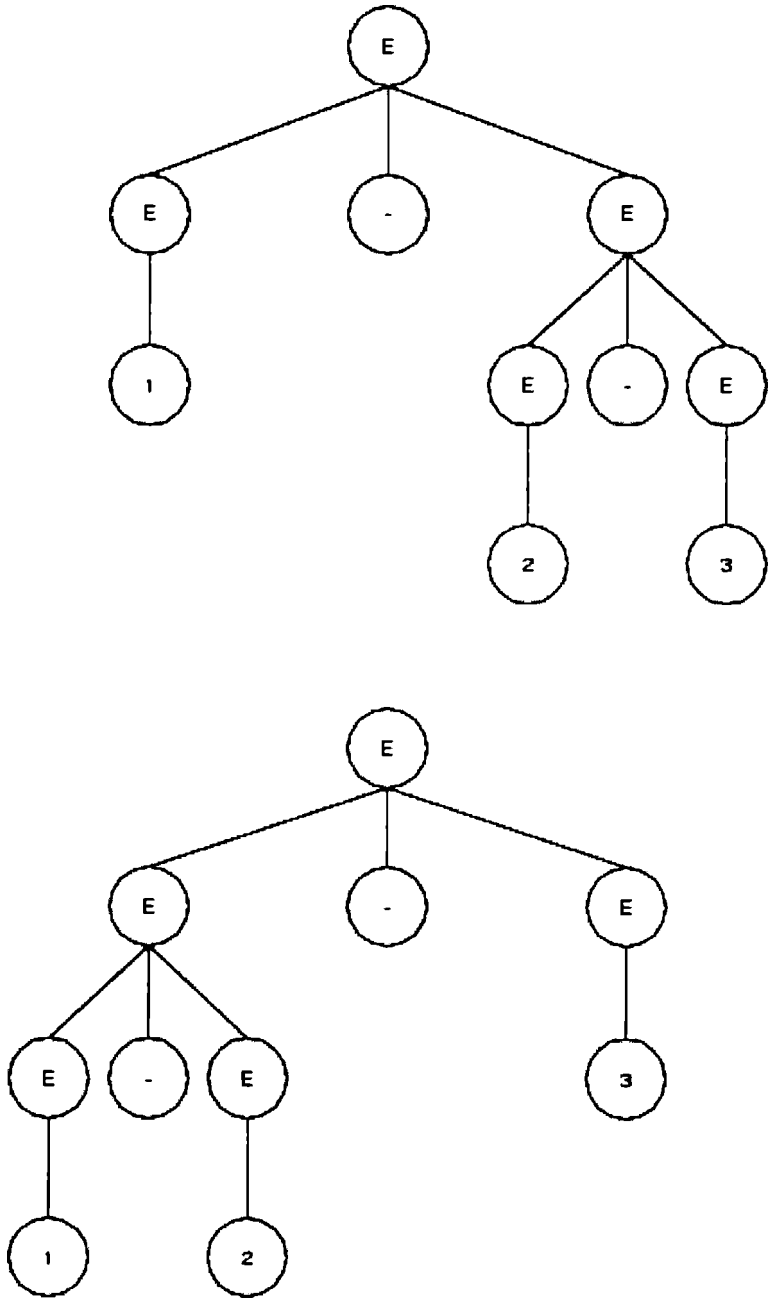

Figura 2.2: Arboles de derivación para : $1 - 2 - 3$.

esto implica que se debe conocer el orden de evaluación de los atributos (o el resultado puede ser impredecible). Los atributos están divididos en dos clases disjuntas, *atributos heredados* y *atributos sintetizados*. Cada ecuación define los valores para los atributos sintetizados del lado izquierdo de la producción y los valores para los atributos heredados del lado derecho de la ecuación. El término sintetizado y heredado se refiere a la evaluación de los atributos en el árbol de derivación correspondiente. Los atributos sintetizados de un nodo son computados a partir de los valores de los hijos, mientras que los atributos heredados de un nodo u hoja son computados a partir de atributos del padre. Esto puede plantearse como que los atributos sintetizados *suben* en el árbol de derivación en tanto que los heredados *bajan* en el árbol de derivación. En la gramática de la Fig. 2.3 se puede ver claramente ejemplos de dichos atributos. Las ecuaciones de las producciones 1, 2, 3, 4 y 5 definen valores de los atributos sintetizados *seq*, *pos* y *env* de los noterminales del lado izquierdo de la producción, en tanto que la ecuación de la producción 1, define también el valor del atributo heredado *env* del noterminal *APPS* en el lado derecho de la producción.

La semántica de las GAs es similar a la semántica de las CFGs, esto es, la evaluación de una GA genera un árbol de derivación; la diferencia que poseen es que los nodos del árbol están adornados con atributos cuyo valor es el resultado de la evaluación de las funciones semánticas de los atributos de la gramática. Como ejemplo de una GA se mostrará una gramática que mapea dos secuencias, una definiendo identificadores y la otra secuencia utilizando estos identificadores, a una lista de enteros que representa el índice del identificador en la definición de los mismos. Por ejemplo en la definición:

let [*a*, *b*, c]

in [*a*, c, *c*, *a*]

es mapeada a:

[0, 2, 2, 0]

En el ejemplo de la Fig. 2.3 se utiliza un lenguaje similar a *Haskell*

$$
\begin{array}{lcl}
lookup & :: & String \rightarrow [(String,\ Int)] \rightarrow Int \\
lookup\ id\ ((id', idx) : env) & = & \textbf{if}\ (id == id')\ \textbf{then} \\
 & & \quad idx \\
 & & \textbf{else} \\
 & & \quad lookup\ id\ env \\
lookup\ id\ [] & = & error\ \text{"}Error : invalid\ identifier.\text{"}
\end{array}
$$

Cuadro 2.4: Definición de la función *lookup*.

[HJW+92] para definir y dar tipo a las acciones semánticas debido a su carácter declarativo y ausencia de efectos laterales. Si bien las definiciones de funciones utilizando Haskell son explicativas por sí mismas, los interesados en obtener información adicional sobre él pueden referirse a [HPF99] o [Tho96].

Para denotar atributos heredados se utilizará el símbolo ↑ y para atributos sintetizados el símbolo ↓. En la definición de la gramática de atributos se ha utilizado el símbolo "." para seleccionar el atributo de un noterminal y subíndices para diferenciar la ocurrencia múltiple de noterminales en la misma producción. En el ejemplo, *ID* será tratado como un terminal donde el valor de su atributo *name* es el valor del terminal que asumiremos asignado por un analizador léxico. La definción de la función auxiliar *lookup* se puede observar en el Cuadro 2.4.

En el ejemplo se puede observar cómo la lista de identificadores y su correspondiente posición dentro de la misma se calcula por medio del atributo sintetizado *env* asociado al noterminal *DECLS*. Por cada identificador introducido en la declaración se agrega un elemento al entorno, representado como una lista de tuplas *(identificador* × *posición)*, y el atributo *pos* que lleva la cuenta de identificadores definidos es incrementado. Este entorno es heredado por el noterminal *APPS* y es utilizado por la función semántica *lookup* en la computación del atributo sintetizado *seq* para calcular la posición del identificador que ocurre en la aplicación dentro de la declaración. Por último en la producción *ROOT* el atributo *seq* contiene el resultado del cálculo de la secuencia de ocurrencias sintetizandolo a partir del valor acumulado en

$$
\begin{array}{rl}
ROOT & \{\downarrow seq :: Int\} \\
DECLS & \{\downarrow pos :: Int, \downarrow env :: [(String, Int)]\} \\
APPS & \{\uparrow env :: [(String, Int)], \downarrow seq :: [Int]\} \\
ID & \{\downarrow name :: String\}
\end{array}
$$

$$
\begin{array}{rcll}
\textbf{ROOT} & \rightarrow & < \textbf{let} > DECLS < \textbf{in} > APPS \ \{ & (1) \\
 & & \quad APPS.env := DECLS.env & \\
 & & \quad , ROOT.seq := APPS.seq\} & \\
\textbf{DECLS} & \rightarrow & ID \ \{ & (2) \\
 & & \quad DECLS.pos := 0 & \\
 & & \quad , DECLS.env := [(ID.name, DECLS.pos)]\} & \\
\textbf{DECLS} & \rightarrow & DECLS <,> ID \ \{ & (3) \\
 & & \quad DECLS_0.pos := DECLS_1.pos + 1 & \\
 & & \quad , DECLS_0.env := & \\
 & & \qquad (ID.name, DECLS_0.pos) : DECLS_1.env\} & \\
\textbf{APPS} & \rightarrow & ID \ \{ & (4) \\
 & & \quad APPS.seq := [lookup\, ID.name\, APPS.env]\} & \\
\textbf{APPS} & \rightarrow & ID <,> APPS \ \{ & (5) \\
 & & \quad APPS_1.env := APPS_0.env & \\
 & & \quad , APPS_0.seq := & \\
 & & \qquad APPS_1.seq{+}{+}[lookup\, ID.name\, APPS_0.env]\} &
\end{array}
$$

Figura 2.3: GA para el cálculo de secuencias.

APPS. La Fig. 2.4 muestra un ejemplo de un árbol parcial de derivación generado a partir de esta gramática. En dicho árbol se puede observar cómo se van evaluando atributos heredados y sintetizados de cada nodo.

2.2. Gramáticas de Diagrama

Esta sección contiene el principal aporte de este trabajo: la definición de una variante de las gramáticas de atributos para la descripción de los lenguajes visuales (LVs) llamada *gramática de diagramas* (GDs). Se mostrará como muchos de los conceptos simples, válidos para lenguajes textuales, se reiteran en la formalización de LVs.

Introducción a las Gramáticas de Diagrama

Las gramáticas de diagramas permiten describir en forma precisa la estructura de los lenguajes visuales. Un lenguaje visual L determina el conjunto de diagramas D_L que pertenecen al lenguaje, de la misma forma que un lenguaje textual determina que secuencia de *palabras* pertenecen al lenguaje. Una GD es básicamente una GA extendida para soportar la descripción de lenguajes visuales, salvo que en lugar de describir sólo palabras, describen estructura; o sea, si bien las GD son estructuralmente similares a las GA, su interpretación será diferente.

Modelo Semántico y Evaluación

Una de las diferencias más notorias que se encuentra en un lenguaje visual descripto por una GD con respecto a un lenguaje textual, es que en lugar de describir árboles describirán DAGs; dado que un lenguaje visual presenta información en 2 o más dimensiones, los métodos habituales de parsing no son aplicables. En un lenguaje textual generalmente se define un proceso de evaluación que dada una secuencia de palabras, verifica si ésta pertenece o no al lenguaje, generándose el árbol de derivación correspondiente durante

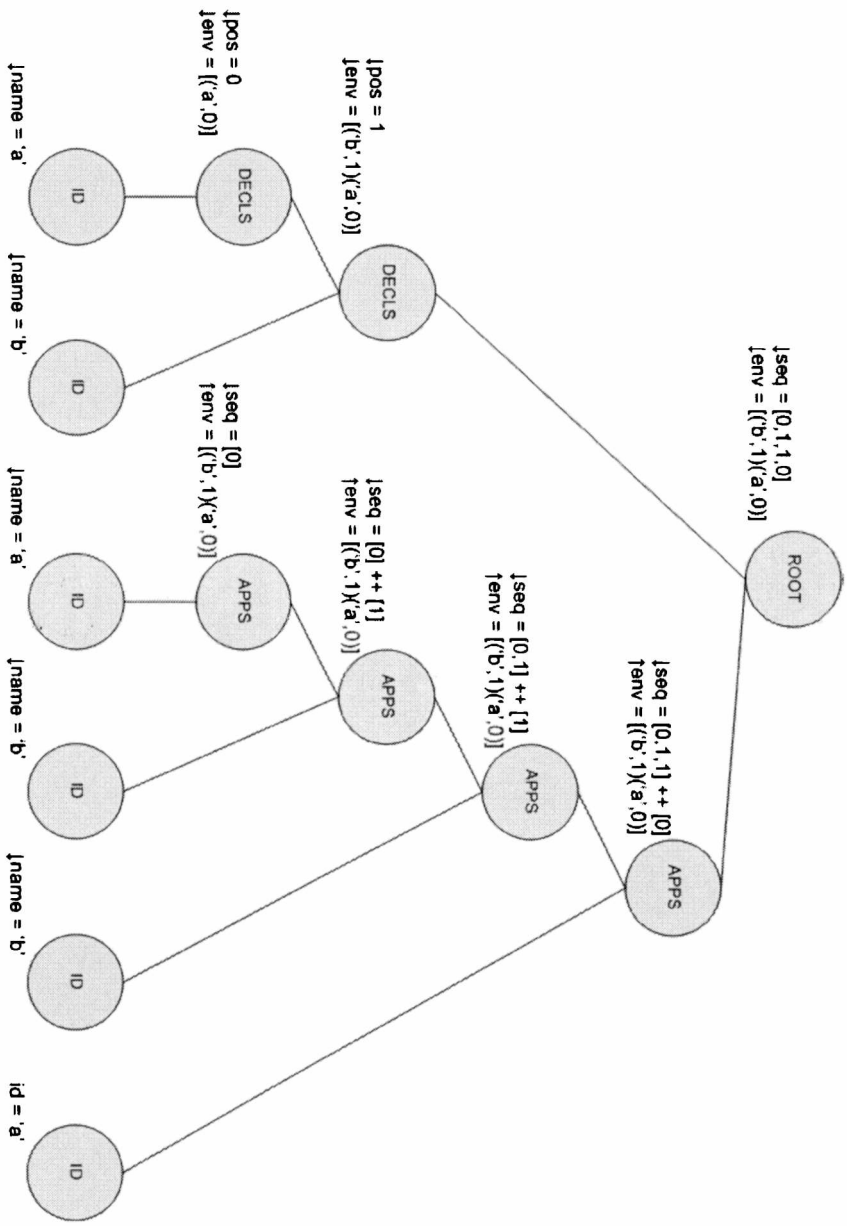

Figura 2.4: Arbol de derivación.

este proceso. En este trabajo el proceso elegido para el reconocimiento o evaluación de lenguajes visuales es totalmente diferente: en lugar de dado un diagrama determinar si éste es válido o no de acuerdo a la GD, el diagrama se va construyendo a partir de las reglas (producciones) impuestas por la GD, con la consecuente construcción del DAG de derivación. De esta forma los diagramas construídos siempre pertenecen al lenguaje visual descripto por la GD. Este es un concepto fundamental que será utilizado extensivamente en el desarrollo práctico de las GDs.

Relación y Diferencias con Lenguajes Textuales

Por otra parte, no tiene sentido pensar en una "palabra" asociada a un árbol de derivación, por lo que la interpretación de estos será diferente. Para ello la GD posee atributos obligatorios especiales que determinan la apariencia gráfica de los diagramas pertenecientes al LV. Esta es otra característica distintiva de las GD ya que en un lenguaje textual el valor de un terminal está determinado por el texto de cada palabra y el de los noterminales por la concatenación de estas palabras. La semántica del lenguaje visual está determinada por atributos opcionales al igual que en los lenguajes textuales descriptos por las gramáticas de atributos por lo cual un diagrama puede tener o no semántica asignada según lo decida el diseñador del LV. Durante el transcurso de la sección se brindará la definición formal de las gramáticas de diagramas.

Organización

El desarrollo de este capítulo permitirá al lector tener una idea inicial para la implementación de una herramienta para la manipulación de lenguajes visuales. Finalmente se mostrarán unos ejemplos concretos que intentarán terminar de asentar los conceptos que se irán introduciendo.

2.2.1. Definición

Una gramática de diagramas es una tupla $\langle T, N, S, P, A, \gamma, \rho, Lexp \rangle$ donde:

- T es el conjunto de terminales; representan diagramas atómicos, esto es, no pueden subdividirse y conforman las hojas del DAG de derivación correspondiente.

- N es el conjunto de noterminales; representan diagramas compuestos y conforman los nodos internos del DAG de derivación. Un nuevo sub-DAG de derivación se construye agrupando un conjunto de terminales o noterminales según las producciones definidas.

- S es el conjunto de símbolos iniciales de la gramática; son noterminales distinguidos que sirven como regla de validación para los diagramas construidos. El proceso de creación de un diagrama es válido si, y sólo si, los DAGs de derivación generados tienen como raíz alguno de los noterminales contenidos en S. Esto puede expresarse como:
  "sea $D \in G_D$ un diagrama, decimos que D es un diagrama válido con respecto a la gramática G_D sii $\forall\ A_D\ :\ root(A_D) \in S$, donde *root* es la función que dado un DAG nos retorna la raíz del mismo y A_D denota algún DAG de derivación asociado al diagrama D.

  Cuando se introduzca la semántica de una GD se verá por qué se hace referencia a DAGs de derivación y no se habla de uno solo.

- P es el conjunto de producciones, con dos tipos bien definidos. En el primer grupo se encuentra un tipo especial que sólo asigna valores iniciales a los atributos de los terminales; se denominan *producciones terminales*. El segundo contiene un conjunto de producciones llamadas *producciones noterminales* que especifican cómo pueden componerse los diagramas a partir de otros (definiendo de esta manera qué diagramas

pueden construirse y cómo), y también las ecuaciones que definen los valores de los atributos asociados a la producción.

- *Lexp* es el lenguaje de expresiones que permite definir los atributos de la gramática. La idea en la inclusión de *Lexp* en la gramática es dejar absoluta libertad para definir el lenguaje para las expresiones de los atributos. Usualmente este puede ser algún subconjunto de un lenguaje de programación estándar (como Haskell o Java), extendido para referenciar símbolos de la gramática.

- A es el conjunto de atributos de la GD. Los atributos acarrean información contextual acerca del diagrama. Existen 3 conjuntos de atributos: A_V, A_C, $A_D \subset A$ con $A_V \neq \emptyset$ donde:

  A_V: es el conjunto de *atributos visuales* del diagrama descripto por la GD. Los mismos son obligatorios y describen la apariencia visual del diagrama.

  A_C: son llamados *atributos de validación* de la GD. Imponen restricciones que deben satisfacerse para que una producción pueda ser aplicada. Una función de validación se define como:
  $c_n : Lexp \rightarrow Bool$,
  que asigna valor a cualquier atributo $n \in A_C$, siendo $Bool = \{\texttt{true}, \texttt{false}\}$.
  Ante la ausencia de estos atributos, esto es $A_C = \emptyset$, se interpreta que las validaciones siempre se satisfacen.

  A_D: es el conjunto de *atributos específicos del dominio*. Estos atributos son definidos por el usuario y son utilizados para dar semántica al modelo subyacente del lenguaje visual especificado por la gramática – e.g., un atributo que contenga el código de un programa PERL para reconocer la expresión regular resultante en un editor de expresiones regulares o un programa para la ejecución de

una simulación de una red de Petri. No existen restricciones para estos atributos más allá de las que puedan imponer la semántica del lenguaje visual que se está describiendo.

- γ es una función $T \cup N \rightarrow A_n$, donde $A_n \subseteq A$, que asigna un conjunto de atributos a cada terminal y noterminal de la gramática.

- ρ es una función de evaluación para las expresiones de los atributos. Tanto el lenguaje de las expresiones como las reglas de evaluación deben ser definidas por el usuario. Generalmente esta función está dada por la semántica del lenguaje elegido.

2.2.2. Semántica y Evaluación de las GD

Como hemos mencionado, el resultado del proceso de *parsing* difiere de los árboles ordenados estándar generados a partir de las GAs en la descripción de un lenguaje textual. En el caso de las GDs la estructura resultante es un conjunto de *grafos dirigidos acíclicos* (DAG, [AHU83]), con los atributos correspondientes adosados a cada nodo. El nodo raíz de cada DAG debe corresponder a un símbolo s inicial de la gramática tal que $s \in S$, para satisfacer las restricciones de la misma. La diferencia en el modelo semántico de las GDs con respecto a las GAs es que permiten compartir (sub)diagramas. Más adelante, en los ejemplos brindados en este capítulo, se muestra el uso y necesidad de esta característica. En este trabajo sólo se estudió la utilización de atributos sintetizados, el estudio del uso de atributos heredados en la GDs quedó como linea de investigación para trabajos futuros.

Como ya hemos mencionado, debido a la naturaleza de los diagramas (información en 2 o más dimensiones), un proceso de parsing – un proceso que dado un diagrama, determina si es válido o no generando el DAG correspondiente – es difícil de definir. En su lugar se puede definir un proceso de construcción del DAG asociado. Este proceso se puede describir mediante tres operaciones básicas:

Creación de una hoja: una hoja del DAG corresponde a un terminal simple. La creación de una nueva hoja se produce aplicando una producción terminal y esta acción asigna valores iniciales a los atributos de las hojas, entre ellos su apariencia visual (obligatoria).

Agrupamiento de DAGs: un conjunto de DAGs se pueden agrupar en un solo DAG bajo una raíz común. Esta operación se realiza aplicando una producción noterminal. Esta acción dispara la computación de los atributos del nodo.

Descomposición de DAG: un DAG puede descomponerse en sus DAG componentes. La raíz del DAG original es eliminada y habrá un nuevo DAG por cada hijo de ésta. Esta operación es similar a la de deshacer comúnmente encontrada en los editores visuales.

La operaciones descriptas manipulan directamente la sintaxis abstracta del diagrama y se llevan a cabo si, y sólo si, los atributos de validación de la producción se satisfacen. El proceso de construcción descripto garantiza que todos los diagramas generados son válidos. Más aún, las gramáticas ambiguas dejan de ser un problema dado que en cada paso de la construcción el usuario selecciona explícitamente qué producción aplicar, desambiguando de esta forma la gramática en forma explícita., Este proceso de construcción será la clave de una herramienta visual que asista la construcción de diagramas – el *Editor Parametrizable* (ver Capítulo 3) presentado en este trabajo es una implementación de esta idea. Este editor permite al usuario interactuar, eligiendo en cada paso qué producción aplicar. Dada la naturaleza de las GD respecto del mecanismo de parsing, esta herramienta es imprescindible para que las GDs puedan ser aplicables en la práctica.

2.3. Ejemplos de Gramáticas de Diagramas

Para clarificar y ejemplificar la utilización de las GDs en la especificación de lenguajes visuales se comenzará con dos ejemplos muy sencillos. Esto nos permitirá explorar las características salientes del formalismo presentado. El primer ejemplo mostrará una gramática sencilla para la edición de expresiones regulares. El segundo mostrará un caso ejemplo más complicado (esto no significa complejo) que explora las principales características de las GD que quedaron pendientes del primer ejemplo; para este propósito utilizaremos diagramas de Entidad-Relación.

2.3.1. Expresiones Regulares

En esta sección se definirá un lenguaje visual para expresiones regulares (REGEX) [HU80]. El Cuadro 2.5 muestra una gramática para dicho propósito. Existe un solo terminal (*Símbolo*) y cuatro noterminales: *Regex* (que es también símbolo inicial de la gramática), *Concatenación*, *Alternativa* y *Repetición*. En este ejemplo, los únicos atributos que interesan son los que guardan la REGEX resultante. Otros atributos, por ejemplo los que definen la apariencia gráfica de la REGEX, fueron omitidos dado que el ejemplo pretende enfocarse sobre la estructura de la gramática. Las expresiones de los atributos (el componente *Lexp* de la definición) son código *Haskell* [Tho96] extendido con un mecanismo para referenciar otros atributos, en un estilo similar al que encontramos en *yacc* [LMB92]. Mientras que ρ son las ecuaciones del ejemplo.

En la Fig. 2.5 se puede ver una instanciación de la GD para expresiones regulares. En el lado izquierdo del mismo se observa una posible representación gráfica para la REGEX, teniendo en cuenta que ésta puede ser modificada mediante la manipulación de los atributos gráficos de los terminales y no terminales de la gramática. A la derecha del mismo se ve el correspondiente árbol de derivación con el correspondiente atributo semántico "*regex*" que

Terminales:		{*Símbolo*}	
Noterminales:		{*Regex, Concatenación, Alternativa, Repetición*}	
Símbolo inicial:		{*Regex*}	
Producciones:		{	
Símbolo	→	{*nombre = requestDialog "Nombre del Símbolo:"; ...*}	**(1)**
Regex	→	*Símbolo* {*regex = $1.regex; ...*}	**(2)**
Regex	→	*Alternativa* {*regex = $1.regex; ...*}	**(3)**
Regex	→	*Repetición* {*regex = $1.regex; ...*}	**(4)**
Regex	→	*Concatenación* {*regex = $1.regex; ...*}	**(5)**
Concatenación	→	*Regex Regex* {*regex = "(" ++ $1.regex ++ $2.regex ++ ")"; ...*}	**(6)**
Alternativa	→	*Regex Regex* {*regex = "(" ++ $1.regex ++ " " ++ $2.regex ++ ")"; ...*}	**(7)**
Repetición	→	*Regex* {*regex = "(" ++ $1.regex ++ ")"*; ...*}	**(8)**
		}	

Cuadro 2.5: Una DG para la construcción de Regex.

acompaña a cada nodo; el mismo contiene un *string* que representa la expresión regular que fuera "dibujada". Al igual que los atributos visuales, dicho atributo semántico puede ser modificado o pueden agregarse nuevos para reflejar la semántica deseada.

El proceso de construcción se lleva a cabo de la siguiente forma: en primer lugar se dibujan dos terminales *Símbolo* con valores "a" y "b" asignados a los atributos *nombre* de cada uno de ellos, para lo cual se aplica la producción terminal (1). Luego aplicando la producción (2) cada terminal es promovido a *Regex*. Este paso es necesario para luego aplicar la producción (7) y crear una *Alternativa*, pero no implica ningún cambio gráfico ni semántico. En este punto, luego de aplicar la producción (7), el atributo semántico *regex* contiene el *string* que representa una alternativa: "$a|b$". Otra vez la alternativa es promovida a un noterminal *Regex* mediante la producción (3). Luego aplicando las producciones (8) y (4) respectivamente construimos una *Repeticion* y nuevamente una *Regex* dado que éste es el símbolo inicial de nuestra gramática. Como resultado de esta manipulación vemos que la raíz del DAG generado lleva la expresión regular: "$(a|b)^*$".

En varios puntos se ha tenido que aplicar la producción intermedia que crea un noterminal *Regex* para luego aplicar las producciones que generan los noterminales *Alternativa* y *Concatenación*, o como paso final para que el nodo raíz del DAG asociado sea el símbolo inicial de la gramática. Estos

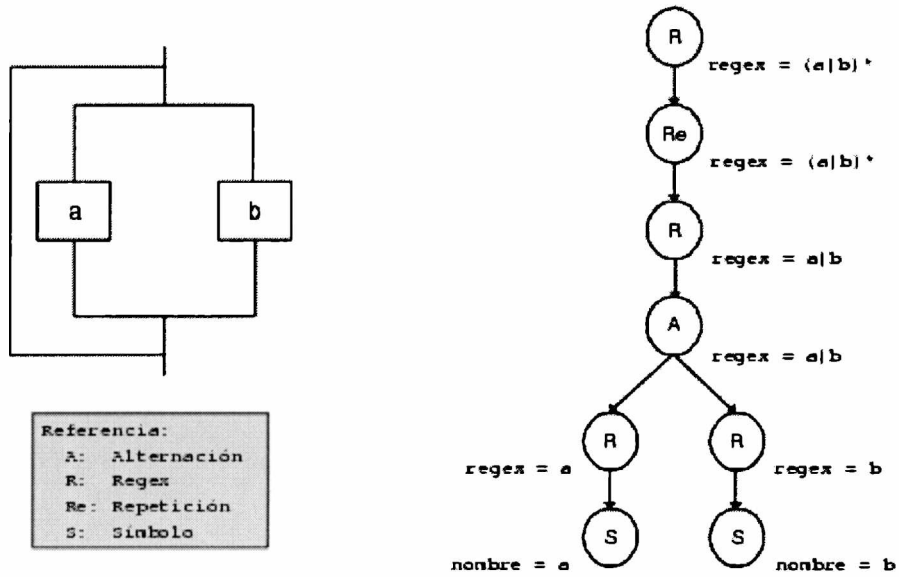

Figura 2.5: Ejemplo de diagrama con su correspondiente DAG.

pasos intermedios podrían evitarse aumentando la gramática con producciones del estilo "*Alternativa* → *Símbolo Símbolo*" entre otras. Por otro lado la aplicación de dichas producciones podrían aliviarse contando con un editor "inteligente" que resuelva automáticamente las producciones intermedias.

En este ejemplo se va vislumbrando cómo, fácilmente, mediante la manipulación de los valores de los atributos, se puede modificar la semántica del diagrama. Por ejemplo podríamos modificar el atributo *regex*, o agregar uno nuevo, tal que en lugar de contener el *string* con la representación textual de la REGEX, se genere automáticamente el código para un programa que chequee la validez de cierta entrada con respecto a la REGEX editada – esto se muestra en el Cuadro 2.6, donde se genera un parser basado en combinadores de parsers para Haskell [Fok95]. Un subconjunto de estos combinadores utilizados en el ejemplo es presentado en la Fig. 2.6. En esta GD se ha modificado el atributo semántico *regex* para que guarde el parser generado a partir de la manipulación de la gramática que permite determinar si una palabra

Terminales:		{*Símbolo*}	
Noterminales:		{*Regex, Concatenación, Alternativa, Repetición*}	
Símbolo inicial:		{*Regex*}	
Producciones:		{	
Símbolo	→	{*nombre = requestDialog "Nombre del Símbolo:"; ...*}	**(1)**
Regex	→	*Símbolo* {*regex = token $1.nombre; ...*}	**(2)**
Regex	→	*Alternativa* {*regex = $1.regex; ...*}	**(3)**
Regex	→	*Repetición* {*regex = $1.regex; ...*}	**(4)**
Regex	→	*Concatenación* {*regex = $1.regex; ...*}	**(5)**
Concatenación	→	*Regex Regex* {*regex = ($1.regex) < * > ($2.regex) < @ (\(x,y) → x ++ y); ...*}	**(6)**
Alternativa	→	*Regex Regex* {*regex = ($1.regex) < | > ($2.regex); ...*}	**(7)**
Repetición	→	*Regex* {*regex = many ($1.regex); ...*}	**(8)**
		}	

Cuadro 2.6: GD Regex modificada para la construcción de un parser.

pertenece o no al lenguaje definido por la expresión regular resultante. Luego mediante la función:

$$\begin{array}{lcl} valid & & Parser\ a \rightarrow String \rightarrow Bool \\ valid\ parser\ string & = & not\,\$\,null\ (just\ parser\,\$\,string) \end{array}$$

que recibe como parámetro el parser generado p y un string s, podemos testear la validez de s con respecto a la expresión regular validada por p. Si bien en este ejemplo se cambió el atributo *regex* para que contenga el parser generado, se podría haber agregado un nuevo atributo para este propósito, conservando el anterior. Más aún, podríamos agregar o modificar un atributo para que el mismo guarde una regex *PERL*, *awk* o cualquier semántica deseada. De la misma manera en que se modificaron los atributos semánticos, podemos cambiar la apariencia en forma sencilla, modificando los atributos que describen la apariencia gráfica del diagrama (no mostrada en el ejemplo). Más aún, se puede modificar fácilmente la estructura de los diagramas descritos por la GD agregando o quitando reglas (producciones). Por ejemplo se podría agregar una producción:

$$Repeticion1 \quad \rightarrow \quad Regex$$

```
type Parser symbol result  =  symbol → [([symbol], result)]

just     :: Parser s ()
just p   =  filter (null . fst) . p
epsilon  :: Parser s ()
epsilon p = [(xs, ())]
token    :: Eq [s] ⇒ [s] → Parser s [s]
token t xs = let
               n = length t
             in
               if (t == take n xs) then
                 [((drop n xs), t)]
               else
                 []
(<*>)          :: Parser s a → Parser s b → Parser s (a, b)
(p1 <*> p2) xs  =  [(xs2, (v1, v2)) |
                      (xs1, v2) ← p1 xs, (xs2, v2) ← p2 xs1]
(<|>)          :: Parser s a → Parser s a → Parser s a
(p1 <|> p2) xs  =  p1 xs ++ p2 xs
many     :: Parser s r → Parser s [r]
many p   =  p <*> many p <@ (\(x, xs) → x : xs)
            <|> epsilon <@ (\p → [])
(<@)           :: Parser s a → (a → b) → Parser s b
(p <@ f) xs     =  [(ys, f v) | (ys, v) ← p xs]
```

Figura 2.6: Combinadores de parsers.

similar a *Repetición*, donde la misma representa una o más repeticiones de una expresion regular en lugar de cero o más. Si bien esto lo podríamos haber logrado utilizando las construcciones existentes (*Concatenación* y *Repetición*), y por lo tanto no brinda mayor expresividad, sí facilita la edición gráfica y por otro lado reduce el tamaño del DAG asociado al diagrama. Este tipo de detalles deben ser tenidos en cuenta por el diseñador de la gramática.

2.3.2. Diagramas de E-R

Los diagramas E-R son de amplio uso y utilidad en el área del diseño de bases de datos relacionales [Che76b]. En el siguiente ejemplo se muestra una gramática de diagramas para la construcción de diagramas E-R que explora detalles de las GDs que no fueron tratados en el ejemplo anterior: atributos de validación y la motivación de la necesidad de compartir nodos en los árboles de derivación (justificando de esta manera el uso de DAGs en el modelo semántico de las GDs). Ciertas construcciones, como entidades débiles, generalizaciones y la cardinalidad de las relaciones, fueron excluídas con el fin de mantener el ejemplo simple y enfocar la discusión en los detalles salientes de las GDs y no en el lenguaje visual modelado.

Un diagrama de E-R está constituido básicamente por entidades, relaciones, atributos y agregaciones. La gramática de diagramas presentada en el Cuadro 2.8 permite la construcción de este subconjunto de los diagramas de E-R. En las Fig. 2.8 y 2.9 se pueden observar las funciones auxiliares utilizadas para dar semántica a los diagramas generados, implementadas, al igual que el ejemplo anterior, en Haskell. La GD consta de un solo terminal *Entidad* que posee 3 atributos de dominio: *nombre*, *atributos* y *er* cuyos valores son asignados mediante las ecuaciones de la producción (1). Como decisión de diseño se optó que los atributos de las entidades (no confundir con los atributos de la GD) y relaciones sean representados, respectivamente, mediante un atributo de los símbolos *Entidad* y *Relación*, en lugar de terminales de la gramática; esto simplifica notablemente la gramática y el DAG

asociado en la construcción del diagrama de E-R (por supuesto, podrían haberse incluído como terminales de la gramática si hubiese resultado más conveniente). Luego se encuentran las producciones (2)-(6) que permiten la construcción de relaciones a partir de entidades, agregaciones y otras relaciones. Las producciones (3) y (5) deben ser observadas con especial cuidado ya que las mismas introducen atributos de *validación*, que hasta este punto no habían sido tratados. Estos atributos se interpretan exigiendo la aplicación de producciones sólo a relaciones binarias, esto es una relación entre dos componentes. Por lo tanto esta gramática limita los diagramas de E-R construibles sólo a relaciones como máximo ternarias. Dicha restricción es arbitraria y se podrían haber limitado las relaciones a binarias o no limitarlas en absoluto. Como muchos lectores ya habrán intuido, las restricciones podrían haberse contemplado cambiando la gramática, pero la utilización de atributos de validación permite escribir una gramática más consisa y con un mayor grado de expresividad. En la producción (7) se encuentra la regla que permite crear agregaciones a partir de una relación. Por último las producciones (8) y (9) permiten alcanzar el símbolo inicial de la gramática, *E-R*, lo que establece que tanto una entidad como una relación son diagramas válidos, pero no así una agregación, si ésta no es luego utilizada para crear una nueva relación. En la figura 2.7 se puede ver una posible representación visual para el lenguaje visual E-R que hemos definido y el DAG generado.

En este ejemplo se puede observar otro detalle sobresaliente de las GDs: la posibilidad de compartir terminales y noterminales de la gramática. En este caso el noterminal *Agregación* es compartido por 2 relaciones; esto se hace más evidente en el DAG asociado que se encuentra a la derecha del diagrama. Esta característica es fundamental en el desarrollo de lenguajes visuales, ya que permite compartir estructuras (no es necesario en un lenguaje textual ya que siempre se introduce una nueva ocurrencia del símbolo con idéntico texto, al cual se le asigna un nuevo nodo dentro del árbol de derivación manteniendo de esta forma la estructura de árbol).

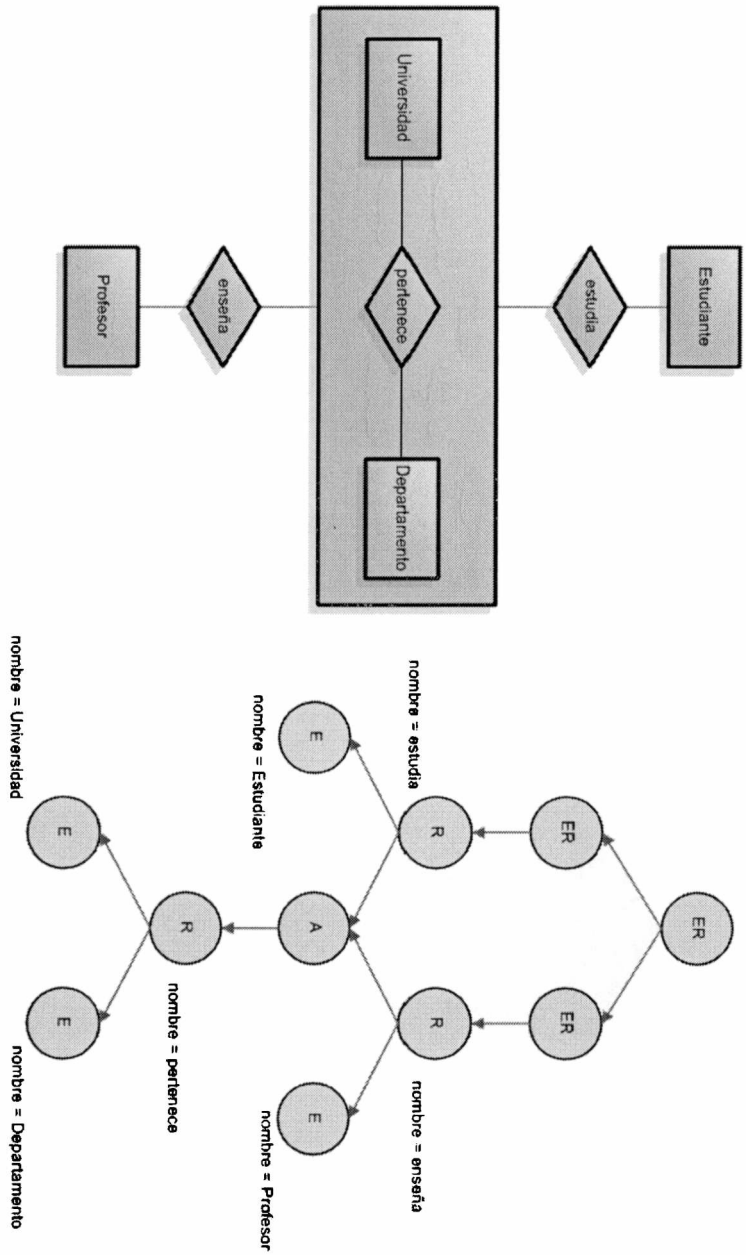

Figura 2.7: Diagrama de E-R con su correspondiente DAG.

Terminales:		{*Entidad*}	
Noterminales:		{*E-R, Relación, Agregación*}	
Símbolo inicial:		{*E-R*}	
Producciones:		{	
Entidad	→	{*nombre = requestDialog "Nombre?";*	**(1)**
		atributos = inputAttributes "Atributos?";	
		er = newEntity $$.nombre $$.atributos; ...}	
Relación	→	*Entidad Entidad {nombre = requestDialog "Nombre?"*	**(2)**
		atributos = inputAttributes "Atributos?";	
		er = newRelation $$.atributos $1.er $2.er; ...}	
Relación	→	*Relación Entidad {er = newRelation // $1.er $2.er; ...}*	**(3)**
		: {_ = isBinRelation $1.er}	
Relación	→	*Agregación Entidad {nombre = requestDialog "Nombre?";*	**(4)**
		atributos = inputAttributes "Atributos?";	
		er = newRelation $$.nombre $$.atributos $1.er $2.er; ...}	
		: {_ = isBinRelation $1.er}	
Relación	→	*Relación Agregación {er = newRelation // $1.er $2.er; ...}*	**(5)**
		: {_ = isBinRelation $1.er}	
Relación	→	*Agregación Agregación {nombre = requestDialog "Nombre?";*	
		atributos = inputAttributes "Atributos?";	**(6)**
		er = newRelation $$.nombre $$.atributos $1.er $2.er; ...}	
		: {_ = isBinRelation $1.er}	
Agregación	→	*Relación {er = newAgregation $1.er; ...}*	
E-R	→	*Entidad {er = $1.er; ...}*	**(8)**
E-R	→	*Relación {er = $1.er; ...}*	**(9)**
		}	

Cuadro 2.8: GD para la construcción de diagramas Entidad-Relación.

El proceso de construcción es similar al detallado en el ejemplo anterior, por lo cual fue omitido.

```
data ER   =  Relation String [ER] [Attr] | Agregation ER
          |  Entity String [Attr]
data Attr =  Attr String | Key String

newEntity                 ::  String → Attr → ER
newEntity name attrs      =   Entity name attrs

newRelation :: String → [Attr] → ER → ER → ER
newRelation _ _ r@(Relation _ _ _) e@(Entity _ _)                =
    Relation (getName r) (e : getParticipants r) (getAttrs r)
newRelation _ _ r@(Relation _ _ _) a@(Agregation _)               =
    Relation (getName r) (a : getParticipants r) (getAttrs r)
newRelation name attrs e'@(Entity _ _) e''@(Entity _ _)          =
    Relation name [e', e''] attrs
newRelation name attrs a'@(Agregation _) a''@(Agregation _)       =
    Relation name [a', a''] attrs
newRelation name attrs a@(Agregation _) e@(Entity _ _)           =
    Relation name [a, e] attrs

newAgregation                 ::  ER → ER
newAgregation r@(Relation _ _ _)  =  Agregation r
```

Figura 2.8: Modelo semántico de los diagramas E-R (Parte 1).

```
isBinRelation                    :: ER → ER
isBinRelation (Relation _ es _)  =  length es == 2

getKey (Entity _ as)       =  filter p as == 2
                                where
                                  p (Key _) = True
                                  p _ = False
getKey (Agregation r)      =  getKey r
getKey (Relation _ ers _)  =  concat (map getKey ers)

getAttrs (Relation _ _ as)  =  as
getAttrs (Agregation r)     =  getAttrs r
getAttrs (Entity _ as)      =  as

getParticipants (Relation _ ers _)  =  ers

getName (Relation n _ _)  =  n
getName (Entity n _)      =  n
```

Figura 2.9: Modelo semántico de los diagramas E-R (Parte 2).

Capítulo 3

El Editor Parametrizable

En este capítulo se discutirá la implementación de una herramienta que permite la edición gráfica de lenguajes visuales. La herramienta está basada en los fundamentos teóricos presentados en el Capítulo 2. Este desarrollo demuestra la importancia práctica de las gramáticas de diagramas y su aplicación. El propósito de este trabajo no es la implementación de una herramienta profesional, por lo cual el prototipo que se brindará debe tomarse como una implementación de referencia o la base para futuras implementaciones. El editor posee muchas limitaciones que no necesariamente implican una falencia en el formalismo. Existen excelentes herramientas para la edición de diagramas y lenguajes visuales en general (Visio[1], Kivio[2]), las cuales pueden ser potenciadas con la utilización de una herramienta formal subyacente como la presentada en este trabajo.

El editor gráfico que se presentará permite la edición de diagramas por composición, según el proceso descripto para las GD: a partir de diagramas básicos permite construir diagramas complejos aplicando las producciones descriptas por la gramática de diagramas con la cual se parametriza. En el contexto del editor gráfico denominamos a las producciones, *reglas de compo-*

[1] `http://www.microsoft.co m/office/visio/`
[2] `http://www.thecompany.com`

sición, ya que denotan de forma más precisa su propósito. Durante el proceso de edición la computación de atributos (tanto visuales como semánticos o de dominio) se produce en forma implícita obteniendo una respuesta inmediata del resultado visual. De esta forma el desarrollador de lenguajes visuales puede experimentar con nuevos lenguajes sólo mediante la manipulación de la gramática (producciones, atributos gráficos y semánticos) mientras que el editor parametrizable es el responsable de producir, como resultado de la parametrización, la herramienta visual para los mismos. Este proceso es similar al que comúnmente se observa en herramientas para la generación automática de lexers y parsers como *lex* y *yacc* [LMB92]. Estas herramientas ampliamente utilizadas y difundidas permiten al diseñador o implementador de lenguajes textuales experimentar con la sintaxis y semántica de los mismos mediante la manipulación de un meta-lenguaje que describe su gramática y reglas semánticas. A lo largo de este capítulo se mostrarán en detalle los distintos componentes del editor así como las diferentes herramientas utilizadas en el desarrollo de cada componente del mismo.

3.1. El Editor

El editor de diagramas está dividido en 2 partes. En la parte superior se encuentra la barra de herramientas, compuesta de las figuras básicas y la herramienta de manipulación. La mayor parte del editor la abarca el *canvas*, que es el lugar donde se construyen los diagramas agregando figuras que se encuentran en la barra de herramientas o componiendo las figuras que se encuentran en él, aplicando las reglas de composición de la GD por medio de un menú desplegable. En la Figura 3.1 puede observarse el editor con sus distintos componentes.

El manejo gráfico (la edición y manipulación de diagramas) está basada en el framework Hotdraw. Hotdraw es una capa de abstracción sobre la

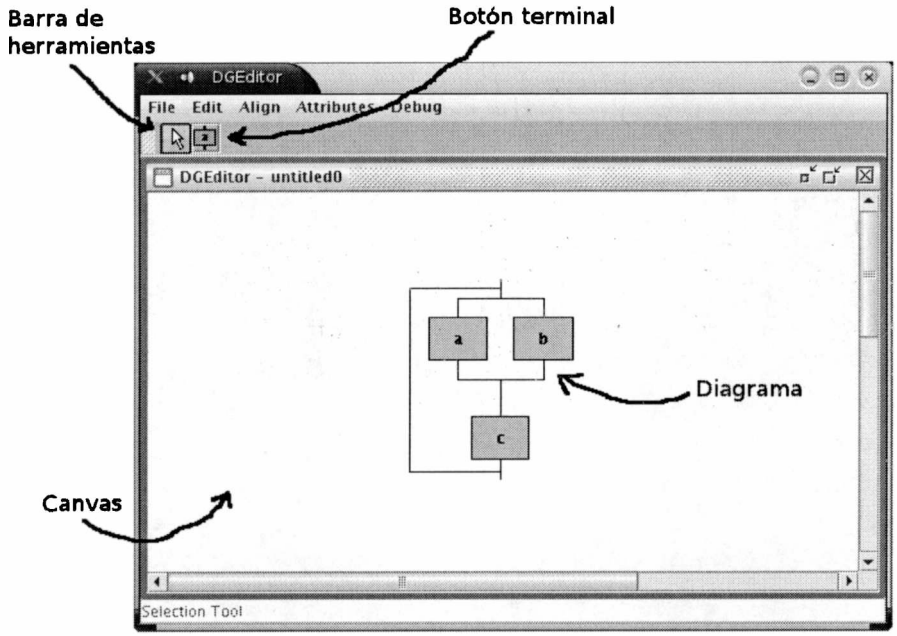

Figura 3.1: Editor de diagramas para Regex $((a|b) * c)$.

API gráfica de Java: *Java 2D*[1]. Este framework provee el marco necesario para el desarrollo de editores gráficos de diagramas mediante la utilización de construcciones básicas provistas y la extensión del mismo. El trabajo presentado, a su vez, provee una capa de abstracción conveniente por encima, en la cual recae la responsabilidad de manejar las GDs con las cuales el editor es parametrizado y la sintaxis abstracta del diagrama (DAG).

La construcción de esta herramienta se realizó completamente en el lenguaje de programación *Java* [Eck00] y estuvo suplementado con la utilización de distintos frameworks y bibliotecas. Entre las bibliotecas utilizadas se destacan *Castor*[2] que permite la reconstrucción de objetos a partir de archivos XML [MUT99], *Dynamic Java*[3] para la ejecución dinámica (interpretación) de código Java y *GNU Regex*[4] para el manejo de las expresiones regulares en la sustitución de variables de los atributos. Para el manejo de la edición gráfica se utilizó el framework para la construcción de editores gráficos *Hotdraw*, mencionado anteriormente, [Joh92a, Joh92b, Bra95] en su versión Java.

3.2. Descripción General

La herramienta descrita en en este capítulo pretende automatizar la construcción de editores gráficos para lenguajes visuales. La misma es parametrizada por medio de archivos de configuración XML que describen una GD. Dada una gramática de diagramas GD_L (gramática de diagramas para un lenguaje visual L) el resultado de la parametrización es el editor gráfico E_L para dicho lenguaje. La Figura 3.2 ilustra el proceso de parametrización del editor genérico. Una de las características notables de este editor es que permite la manipulación transparente de la sintaxis abstracta del diagrama mientras que el resultado de dicha manipulación es la visualización gráfica

[1] `http://java.sun.com/products/java-media /2D/index.jsp`
[2] `http://www.exolab.org/castor`
[3] `http://koala.ilog.fr/djava/`
[4] `http://www.cacas.org/java/gnu/regexp/`

del diagrama que el usuario está editando. Cuando decimos "la manipulación transparente de la sintaxis abstracta", nos referimos a que cada interacción con el editor dispara una modificación de la sintaxis abstracta del diagrama que se está editando, y esto a su vez modifica la apariencia del diagrama sin que el usuario final esté necesariamente consciente de todo este proceso. Más adelante en este capítulo se mostrará en detalle este proceso de ida y vuelta entre la sintaxis abstracta y el diagrama visual subyacente.

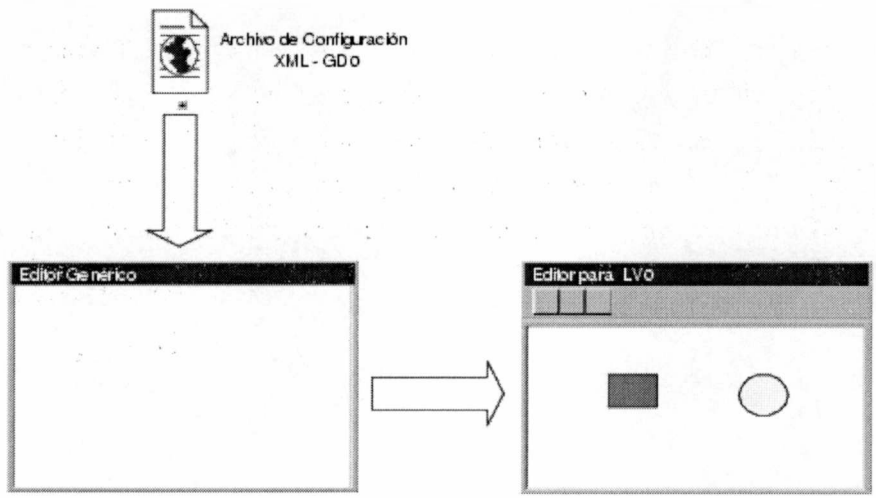

Figura 3.2: Arquitectura general del editor parametrizable.

La herramienta presentada está dividida en cuatro partes principales. La primer parte es la que permite la configuración del sistema a partir de una GD. La segunda provee la evaluación de atributos y manejo de sintaxis abstracta de una GD. Las últimas dos partes muestran el proceso de selección de reglas del editor y el subsistema de manipulación gráfica que permite la edición de diagramas y combina los anteriores.

3.3. Configuración del Editor

El editor permite ser parametrizado mediante archivos de configuración *XML*, que describen un lenguaje visual mediante gramáticas de diagramas. La estructura básica de un archivo de configuración se puede observar en la Fig. 3.3. En los Apéndices A y B de este trabajo se pueden observar el *DTD* que describe la estructura de estos archivos de configuración y ejemplos completos de los mismos para distintos lenguajes visuales.

```
<diagram>
    <toolbar>
        ...
   </toolbar>
    <start>...</start>
    <compositionrules>
        ...
    </compositionrules>
</diagram>
```

Figura 3.3: Estructura general del archivo de configuración.

El archivo de configuración está dividido en tres partes bien definidas. La primera es la sección que definirá los componentes de la barra de herramientas del editor compuesta por los terminales de la gramática. La segunda muestra los símbolos iniciales de la gramática, utilizados para validar los diagramas construidos. Por último se pueden observar las reglas de composición que dictan cómo pueden construirse los diagramas en el editor.

La sección de los símbolos iniciales es indicativa por sí misma. Está compuesta por los símbolos que denotan diagramas autocontenidos, esto es, diagramas cuyo correspondiente nodo en el DAG de la sintaxis abstracta del diagrama puede ser raíz, como se explicó en el Capítulo 2. Tanto la primer sección del archivo de configuración como la última requieren un tratamiento extensivo y serán desarrollados en detalle en las siguientes secciones.

3.3.1. La Barra de Herramientas

La barra de herramientas (`toolbar`) está compuesta por una secuencia de herramientas (`tool`), que determinan qué figuras básicas podrán agregarse al canvas (ver Sec. 3.1). Cada herramienta consta de una serie de atributos divididos conceptualmente en dos grupos siguiendo las definiciones del Cap. 2. El primer grupo de atributos determina los aspectos relacionados con los detalles gráficos del editor y su configuración, y el segundo, los detalles semánticos de la figura que la herramienta manipula. Desde el punto de vista de las gramáticas de diagramas la barra de herramienta representa el conjunto de terminales de la misma. Cada terminal de la GD es mapeado a una herramienta del editor de diagramas. Sin embargo existen herramientas que no corresponden a ningún terminal, como por ejemplo la herramienta de selección y manipulación de figuras. En la Fig. 3.4 se puede observar un fragmento del archivo de configuración donde se describe la barra de herramienta del editor de caras.

Cada herramienta consta de un *nombre* (`name`), que es el nombre del terminal de la GD representada por el archivo de configuración. Dicho nombre generalmente es descriptivo de la figura, y al igual que en cualquier gramática, es utilizado en sus producciones; veremos más adelante en este capítulo cómo el mismo es utilizado dentro de las reglas de composición. Además, cada herramienta consta de un ícono (`icon`) que es mostrado en la barra de herramienta y una ayuda (`tooltip`) que es mostrada al posicionar el cursor del mouse sobre ésta. Luego, como se observa en el archivo de configuración, se encuentra el atributo de mayor importancia para la visualización del terminal: el atributo figura (`figure`). El resultado de la evaluación de este atributo es lo que se agregará al canvas cuando la herramienta esté seleccionada y se presione el botón izquierdo del mouse sobre éste. Por último existe el atributo de propiedades (`properties`), que se utiliza para manejar la configuración de la figura. Presionando el botón derecho del mouse sobre cada figura se despliega un menú con la opción "propiedades" que evalúa este atributo y

```
<toolbar>
  <tool>
    <name>Mouth</name>
    <icon>/CH/ifa/draw/images/ELLIPSE</icon>
    <tooltip>New Mouth...</tooltip>
    <figure>
      import CH.ifa.draw.figures.EllipseFigure;
      import ar.edu.dgeditor.figures.FixedSizeFigure;

      new FixedSizeFigure(new EllipseFigure(), 30, 10);
    </figure>
    <properties/>
    <domain_attributes/>
  </tool>
  <tool>
    <name>Eye</name>
    <icon>/CH/ifa/draw/images/ELLIPSE</icon>
    <tooltip>New Eye...</tooltip>
    <figure>
      import CH.ifa.draw.figures.EllipseFigure;
      import ar.edu.dgeditor.figures.FixedSizeFigure;

      new FixedSizeFigure(new EllipseFigure(), 10, 10);
    </figure>
    <properties/>
    <domain_attributes/>
  </tool>
</toolbar>
```

Figura 3.4: Configuración de la barra de herramientas.

como resultado generalmente abre una ventana que permite la edición de los atributos modificables del diagrama. Los atributos hasta aquí descriptos son obligatorios para la construcción del editor, aunque algunos pueden estar vacíos.

Luego se encuentran los atributos semánticos del diagrama. Los mismos son opcionales, dependiendo de si el lenguaje visual definido tiene semántica asociada o no. En el ejemplo mostrado, los terminales de la GD no tienen atributos semánticos. Se pueden definir tantos atributos semánticos como el lenguaje visual descripto por la DG demande. En el ejemplo de la Sección 2.3.1 por ejemplo, la gramática definía un atributo semántico `regex` que contenía el resultado de la construcción de un parser para la expresión regular editada utilizando combinadores de parser para Haskell.

3.3.2. Las Reglas de Composición

En esta sección se verá la parte más interesante de la configuración del editor: las reglas de composición. La Figura 3.5 muestra un extracto del archivo de configuración conteniendo las reglas de la gramática brindada en Cuadro 3.1. Estas reglas dictan cómo pueden componerse diagramas en el editor. Como se puede comprobar, las reglas de composición juegan un papel central en la construcción de diagramas.

Terminales:	$\{Eye, Mouth\}$
Noterminales:	$\{Head\}$
Símbolo inicial:	$\{Head\}$
Producciones:	$\{\ Head \rightarrow Eye\ Eye\ Head\ \}$

Cuadro 3.1: GD para la construcción de caras.

Las reglas de composición (`compositionrules`) están compuestas por una secuencia de reglas (`rule`). Cada regla, al igual que las herramientas, posee una serie de atributos obligatorios con el mismo propósito: `nombre` y `figura`. Además una regla contiene una lista de componentes (`components`) donde cada componente (`component`) denota o bien el nombre de una herra-

```
<compositionrules>
  <rule>
    <name>Head</name>
    <components>
      <component>Eye</component>
      <component>Eye</component>
      <component>Mouth</component>
    </components>
    <icon>/ar/edu/dgeditor/images/face.gif</icon>
    <figure>
      import CH.ifa.draw.framework.Figure;
      import ar.edu.dgeditor.figures.faces.FaceFigure;
      import java.awt.Color;

      FaceFigure face = new FaceFigure((Figure)$1.figure,
                                       (Figure)$2.figure,
                                       (Figure)$3.figure);
      face.setFillColor(Color.ORANGE);
      face;
    </figure>
    <domain_attributes>
      <attribute>
        <name>face_name</name>
        <value>
          import javax.swing.*;
          String name =
              JOptionPane.showInputDialog(null,"Name?");
          name == null ? "" : name;
        </value>
      </attribute>
    </domain_attributes>
  </rule>
</compositionrules>
```

Figura 3.5: Configuración de las reglas de composición.

mienta (terminal) o el nombre de alguna regla de composición (noterminal). Los nombres de las reglas no son más que los nombres de los noterminales de la GD descripta por el archivo de configuración, y los componentes, los terminales y noterminales del lado derecho de una producción; las reglas de composición son las producciones de las GD. También, al igual que las herramientas, poseen un conjunto de atributos semánticos opcionales que dependen del lenguaje visual modelado por la gramática. En el ejemplo de la Fig. 3.5 podemos ver la regla de composición que a partir de dos *ojos* y una *boca* construye una *cara*, y cuyo atributo semántico es el nombre que se le da a la cara.

3.3.3. Construcción del Editor

Hasta el momento se han evitado los detalles de implementación de la construcción del editor a partir de una DG, para enfocarse en su configuración (parametrización). En esta sección se mostrarán los detalles intrínsecos de la configuración. La Figura 3.6 muestra el diagrama estático de clases que representan la configuración del editor. Este modelo es reconstruido a partir del archivo de configuración XML que se mostró en las Subsecciones anteriores utilizando la biblioteca Castor[1] para mapeo de Objetos $\leftrightarrow$ XML.

La construcción del modelo se realiza en forma automática con la herramienta `org.exolab.castor.builder.SourceGenerator` provista con Castor. Esta herramienta genera el código correspondiente al modelo a partir del archivo de definición de esquema del XML (XSD), con lo cual sólo modificando la especificación del archivo de configuración podemos obtener nuevamente el código Java correspondiente. Este archivo de definición de esquema del XML es generado a su vez con otra herramienta provista por Castor (`org.exolab.castor.builder.Converter`) a partir del archivo de DTD mostrado en el Apéndice A. En este trabajo se incluyó el archivo de DTD y no su correspondiente XSD debido a la simplicidad del primero y

[1] `http://www.exolab.org/castor`

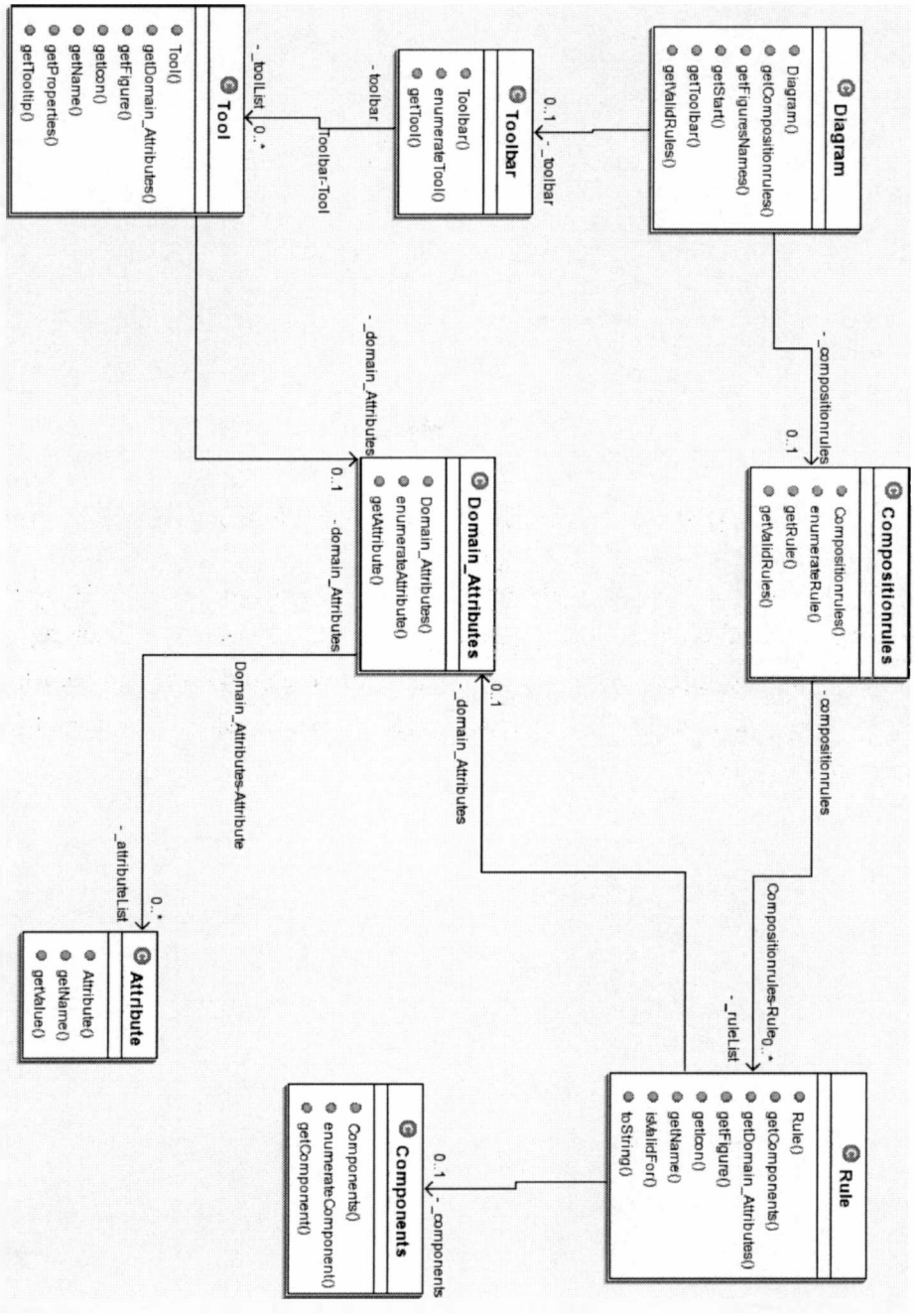

Figura 3.6: Diagrama estático de configuración del editor.

su clara correspondencia con el modelo de configuración del editor mostrado en la Figura 3.6. Se puede observar claramente en el DTD del archivo de configuración mostrado en el Apéndice A como éste especifica textualmente el modelo de configuración del editor. En el diagrama estático de clases se puede obervar que un "diagrama" (`Diagram`) está compuesto por una "barra de herramientas" (`Toolbar`) que a su vez contiene una colección de "herramientas" (`Tools`). El editor presentado configura su barra de herramientas a partir de estos objetos utilizando algunos de los atributos descriptos previamente (`icon`, `tooltip`, ...). Además un "diagrama" está compuesto por las "reglas de composición" (`Compositionrules`) que contiene una colección de "reglas" (`Rules`). Tanto una regla como una herramienta tienen "atributos de dominio" (`Domain_Attributes`) compuestos por una colección de atributos (`Attribute`) utilizados para asignarle semántica a los diagramas. La reconstrucción de los objetos del modelo de configuración a partir de un archivo de configuración XML también es realizada automáticamente por el framework de mapeo Objetos↔XML. En las siguientes secciones se explorará como esta configuración es utilizada por el editor durante la edición para la visualización del diagrama editado, selección de reglas y construcción de la sintaxis abstracta.

3.4. Sintaxis Abstracta

La estructura utilizada para el modelado de la sintaxis abstracta de un diagrama puede observarse en la Figura 3.7; este modelo permite la construcción de DAGs. Dado que el propósito de este trabajo no está centrado en la computación eficiente de gramáticas de atributos, el modelo para esta implementación fue elegido en base a su simplicidad. Como se puede observar, la sintaxis abstracta (AS, `AbstractSyntax`) está compuesta por una colección de DAGs. La AS permite la manipulación transparente (façade [GHJV95]) de un DAG, ignorando su implementación concreta, mediante la utilización

de los métodos `addLeaf(...)` y `addNode(...)`. También permite el recorrido en amplitud mediante el método `bfs(...)` y obtener la raíz de un DAG mediante el método `getRoot(...)`. El resto de la funcionalidad es auxiliar y se puede revisar con mayor detalle en la implementación del editor provista con este trabajo. Un DAG es un *composite* [GHJV95] que puede ser o bien una hoja (`Leaf`) con un valor o bien un nodo (`Node`) que contiene una colección de DAGs (`DAG`) hijos. En forma más concisa la estructura de un DAG puede expresarse como una unión discriminada de la siguiente forma:

$$DAG\ a = Node\ [DAG\ a] \mid Leaf\ a$$

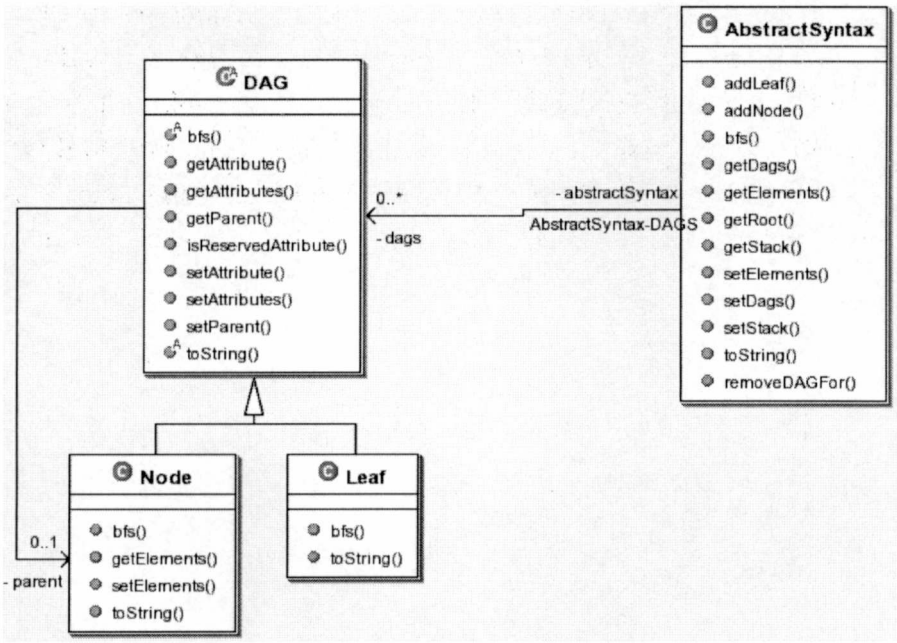

Figura 3.7: Modelo de la sintaxis abstracta de un diagrama.

Cada elemento de un DAG contiene una Hashtable con las asociaciones *(clave, valor)*, donde la clave es el nombre del atributo, y el valor, el resultado de la evaluación de la expresión del atributo, salvo para los atributos *nombre*,

ícono y *ayuda* que son utilizados como String y *propiedades* cuyo valor es el mismo código de la configuración del editor y es evaluado cada vez que este es requerido. Cada nodo de la sintaxis abstracta corresponde a un noterminal de la gramática, mientras que una hoja corresponde a un terminal.

3.5. Evaluación de Atributos

Los atributos de las DG en esta implementación están expresados en Java aumentado con la posibilidad de referenciar atributos. La evaluación de estos atributos se realizó con la ayuda de la biblioteca para la ejecución dinámica de código Java, Dynamic Java[1]. Además, para el reemplazo de referencias a atributos por código Java puro se utilizó la biblioteca para el manejo de expresiones regulares GNU Regex.

La figura 3.8 muetra un ejemplo de una producción con atributos que contiene referencia a otros atributos. La referencia de atributos se expresa mediante el caracter '$' seguido por la posición del terminal o noterminal dentro del lado derecho de la producción para denotar el símbolo y finalmente con el calificador '.' seguido por el nombre del atributo al cual se hace referencia. De esta forma, por ejemplo, el atributo *figure* del símbolo *Concatenation* contiene las referencias `$1.figure` y `$2.figure` a los atributos *figure* de los dos noterminales Regex del lado derecho de la producción, respectivamente. Además existe una notación especial $$ para los atributos del símbolo del lado izquierdo de la producción. Finalmente, si una referencia a un atributo se encuentra del lado izquierdo de una asignación (`$$.nombre = ...`) esto representa una asignación del valor obtenido al evaluar la expresión del lado derecho de la asignación al atributo. En el Apéndice C se presenta la especificación BNF completa de la referencia y asignación de atributos.

La ventaja de esta notación para la referencia de atributos es su simplicidad y comodidad. Sin embargo, como se ha mencionado, el código de

[1] `http://koala.ilog.fr/djava/`

```
<rule>
  <name>Concatenation</name>
  <components>
    <component>Regex</component>
    <component>Regex</component>
  </components>
  <icon>purple-m.gif</icon>
  <figure>
    import ar.edu.dgeditor.figures.regex.*;
    new ConcatenationFigure(
                (RegexFigure)$1.figure,
                (RegexFigure)$2.figure);
  </figure>
  <domain_attributes>
    <attribute>
      <name>regex</name>
      <value>
        "(" + $1.regex + "." + $2.regex + ")";
      </value>
    </attribute>
  </domain_attributes>
</rule>
```

Figura 3.8: Ejemplo de referencias a atributos.

los atributos debe ser Java puro para ser aceptado por el intérprete, y las referencias a atributos como fueron descriptos, no conforman esta propiedad. Debido a esto, fué necesario introducir un mecanismo de reescritura para las referencias y asignaciones de atributos. Esto fué logrado utilizando expresiones regulares y sustituciones de la siguiente manera.

1. **Referencias a atributos:**
   Las referencias a atributos de la forma `$x.atributo` son reemplazadas por el fragmento de código Java:

   (DAG)params.get(x).getAttribute(atributo)

   donde *params* es la colección (*ArrayList*) ordenada (según el orden de selección en el editor) de los DAGs correspondientes a la figuras seleccionadas en el editor. Luego x es la posición del elemento del cual se obtiene el atributo. Cabe aclarar que el DAG asociado al símbolo del lado izquierdo de la producción que se está construyendo es pasado como parámetro al evaluador de atributos en la posición 0 de la colección por lo cual referencias del estilo `$$.atributo` se reemplazan por el fragmento:

   (DAG)params.get(0).getAttribute(atributo)

2. **Asignación de atributos:**
   Las asignaciones de atributos en una GD son de la forma `$x.atributo = expresión`, y las mismas son reemplazadas por el fragmento de código Java:

   (DAG)params.get(x).setAttribute(atributo, expresión)

La expresión regular utilizada para capturar las ocurrencias de referencias y asignaciones a atributos es la siguiente:

$$(\backslash\$\backslash\$|\backslash\$[0-9]) + \backslash.[a-zA-Z]([a-zA-Z0-9\_\ backslash-]*)((\backslash W)? = (\backslash W)?[a-zA-Z0-9\_\backslash-](a-zA-Z0-9\_\backslash-])*;)?$$

En la expresion regular mostrada los caracteres con significado especial en una REGEX están precedidos por "\" tales como "$","−" , "." y el mismo "\". Por otro lado el caracter "−" significa rango, el "$\backslash W$" cualquier secuencia de blancos (incluidos *TABS*). El caracter "?" significa que la secuencia es opcional, el "|" alternativa, la ausencia de separadores, concatenación y finalmente "*" y "+" la repetición de 0 ó más expresiones y 1 ó más respectivamente. Una referencia completa sobre el uso de expresiones regulares y sustituciones se puede encontrar en cualquier manual de *awk*[1] o *Perl*[2].

Una vez reemplazadas las ocurrencias de referencias a atributos, el resultado es código Java puro que puede ser evaluado por la biblioteca elegida en esta oportunidad: Dynamic Java (DJ). Un fragmento de código evaluado por DJ puede recibir parámetros externos (en este caso, la colección de DAGs asociados a los símbolos del lado derecho de la producción más el DAG que se está creando para el símbolo del lado izquierdo de la producción) en la variable *params*. El resultado de la evaluación de la última sentencia Java será el resultado del fragmento de código que será asignado al atributo del DAG.

3.6. Selección de Reglas

Los diagramas que pueden ser construídos dependen de la selección actual en el editor y el conjunto de reglas de composición pesentadas por la GD.

[1] `http://www.gnu.org/manual/gawk-3.1.1/html_node/`
[2] `http://www.perldoc.com/perl5.8.0/pod/perl.html`

Denominamos "proceso de selección de reglas" al mecanismo que a partir de un conjunto de diagramas seleccionados calcula las reglas disponibles

Sea S el conjunto de diagramas seleccionados, N_s el conjunto nombres de los terminales y noterminales representados por las raíces de los DAGs asociados a los diagramas, P el conjunto de producciones de la GD utilizada y $rhs(p)$ la función que retorna el conjunto de strings con los nombres de los terminales y noterminales del lado derecho de una producción, el conjunto de reglas de producciones válidas se calculan con el siguiente algoritmo:

$validRules = \emptyset;$

foreach$(p \in P)\{$
  if $(N_s \subseteq rhs(p) \wedge rhs(p) \subseteq N_s)$
    $insert(validRules, p);$
$\}$

Como puede observarse el conjunto de reglas válidas se calcula obteniendo todas las producciones cuyos nombres de terminales y no terminales del lado derecho, son iguales a todos los nombres de terminales y noterminales asociados a los nodos raíz de la selección, sin importar el orden. Esto puede generar ambigüedades dado que más de una producción puede ser aplicada; sin embargo como se explicó en el Capítulo 2, el usuario selecciona qué regla se aplicará, eliminando la ambigüedad. Es importante reconocer en este punto la diferencia con un lenguaje textual ya que en este último la forma de desambigüar debe ser establecida a priori ya sea mediante la modificación de la gramática o reglas preestablecidas en el proceso de parsing. Más aún a diferencia de un lenguaje textual podríamos seleccionar una misma figura más de una vez mientras que en un lenguaje textual introduciríamos una nueva instancia del string. En la Figura 3.9 se pueden observar el diagrama de secuencia que muestra los pasos involucrados en el editor para el proceso de selección de reglas.

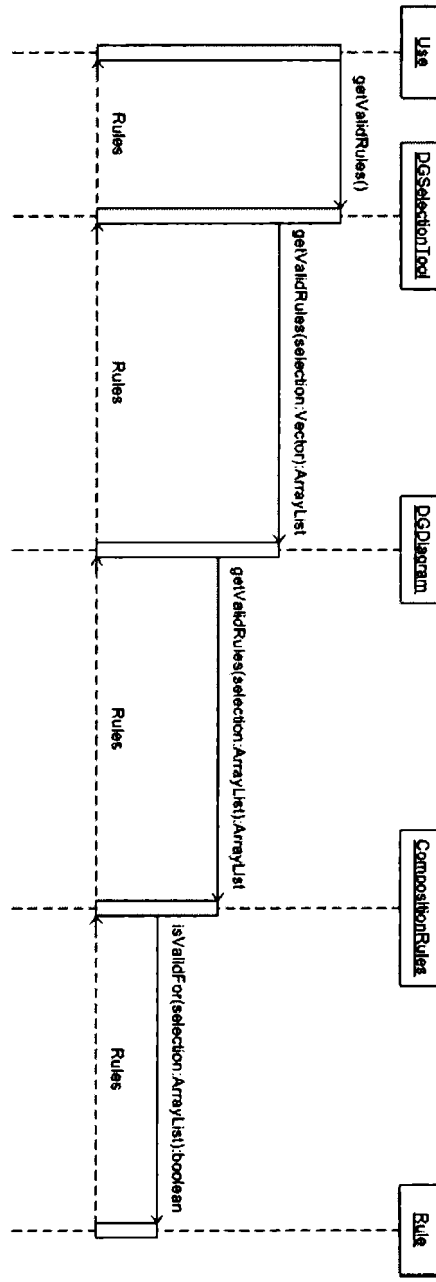

Figura 3.9: Proceso de selección de reglas.

3.7. Manipulación de Diagramas y Sintaxis Abstracta

En las figuras 3.10 y 3.11 se puede observar, a alto nivel, el flujo del proceso de construcción de una hoja y un nodo del DAG asociado a un diagrama, respectivamente. Estos procesos son ejecutados al agregarse una figura al canvas o componerse figuras existentes, respectivamente.

La construcción de una hoja del DAG es iniciada por el usuario al seleccionar una herramienta correspondiente a un diagrama básico del editor y presionar el botón izquierdo del mouse sobre el canvas. Esta acción dispara la creación de la figura correspondiente por la herramienta seleccionada – en este caso una extensión de `CreationTool` de hotdraw, `DGCreationTool`. La clase base `CreationTool` permite la creación de nuevas figuras en base a un prototipo como se describe en el pattern *prototype* [GHJV95]. La nueva herramienta extiende el método `createFigure()` para interactuar con la configuración del editor y la sintaxis abstracta del diagrama creando la hoja correspondiente en el DAG asociado al diagrama. La creación de la hoja dispara la computación de los atributos gráficos y semánticos; de estos atributos gráficos se obtiene la figura que es insertada en el diagrama y el usario finalemente obtiene la respuesta visual esperada.

De forma similar, la construcción de un nodo comienza con la interacción del usuario al activar la herramienta de selección y luego seleccionar en el canvas las figuras que desea componer. Al presionar el botón derecho del mouse sobre el canvas se presentan al usario las reglas de composición permitidas en base a la selección. Esto es manejado por una extensión de la herramienta de selección `SelectionTool` brindada por Hotdraw, `DGSelectionTool`. Esta extensión colabora con el proceso de selección de reglas, que en base a la configuración del editor retorna las reglas aplicables, asignándole una acción a cada una. En base a la regla seleccionada y los diagramas, la acción es ejecutada. Esta acción toma los nodos y hojas seleccionadas y en base a

éstos ejecuta la construcción de un nuevo nodo del DAG, cuyos hijos serán los nodos y hojas asociados a las figuras seleccionadas en el cavas; el nuevo DAG es agregado a la sintaxis abastracta del diagrama. La construcción del nodo dispara la computación de sus atributos, de los cuales obtiene la figura que es agregada finalmente al canvas.

Es interesante notar lo que se mencionó anterioremente: la manipulación de diagramas tiene directa relación con la construcción de la sintaxis abstracta, y viceversa, sin que el usuario final del editor esté consciente de este proceso subyacente. Sin embargo, el diseñador del lenguaje visual cuenta con todo el poder del formalismo.

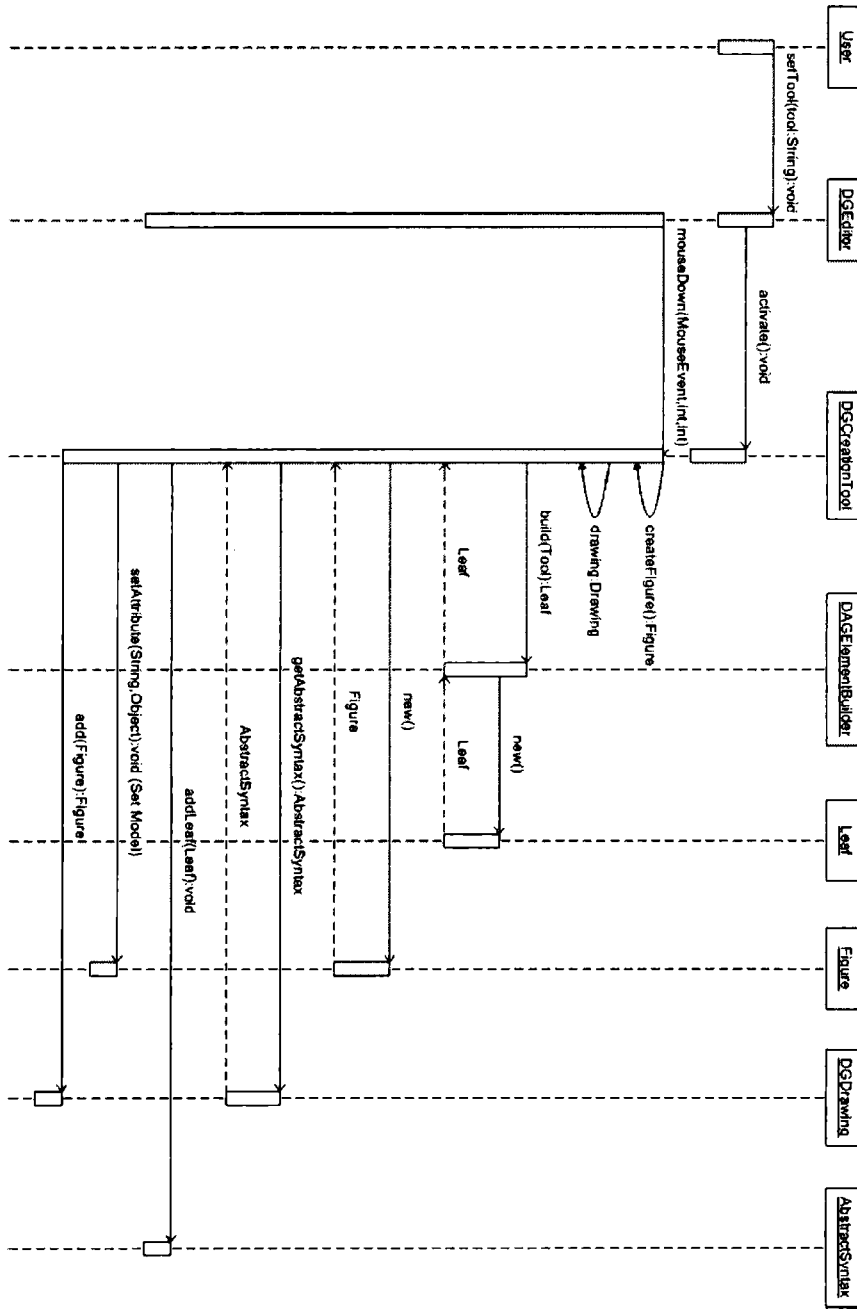

Figura 3.10: Proceso de creación de una hoja.

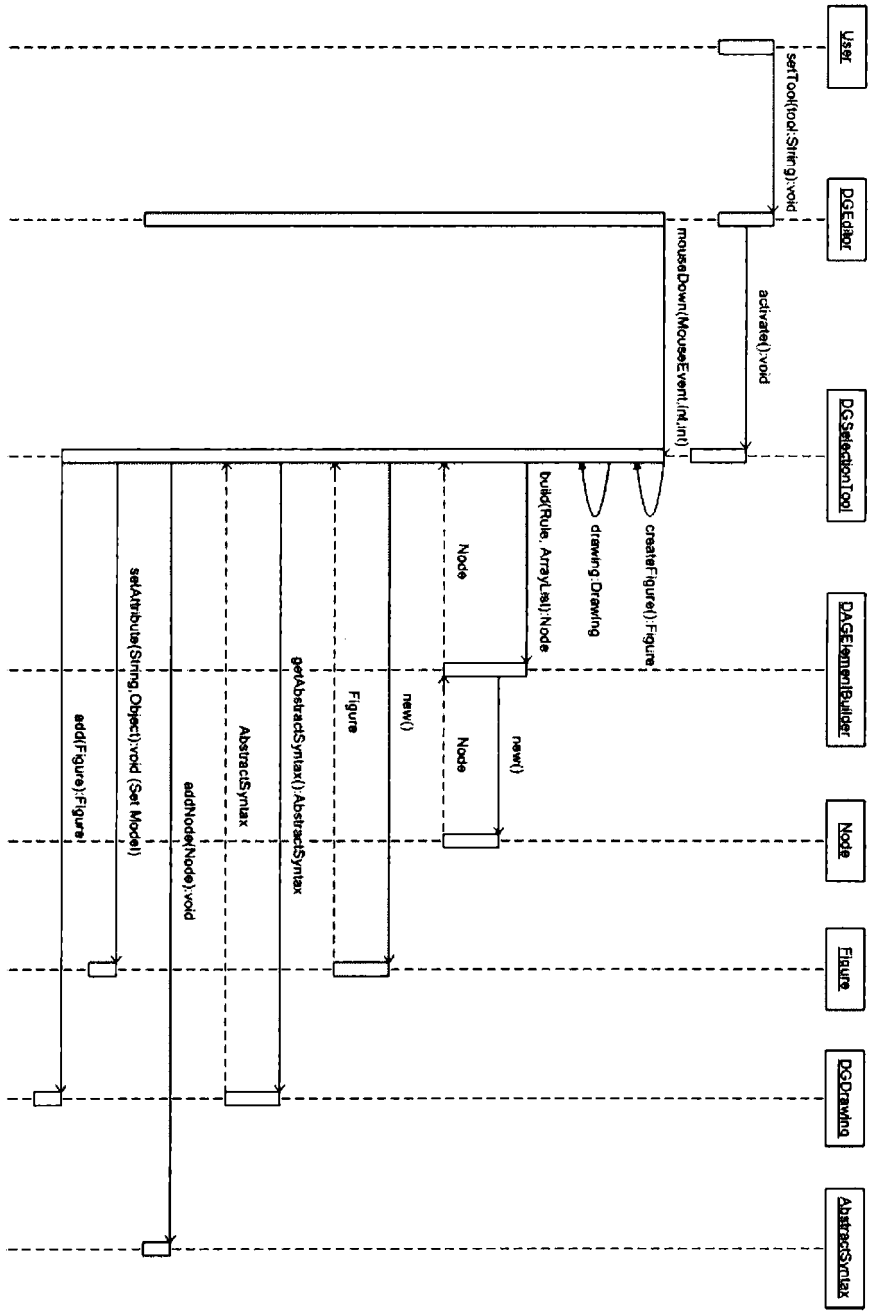

Figura 3.11: Proceso de creación de un nodo.

Capítulo 4

Estudio de Lenguajes Visuales

En este capítulo se presentarán en profundidad tres ejemplos prácticos. Por medio de estos ejemplos se verán las características salientes del formalismo presentado. Cada ejemplo se centrará en alguno de los puntos destacables del formalismo que dictaron su desarrollo. Además permitirá al lector seguir los pasos involucrados en el diseño de gramáticas de diagramas para lenguajes visuales. También se mostrará cómo, vía el editor parametrizable mostrado en el Capítulo 3, se obtiene una herramienta adecuada para la edición y manipulación de estos lenguajes visuales con esfuerzo adicional mínimo. En estos ejemplos sólo se presentará un esbozo de las gramáticas de diagramas que dan origen a los ejemplos con una sintaxis simplificada y sólo con los atributos relevantes en el contexto del ejemplo introducido; la versión completa de estos ejemplos en el formato XML aceptado por el editor parametrizable se puede encontrar en el Apéndice B.

4.1. Diagramas Didácticos

Generalmente, en el diseño de un lenguaje visual, se establece en primer lugar cuáles serán los diagramas básicos de nuestro lenguaje, asignándole a cada uno el significado de manera informal. Esto es conveniente realizarlo

mediante un dibujo en papel que permite ir dándole forma a la apariencia gráfica del lenguaje. Así se definen en forma abstracta los terminales de la gramática, su apariencia y su significado, delineando cuales serán las figuras básicas que compondrán nuestro diagrama. Luego se definen las reglas de composición que permitirán construir diagramas a partir de otros. También en este caso es conveniente realizar en papel una serie de bosquejos que ayuden a entender cómo se irán componiendo los diagramas y qué clase de diagramas queremos construir. Esto permite definir las reglas de composición de la gramática. Otras veces, se quiere dar semántica a una clase de diagramas existentes, sin embargo también conviene, para definir la gramática adecuada y su semántica, empezar con un ejemplo en papel de diferentes diagramas a partir de los cuales razonar.

El ejemplo presentado en esta sección muestra una de las características primarias de las GD que facilitan enormemente la experimentación con lenguajes visuales. A lo largo de estos ejemplos se verá cómo, mediante la sola manipulación de los atributos visuales, se puede experimentar con la apariencia del diagrama. Si bien el ejemplo presentado es sencillo y su importancia práctica nula, su importancia didáctica es notoria. La simplicidad del ejemplo fue elegida deliberadamente con el fin de enfocar la atención en la cualidad que distingue a las DGs para expresar LVs. El ejemplo presentado corresponde a un editor didáctico para chicos que permite la edición de caras y la asignación de un nombre a éstas. La estructura general de la gramática utilizada para la parametrización del editor se puede observar en el Cuadro 4.1.

La primer variante permite la edición de caras *"occidentales"*, en el sentido de los rasgos. Cada cara está compuesta de dos ojos y una boca. La Fig. 4.1 muestra el editor para dichas caras. Se debe prestar atención principalmente al atributo *figure*, el cual, como resultado de su evaluación, brinda la apariencia gráfica del terminal o no terminal en cuestión. A lo largo de las tres versiones del ejemplo, se podrá ver cómo la ecuación que denota el valor

Terminales: {*Ojo, Boca*}
Noterminales: {*Cabeza*}
Símbolo inicial: {*Cabeza*}
Producciones: {

Ojo → {*figure = new OccidentalEye()*}
Boca → {*figure = new OccidentalMouth()*}
Cabeza → *Ojo Ojo Boca* {
nombre = JOptionPane.showInputDialog(null, "Nombre?");
figure = new RoundHead($1.figure, $2.figure, $3.figure, $$.name)
}

Cuadro 4.1: Gramática para la edición de Caras occidentales.

de dicho atributo varía para dar lugar a diferentes instanciaciones del editor para el mismo LV con distintas visualizaciones.

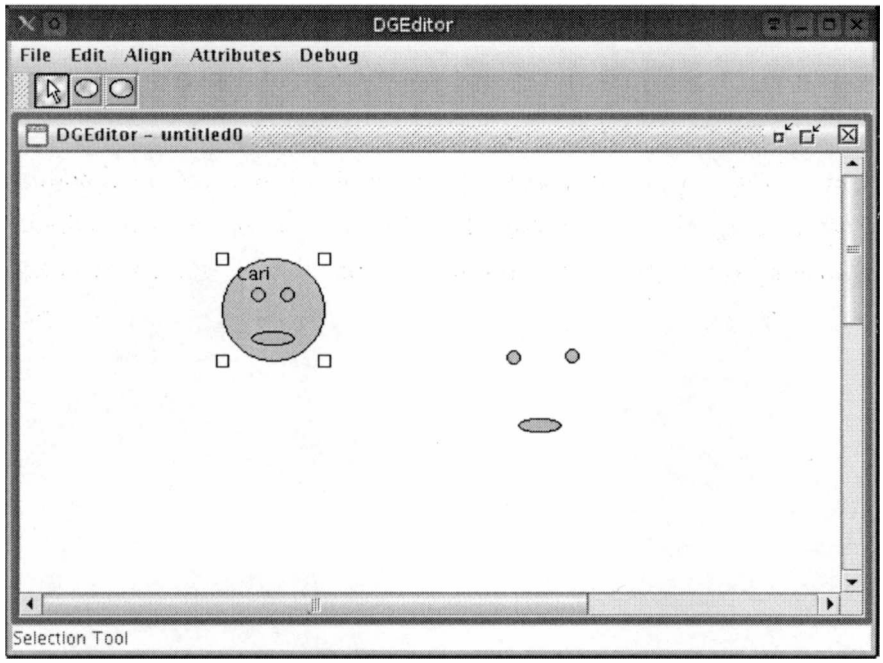

Figura 4.1: Editor de caras occidentales.

La segunda versión ejemplo utiliza la misma estructura de la gramática de caras mostrada anteriormente (Cuadro 4.2), ya que la semántica y estructura de los diagramas no cambia – sólo su apariencia gráfica. En este ejemplo podemos ver el mismo editor que ahora permite dibujar caras *"orientales"*.

En la Fig. 4.2 se puede ver en acción este nuevo editor.

```
Terminales:          {Ojo, Boca}
Noterminales:        {Cabeza}
Símbolo inicial:     {Cabeza}
Producciones:        {
  Ojo          →     {figure = new OrientalEye()}
  Boca         →     {figure = new OrientalMouth()}
  Cabeza       →     Ojo Ojo Boca {
                       nombre = JOptionPane.showInputDialog(null, "Nombre?");
                        figure = new RoundHead($1.figure, $2.figure, $3.figure, $$.name)
                     }
```

Cuadro 4.2: Gramática para la edición de caras orientales.

Por último se modifica el atributo *figure* (Cuadro 4.3) para obtener un nuevo editor a partir de la GD base que dio lugar a los editores de caras. Este nuevo editor es mostrado en la Fig. 4.3.

```
Terminales:          {Ojo, Boca}
Noterminales:        {Cabeza}
Símbolo inicial:     {Cabeza}
Producciones:        {
  Ojo          →     {figure = new RobotEye()}
  Boca         →     {figure = new RobotMouth()}
  Cabeza       →     Ojo Ojo Boca {
                       nombre = JOptionPane.showInputDialog(null, "Nombre?");
                        figure = new RectHead($1.figure, $2.figure, $3.figure, $$.name)
                     }
```

Cuadro 4.3: Gramática para la edición de caras robot.

En los variantes del ejemplo mostrado la mayor complejidad reside en la definición de los atributos gráficos. Sin embargo esto no es siempre así, sino que se debe a la simplicidad del ejemplo elegido. En la mayoría de los diagramas el modelo semántico suele tener mayor complejidad. En este ejemplo se mostró cómo mediante la manipulación de los atributos gráficos se puede experimentar con la apariencia de los diagramas sin necesidad de interferir con la semántica y estructura de los diagramas. También se podrían combinar las tres gramáticas en una sola mediante el renombre de terminales y noterminales, dando lugar a una GD para la edición de los tres tipos de caras simultaneamente en el mismo editor.

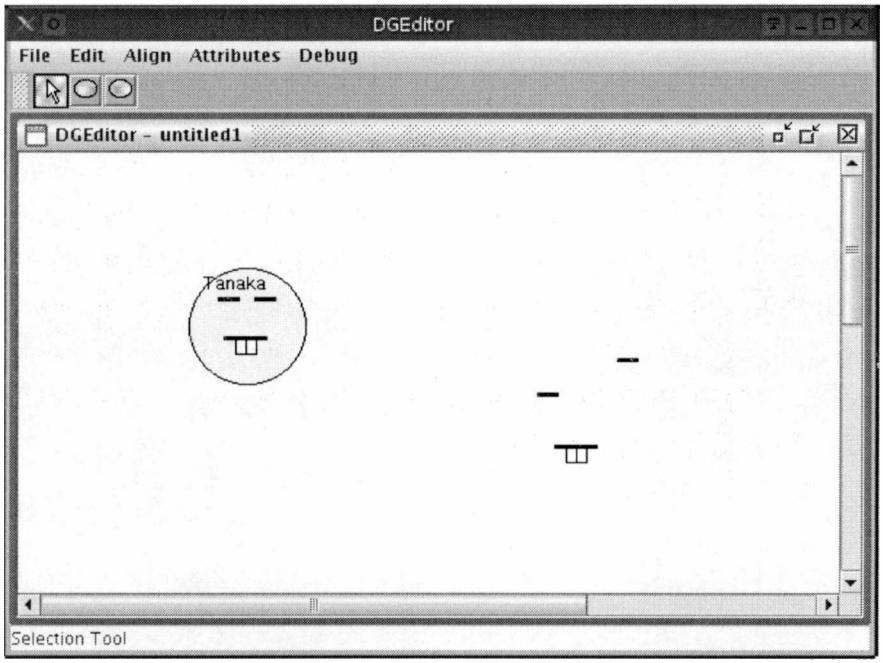

Figura 4.2: Editor de caras orientales.

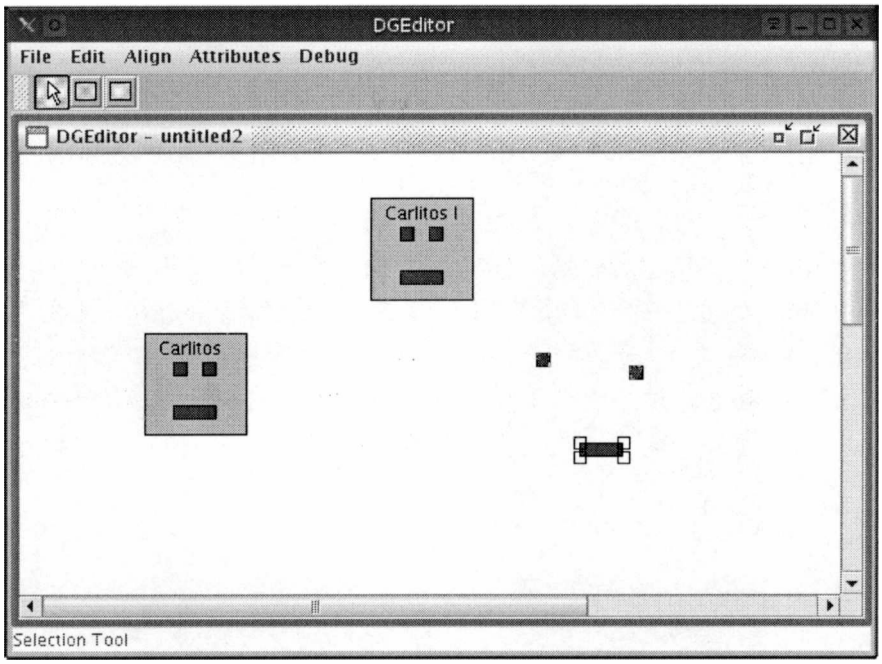

Figura 4.3: Gramática para la edición de Caras robots.

4.2. Expresiones Regulares

Nuestro segundo ejemplo reutiliza la gramática de diagramas de expresiones regulares introducida en el Capítulo 2. El propósito de volver sobre este ejemplo es hacer hincapié en otra de las ventajas principales del modelado de LVs mediante GDs: la posibilidad de modificar y experimentar con la semántica de un diagrama. Al igual que en la sección anterior se experimentó con distintos atributos gráficos que dieron origen a distintos editores con la misma estructura de diagramas, en esta sección veremos que lo mismo sucede con la semántica del diagrama. Esto se llevará a cabo mediante dos variantes del ejemplo: la primera volverá sobre el ejemplo de la Sección 2.3.1 que generaba código Haskell con combinadores de parsers para para determinar si un *String* está o no contenido en el lenguaje regular; la segunda versión, modificará su semántica para que genere una REGEX Perl. En la Figura 4.4 se vé el editor para REGEX de ambos ejemplos ya que la estructura de la GD y los atributos visuales no varían entre ambos.

La gramática del Cuadro 4.4 genera código Haskell para los combinadores de parser vistos en la sección 2.3.1. Puede resultar reiterativo introducir nuevamente esta gramática que sirvió de base a lo largo del desarrollo del las GDs; sin embargo volver sobre un ejemplo conocido nos permite poner foco en las características salientes del uso práctico de las GDs. El atributo *regex* va acumulando a lo largo de la edición el resultado del parser Haskell para la REGEX editada.

La siguiente variante introduce una modificación mínima a la gramática anterior, conservando sus atributos gráficos y estructura intacta. Como se puede observar en este caso el atributo semántico *regex* fue convenientemente modificado para que el mismo contenga en lugar del parser Haskell, el string que representa la REGEX correspondiente.

Convenientemente podríamos haber modificado el código intermedio que se genera a partir de la computación de los atributos semánticos de las GDs introducidas y a partir de éste, sin necesidad de modificar el editor, generar

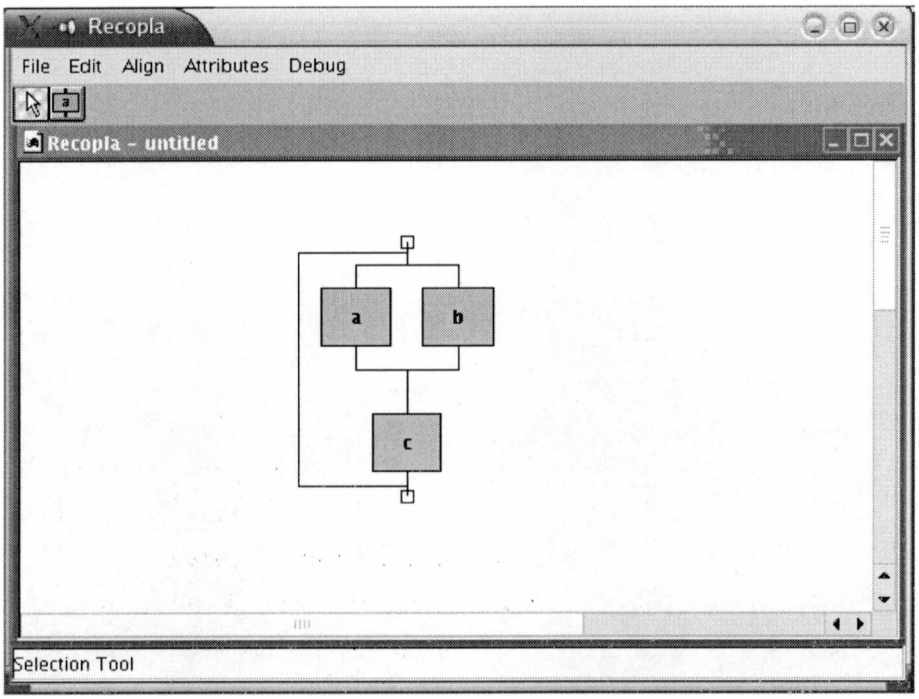

Figura 4.4: Editor de REGEX.

Terminales:		*{Símbolo}*	
Noterminales:		*{Regex, Concatenación, Alternativa, Repetición}*	
Símbolo inicial:		*{Regex}*	
Producciones:		*{*	
Símbolo	→	*{nombre = requestDialog "Nombre del Símbolo:"; ...}*	**(1)**
Regex	→	*Símbolo {regex = token $1.nombre; ...}*	**(2)**
Regex	→	*Alternativa {regex = $1.regex; ...}*	**(3)**
Regex	→	*Repetición {regex = $1.regex; ...}*	**(4)**
Regex	→	*Concatenación {regex = $1.regex; ...}*	**(5)**
Concatenación	→	*Regex Regex {regex = ($1.regex) < * > ($2.regex) < @ (\(x,y) → x ++ y); ...}*	**(6)**
Alternativa	→	*Regex Regex {regex = ($1.regex) < | > ($2.regex); ...}*	**(7)**
Repetición	→	*Regex {regex = many ($1.regex); ...}*	**(8)**
		}	

Cuadro 4.4: GD Regex modificada para la construcción de un parser.

Terminales:		{*Símbolo*}	
Noterminales:		{*Regex, Concatenación, Alternativa, Repetición*}	
Símbolo inicial:		{*Regex*}	
Producciones:		{	
Símbolo	→	{*nombre = requestDialog "Nombre del Símbolo:"; ...*}	**(1)**
Regex	→	*Símbolo* {*regex = $1.nombre; ...*}	**(2)**
Regex	→	*Alternativa* {*regex = $1.regex; ...*}	**(3)**
Regex	→	*Repetición* {*regex = $1.regex; ...*}	**(4)**
Regex	→	*Concatenación* {*regex = $1.regex; ...*}	**(5)**
Concatenación	→	*Regex Regex* {*regex = ($1.regex)($2.regex); ...*}	**(6)**
Alternativa	→	*Regex Regex* {*regex = ($1.regex) | ($2.regex); ...*}	**(7)**
Repetición	→	*Regex* {*regex = ($1.regex)*; ...*}	**(8)**
		}	

Cuadro 4.5: GD Regex modificada para la construcción de la REGEX Perl.

los distintos fragmentos de código Haskell y Perl. Esto en general es una buena práctica de diseño y veremos su importancia en la siguiente sección.

Por otro lado también es posible cambiar la estructura de la gramática sin modificar su semántica. Como hemos visto en este ejemplo para poder editar una *Repetición*, *Concatenación* o *Alternativa* primero siempre se debe editar una *Regex*. Modificando la gramática podríamos llegar a la mostrada en el Cuadro 4.6.

Terminales:		{*Símbolo*}	
Noterminales:		{*Regex, Concatenación, Alternativa, Repetición*}	
Símbolo inicial:		{*Regex*}	
Producciones:		{	
Símbolo	→	{*nombre = requestDialog "Nombre del Símbolo:"; ...*}	**(1)**
Concatenación	→	*Símbolo Símbolo* {*regex = $1.nombre$2Símbolo; ...*}	**(2)**
Concatenación	→	*Símbolo Concatenación* {*regex = $1.regex($2.regex); ...*}	**(3)**
Concatenación	→	*Símbolo Repetición* {*regex = $1.regex$(2.regex); ...*}	**(4)**
Concatenación	→	*Símbolo Alternativa* {*regex = $1.regex($2.regex); ...*}	**(5)**
Concatenación	→	*Concatenación Concatenación* {*regex = ($1.regex)($2.regex); ...*}	**(7)**
Concatenación	→	*Concatenación Repetición* {*regex = ($1.regex)($2.regex); ...*}	**(8)**
Concatenación	→	*Concatenación Alternativa* {*regex = ($1.regex)($2.regex); ...*}	
		...	
		}	

Cuadro 4.6: GD Regex modificada para la simplificar la edición.

Como puede observarse, modificar la gramática con motivo de simplificar la edición puede resultar tedioso e incluso complicarla innecesariamente.

Contando con un editor maduro podría evitarse esto por medio de inferencia automática de múltiples reglas. Por ejemplo, seleccionando un *Símbolo* y una *Regex* construida a partir de una *Repetición*, el editor automáticamente podría ofrecernos como alternativas construir una *Regex* a partir de una *Concatenación*, *Repetición* o *Alternativa*, de manera tal que el proceso que antes el usuario hacía manualmente vía el editor sucedería automáticamente sin su intervención.

4.3. Diagramas UML

En este ejemplo se verá una gramática para un subconjunto de UML. Este ejemplo es importante debido a su aplicación práctica y por tratarse de un ejemplo de mayor relevancia que los repasados hasta el momento. El mismo está siendo utilizado para el desarrollo de una herramienta que permite la generación automática de código Java para una herramienta que mapea objetos Java a bases de datos relacional y viceversa, similar a Toplink[1], Enterprise Objects Framework[2] (EOF) Hibernate[3] o Cayenne[4], entre otros.

Algunos de los productos mencionados brindan herramientas gráficas que asisten a la construcción del mapeo entre un modelo de objetos *Java* o *Smalltalk* a base de datos relacional, pero cada una de estas es de propósito específico. En este ejemplo se muestra una herramienta que genera automáticamente el código necesario para el mapeo de objetos Java. Como se ha visto, se podría modificar para que generase código en otro lenguaje, frameworks de mapeo o incluso más de uno simultáneamente; podría extenderse para que éste genere también la estructura relacional en la correspondiente base de datos. Sin embargo, modificar los atributos semánticos de la GD para generar código en varios lenguajes o el SQL necesario para la generación de

[1] `http://otn.oracle.com/products/ias/toplink/index.html`
[2] `http://developer.apple.com/documentation/WebObjects/Enterprise_Objects/`
[3] `http://www.hibernate.org/`
[4] `http://www.objectstyle.org/cayenne`

tablas en una base de datos relacional es poco práctico. En estos casos es conveniente separar la generación de código de la generación de modelo. Por este motivo la GD provista sólo generará una representación intermedia del diagrama editado, la cual será conveniente para que un proceso posterior genere a partir de esta estructura código de mapeo para distintos lenguajes o frameworks de mapeo convenientemente sin necesidad de modificar la GD. La Figura 4.5 muestra un ejemplo del editor de diagramas de clases estático UML.

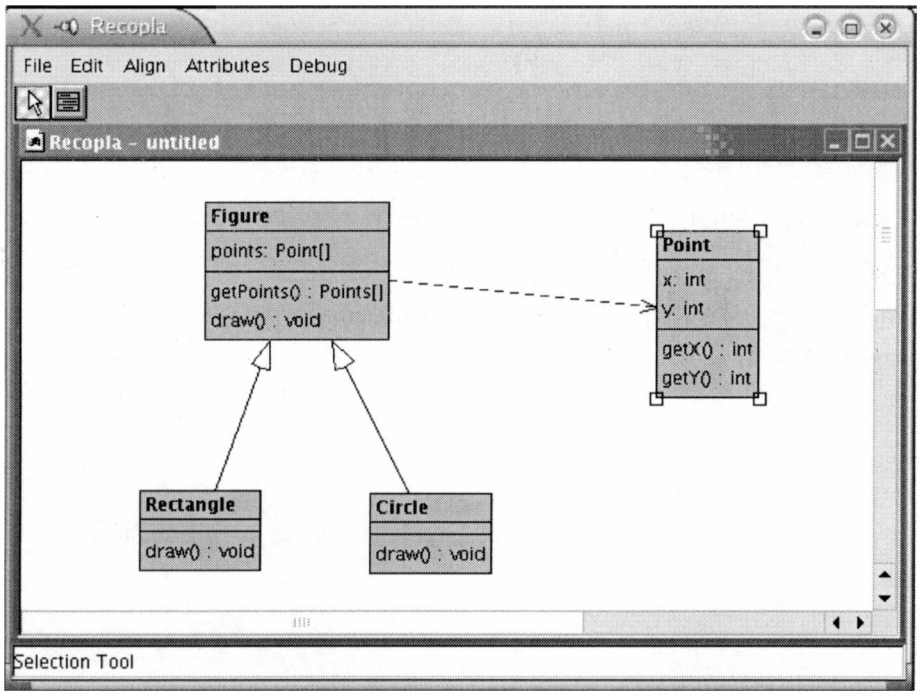

Figura 4.5: Editor de diagramas de clases UML.

La GD del Cuadro 4.7 es utilizada para la parametrización del editor. La GD presentada es un subconjunto de los diagramas de clases estáticos UML, que sólo permite la edición de clases, jerarquías y asociaciones. Sin embargo nuevas construcciones pueden ser agregadas sólo extendiendo las producciones de la GD y el modelo semántico del diagrama para soportarlas.

Terminales:		{*Class, Association*}	
Noterminales:		{*UML*}	
Símbolo inicial:		{*Regex*}	
Producciones:		{	
Class	→	{*figure = new ClassFigure();*	**(1)**
		model = new ClassModel();	
		properties = new ClassPropertiesDialog().open();..}	
Association	→	*Class Class {figure = new AssocConnection($1.figure, $2.figure);*	**(2)**
		model = $1.model.addAssociation($2.model);	
		properties = new AssociationProperties().open();...}	
Hierarchy	→	*Class Class {figure = new HierarchyConnection($1.figure, $2.figure);*	**(3)**
		model = $1.model.setParent($2.model); ...}	
		}	

Cuadro 4.7: GD Regex para la construcción de diagramas UML.

La gramática consta del terminal *Class*, cuyo modelo es la clase *ClassModel*, que modela los diagramas UML y consta de los atributos: *parent*, *methods*, *fields* y *associations*, conteniendo la superclase, los métodos, campos y asociaciones respectivamente. En la Figura 4.6 se encuentra el diagrama estático de clases utilizado para el modelo semántico de los diagramas UML. Cuando se obtienen las propiedades de un diagrama *Class* se permite la edición (agregado, modificación y borrado) de los atributos de la clase (métodos, campos). Luego existen dos reglas de composición: *Hierarchy* que permite asignarle una superclase a una clase dada, con la consecuente actualización del modelo (regla 3) y *Association* que permite crear una asociación entre dos clases. Cuando se editan las propiedades de una asociación se le puede asignar un campo existente de la clase origen que concuerde con el tipo de la clase destino o un nuevo campo. Las asociaciones son particularmente importantes ya que al generar el código de mapeo correspondiente a una clase conteniendo éstas, los atributos involucrados son reemplazados por *proxies* que difieren la búsqueda en la base de datos hasta que el campo es accedido.

En el ejemplo mostrado, la clara separación entre la semántica y la visualización del diagrama permite de manera independiente experimentar con distintos frameworks de mapeo objetos↔relacional subyacente utilizando la misma herramienta de edición gráfica. La separación del modelo generado por

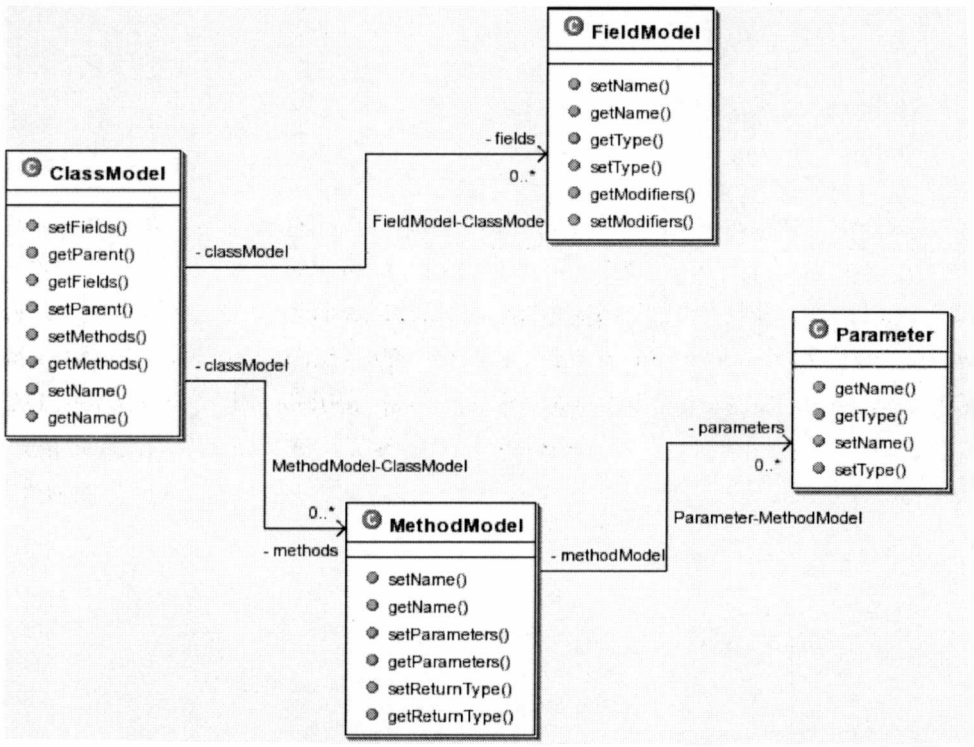

Figura 4.6: Modelo para diagramas UML.

el editor y la generación de código final fue arbitraria. La decisión obedece al sentido común y está basada en buenas prácticas de diseño que promueven la separación clara de dominios para obtener un mayor grado de reusabilidad. Esto último muestra cómo las pautas de diseño de los atributos de una GD no varían de las pautas establecidas para cualquier lenguaje textual. Es conocido en la implementación de compiladores que a partir del código fuente se puede producir en un solo paso código ejecutable; sin embargo, esto no sólo es extremadamente complejo para el análisis semántico, sino que muchas optimizaciones de código intermedias son impracticables. En el área mencionada, es común el uso de varias fases (análisis sintáctico, parsing, análisis semántico, asignación de registros, etc.) con sus correspondientes estructuras de datos intermedias sobre las cuales luego se trabaja. La generación de código de mapeo fue omitida deliberadamente debido a que no reviste mayor importancia en el análisis de este trabajo.

Capítulo 5

Conclusiones y Trabajo Futuro

Durante este trabajo se ha presentado un herramienta formal para describir en forma abstracta la estructura de ciertos tipos de diagramas, las gramáticas de diagramas. También se mostró la utilización del formalismo presentado como la base para la definición de un prototipo de herramienta de edición gráfica configurable que permite dibujar y experimentar con distintos tipos de diagramas instanciándola con la gramática adecuada. De esta forma, para obtener un editor para una nueva clase de diagramas basta con definir la gramática que describe el lenguaje visual, en lugar de programar un nuevo editor desde el comienzo. La inmediata aplicación fue una de las principales motivaciones en la definición de las gramáticas de diagramas. Previo a este trabajo, la construcción de editores para distintos tipos de diagramas debía desarrollarse en la forma tradicional, aprovechando sólo frameworks y componentes reusables, mientras que la semántica y evaluación del diagrama debía incorporarse totalmente de manera ad-hoc. La experimentación con lenguajes visuales y herramientas se tornaba, de esta forma, poco productiva y se perdía el foco de ésta, agregándose a la complejidad inherente del desarrollo de un lenguaje visual la complejidad de la implementación de la herramienta. Una de las características notables de este trabajo, que hemos visto a lo largo de los Capítulos 3 y 4, es que la definición de un nuevo lenguaje visual no

requiere de un conocimiento exhaustivo del formalismo presentado. Durante los capítulos mencionados se utilizaron términos comunes en el dominio de las herraminetas visuales como "barra de herramientas" y "reglas de composición". Si bien un conocimiento del formalismo ayuda en la definición y comprensión de un lenguaje visual, esto no es estrictamente indispensable para la utilización del prototipo.

Las GDs no imponen ni proveen una forma explícita para expresar la sintaxis visual de un diagrama; en una GD, la sintaxis visual está dictada y contenida por los atributos visuales, siendo esto dependiente del lenguaje de atributos elegido y por lo tanto pudiendo variar según la conveniencia del usuario del formalismo. Existen modelos sintácticos para lenguajes visuales que permiten describir sentencias visuales, como *String*, *Plex*, *Graph* y *Box*. Estos modelos son utilizados con éxito por otros formalismos [CP00] para la especificación de la sintaxis visual de los lenguajes. Las GD pueden ser extendidas para soportar estos modelos sintácticos de forma tal de simplificar el desarrollo de una GD para un diagrama específico, al menos en lo que a la apariencia se refiere.

El enfoque presentado fuerza a que el método de construcción sea *orientado a sintaxis*. Muchas veces este método de construcción de diagramas puede ser tedioso. Pero en muchas oportunidades, y dependiendo de las facilidades provistas por la herramienta, puede ser ventajoso que en cada paso el usuario pueda construir sólo subdiagramas correctos. Por otro lado, creemos (si bien no fue probado formalmente) que esta desventaja no representa una disminución en el poder expresivo de las gramáticas de diagramas – esto es, no se disminuye la cantidad de diagramas que se pueden describir. Otra línea de investigación es la de una extensión que permita edición libre y automáticamente se infiera que producción se debe aplicar. Esta idea se basa en el uso de atributos *constraints*. Alternativas que permiten el tipo de edición libre son exploradas en [KM00] y [HW96]. Por otro lado, es conocido que editores para lenguajes textuales son potenciados con herramientas como *"syntax highligh-*

ting" y *"auto-complcción"* que facilitan el proceso de edición sin que esto afecte el formalismo subyacente (incluso dichas herramientas son construídas basadas en este último). El editor presentado u otros basados en las GD, de igual forma, podrían enriquecer el proceso de edición mediante el agregado de características tales como inferencia automática de reglas, herramientas de conexión, entre otras, sin afectar por ello el formalismo.

Una de las ventajas más importantes, que se mostró especialmente en el Capítulo 4 durante los ejemplos presentados, fue la clara separación entre la semántica y la apariencia visual del diagrama, lo cual nos permite de manera totalmente independiente experimentar con distintas variantes tanto para la semántica como para la apariencia de los diagramas. En el ejemplo presentado del editor UML este resultado es de vital importancia, ya que si hubiésemos cambiado el framework de mapeo objetos↔relacional (que comunmente sucede en pos de una mejora de las herramientas utilizadas) hubiésemos tenido que descartar el editor que es vital para el desarrollo de código de mapeo ya que esta tarea es generalmente tediosa y propensa a errores.

Otras de las alternativas que pueden ser explotadas es la posibilidad de potenciar herramientas profesionales, como el Visio[1], con las GD. Esto brindaría toda la potencia y calidad de un editor como el Visio más la posibilidad de validar la construcción de diagramas y agregarle semántica de forma natural y no de manera ad-hoc como lo es actualmente. Existen ya experiencias positivas de esta combinación que alientan a seguir esta alternativa, como el editor de circuitos eléctricos mencionado en [LMH98, MCL98][2].

Por último uno de los puntos interesantes que quedaron fuera del alcance de esta tesis fue el estudio de técnicas avanzadas para la computación eficiente de atributos en GAs y la utilización de GAs de alto orden para el manejo de sub-gramáticas de diagramas con el fin de extender la experiencia de edición, como los presentadas en los trabajos [SSK00, SV91].

---

[1] `http://www.microsoft.com/office/visio/`

[2] `http://www.cse.ogi.edu/pacsoft/projects/VisualHawk/`

Apéndice A

DTD del Archivo de Definición

Un archivo DTD (Document Type Definition) permite definir qué tipos de *elementos* (tags) pueden utilizarse en un archivo XML y cuál es su estructura lógica. Existen dos formas de chequear la consistencia de un archivo XML: la primera consiste en chequear si el XML está bien formado de acuerdo a la especificación [BPV04], lo cual consiste básicamente en verificar que cada elemento abierto en el documento (`<elem>`) tenga su correspondiente fin (`</elem>`). La otra, y más importante, es la validación lógica del documento respecto a la especificación del DTD asignado al documento. La asignación de un DTD a un documento XML es opcional.

En la Fig. A.1 se puede observar el DTD utilizado para validar los archivos de configuración aceptados por el editor parametizable.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT component (#PCDATA)>
<!ELEMENT components (component+)>
<!ELEMENT compositionrules (rule+)>
<!ELEMENT diagram (toolbar, start, compositionrules)>
<!ELEMENT figure (#PCDATA)>
<!ELEMENT icon (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT properties (#PCDATA)>
<!ELEMENT rule
 (name, components, icon, figure, domain_attributes)
>
<!ELEMENT start (#PCDATA)>
<!ELEMENT tool
 (name, icon, tooltip, figure, properties, domain_attributes)
>
<!ELEMENT toolbar (tool+)>
<!ELEMENT tooltip (#PCDATA)>
<!ELEMENT domain_attributes (attribute*)>
<!ELEMENT attribute (name, value)>
<!ELEMENT value (#PCDATA)>
```

Figura A.1: DTD de validación.

Apéndice B

Compendio de Gramáticas

A continuación se ilustran una serie de gramáticas útiles completas en el formato de entrada aceptado editor parametrizable (XML).

B.1. Ejemplo Didáctico (Caras)

Este ejemplo trivial muestra un editor didáctico para chicos que permite la edición de caras con la posibilidad de asignarle un nombre. Correspondiente al ejemplo presentado en el Capítulo 4, Sección 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--DOCTYPE diagram SYSTEM "file:///working/dg/dgeditor.dtd"-->
<!--
Terminals:
    {Symbol}

Nonterminals:
    {Eye, Mouth}

Start Symbol: Head

Productions: {
    Head -> Eye Eye Mouth
```

```
}
-->

<diagram>
  <toolbar>
    <tool>
      <name>Mouth</name>
      <icon>/CH/ifa/draw/images/ELLIPSE</icon>
      <tooltip>New Mouth...</tooltip>
      <figure>
        import ar.edu.dgeditor.figures.faces.OrientalMouth;

        new OrientalMouth();
      </figure>
      <properties/>
      <domain_attributes/>
    </tool>
    <tool>
      <name>Eye</name>
      <icon>/CH/ifa/draw/images/ELLIPSE</icon>
      <tooltip>New Eye...</tooltip>
      <figure>
        import CH.ifa.draw.figures.RectangleFigure;
        import ar.edu.dgeditor.figures.FixedSizeFigure;
        import java.awt.Color;

        FixedSizeFigure eye =
            new FixedSizeFigure(
            new RectangleFigure(),
            15,
            3);
        eye.setFillColor(Color.BLACK);

        eye;
      </figure>
      <properties/>
      <domain_attributes/>
    </tool>
```

```
</toolbar>

<start>Head</start>

<compositionrules>
  <rule>
    <name>Head</name>
    <components>
      <component>Eye</component>
      <component>Eye</component>
      <component>Mouth</component>
    </components>
    <icon>/ar/edu/dgeditor/images/regex.gif</icon>
    <figure>
      import CH.ifa.draw.framework.Figure;
      import ar.edu.dgeditor.figures.faces.FaceFigure;
      import java.awt.Color;

      FaceFigure face =
          new FaceFigure(
              (Figure)$1.figure,
              (Figure)$2.figure,
              (Figure)$3.figure);

      face.setFillColor(Color.YELLOW);

      face;
    </figure>
    <domain_attributes>
      <attribute>
        <name>face_name</name>
        <value>
          import javax.swing.*;
          import ar.edu.dgeditor.abstractsyntax.*;
          import ar.edu.dgeditor.figures.regex.*;

          String name =
              JOptionPane.showInputDialog(
```

```
                    null,
                    "Name?");

            name == null ? "" : name;
          </value>
        </attribute>
      </domain_attributes>
    </rule>
  </compositionrules>
</diagram>
```

B.2. Expresiones Regulares

La siguiente gramática permite editar visualmente expresiones regulares como se describió en la Sección 4.2 del Capítulo 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--DOCTYPE diagram SYSTEM "file:///working/dg/dgeditor.dtd"-->
<!--
Terminals:
    {Symbol}

Nonterminals:
    {Regex, Concatenation, Alternation, Repetition}

Start Symbol: Regex

Productions: {
    Regex         -> Symbol
    Regex         -> Alternation
    Regex         -> Repetition
    Regex         -> Concatanation
    Concatenation -> Regex Regex
    Alternation   -> Regex Regex
    Repetition    -> Regex
}
```

```
-->

<diagram>
  <toolbar>
    <tool>
      <name>Symbol</name>
      <icon>/ar/edu/dgeditor/images/symbol</icon>
      <tooltip>New symbol...</tooltip>
      <figure>
        import ar.edu.dgeditor.figures.regex.SymbolFigure;
        import CH.ifa.draw.figures.RectangleFigure;

        new SymbolFigure(
            new RectangleFigure(),
            (String)$$.symbol_name);
      </figure>
      <properties>
        import javax.swing.*;
        import ar.edu.dgeditor.abstractsyntax.*;
        import ar.edu.dgeditor.figures.regex.*;

        String sym =
            JOptionPane.showInputDialog(
                null,
                "Symbol name?");

        if (sym != null)
            $$.symbol_name = sym;

        SymbolFigure sf = $$.figure;
        sf.setSymbolName(sym);
      </properties>
      <domain_attributes>
        <attribute>
          <name>symbol_name</name>
          <value>"?";</value>
        </attribute>
      </domain_attributes>
```

```
    </tool>
  </toolbar>

  <start>Regex</start>

  <compositionrules>
    <rule>
      <name>Regex</name>
      <components>
        <component>Symbol</component>
      </components>
      <icon>/ar/edu/dgeditor/images/regex.gif</icon>
      <figure>$1.figure;</figure>
      <domain_attributes>
        <attribute>
          <name>regex</name>
          <value>
            "token \"" + $1.symbol_name + "\"";
          </value>
        </attribute>
      </domain_attributes>
    </rule>
    <rule>
      <name>Regex</name>
      <components>
        <component>Alternation</component>
      </components>
      <icon>'regex.gif'</icon>
      <figure>$1.figure;</figure>
      <domain_attributes>
        <attribute>
          <name>regex</name>
          <value>$1.regex;</value>
        </attribute>
      </domain_attributes>
    </rule>
    <rule>
      <name>Regex</name>
```

```
    <components>
      <component>Repetition</component>
    </components>
    <icon>'regex.gif'</icon>
    <figure>$1.figure;</figure>
    <domain_attributes>
      <attribute>
        <name>regex</name>
        <value>$1.regex;</value>
      </attribute>
    </domain_attributes>
  </rule>
  <rule>
    <name>Regex</name>
    <components>
      <component>Concatenation</component>
    </components>
    <icon>'regex.gif'</icon>
    <figure>$1.figure;</figure>
    <domain_attributes>
      <attribute>
        <name>regex</name>
        <value>$1.regex;</value>
      </attribute>
    </domain_attributes>
  </rule>
  <rule>
    <name>Concatenation</name>
    <components>
      <component>Regex</component>
      <component>Regex</component>
    </components>
    <icon>purple-m.gif</icon>
    <figure>
      import ar.edu.dgeditor.figures.regex.*;

      new ConcatenationFigure(
          (RegexFigure)$1.figure,
```

```
        (RegexFigure)$2.figure);
    </figure>
    <domain_attributes>
      <attribute>
        <name>regex</name>
        <value>
          "("
              + $1.regex
              + " &lt;*&gt; "
              + $2.regex
              + " &lt;@ (\\(x,y) -&gt; x ++ y))";
        </value>
      </attribute>
    </domain_attributes>
  </rule>
  <rule>
    <name>Alternation</name>
    <components>
      <component>Regex</component>
      <component>Regex</component>
    </components>
    <icon>'alter.gif'</icon>
    <figure>
      import ar.edu.dgeditor.figures.regex.*;

      new AlternationFigure(
          (RegexFigure)$1.figure,
          (RegexFigure)$2.figure);
    </figure>
    <domain_attributes>
      <attribute>
        <name>regex</name>
        <value>
          "("
              + $1.regex
              + " &lt;|&gt; "
              + $2.regex
              + ")";
```

```
          </value>
        </attribute>
      </domain_attributes>
    </rule>
    <rule>
      <name>Repetition</name>
      <components>
        <component>Regex</component>
      </components>
      <icon>'rep.gif'</icon>
      <figure>
        import ar.edu.dgeditor.figures.regex.*;

        new RepetitionFigure((RegexFigure)$1.figure);
      </figure>
      <domain_attributes>
        <attribute>
          <name>regex</name>
          <value>
            "((many"
                + " ("
                + $1.regex
                + ")) &lt;@ \\xs -&gt; concat xs)";
          </value>
        </attribute>
      </domain_attributes>
    </rule>
  </compositionrules>
</diagram>
```

B.3. Diagramas UML

La siguiente es una gramática para un subconjunto de UML, como fuera presentado en el ejemplo del Capítulo 4, Sección 4.3.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!--DOCTYPE diagram SYSTEM "file:///working/dg/dgeditor.dtd"-->
<!--
Terminals:
    {Class}

Nonterminals:
    {Hierarchy, Asociation}

Start Symbol: Hierarchy, Asociation, Class

Productions: {
    Hierarchy          -> Class Class
    Asociation         -> Class Class
}
-->

<diagram>
  <toolbar>
    <tool>
      <name>Class</name>
      <icon>/images/CLASS</icon>
      <tooltip>New Class...</tooltip>
      <figure>
        import jmodeller.tools.ClassFigure;

        new ClassFigure();
      </figure>
      <properties/>
      <domain_attributes/>
    </tool>
  </toolbar>

  <start>Hierarchy, Asociation, Class</start>

  <compositionrules>
    <rule>
      <name>Hierarchy</name>
      <components>
```

```
      <component>Class</component>
      <component>Class</component>
    </components>
    <icon>/ar/edu/dgeditor/images/hierarchy.gif</icon>
    <figure>
      import CH.ifa.draw.framework.Figure;
      import jmodeller.tools.InheritanceLineConnection;
      import ar.edu.dgeditor.figures.RecoplaConnector;

      Figure figure1 = (Figure) $1.figure;
      Figure figure2 = (Figure)$2.figure;
      InheritanceLineConnection asoclc =
          new InheritanceLineConnection();
      RecoplaConnector ct =
          new RecoplaConnector(asoclc);

      ct.connectionStart(
          figure1, figure1.center().x,
          figure1.center().y);
      ct.connectionEnd(
          figure2,
          figure2.center().x,
          figure2.center().y);

      ct.getAddedFigure();
    </figure>
    <domain_attributes/>
  </rule>
  <rule>
    <name>Asociation</name>
    <components>
      <component>Class</component>
      <component>Class</component>
    </components>
    <icon>/ar/edu/dgeditor/images/asociation.gif</icon>
    <figure>
      import CH.ifa.draw.framework.Figure;
      import jmodeller.tools.DependencyLineConnection;
```

```
        import ar.edu.dgeditor.figures.RecoplaConnector;

        Figure figure1 = (Figure) $1.figure;
        Figure figure2 = (Figure)$2.figure;
        DependencyLineConnection dlc =
            new DependencyLineConnection();
        RecoplaConnector ct = new RecoplaConnector(dlc);

        ct.connectionStart(
            figure1,
            figure1.center().x,
            figure1.center().y);
        ct.connectionEnd(
            figure2,
            figure2.center().x,
            figure2.center().y);

            ct.getAddedFigure();
      </figure>
      <domain_attributes/>
    </rule>
  </compositionrules>
</diagram>
```

Apéndice C

BNF Referencia de Atributos

La siguiente es la especificación BNF de la sintaxis utilizada para referenciar atributos en una GD. Si bien en la Sección 3.5 se utilizó Java como lenguaje de especificación de atributos, la sintaxis para referencias a atributos en una GD es más general y puede adecuarse a cualquier lenguaje elegido para denotar las expresiones de los atributos.

<ATTR_REF> ::= **$**<NUM>.<ID> | **$$.**<ID>
<ATTR_ASGN> ::= <ATTR_REF> = <EXPR>
<NUM> ::= <NUM'> <NUM> | <NUM'>
<NUM'> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
<ID> ::= <ID'> <ID> | <NUM'>
<ID'> ::= _ | **a** .. **z** | **A** .. **Z** | **0** .. **9** | **'** | **-** | ...
<EXPR> ::=

El noterminal EXPR quedó sin definir y puede ser cualquier expresión válida acorde al lenguaje de atributos elegido. En el caso de este trabajo, el lenguaje de atributos elegido fué Java, por lo cual EXPR es cualquier expresión Java válida en el contexto de la asignación del atributo.

Bibliografía

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison–Wedsley, 1983.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[BPV04] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[Bra95] J. M. Brant. HotDraw. Master's thesis, University of Illinois at Urbana-Champaign, 1995.

[CGN+91] C. Crimi, A. Guercio, G.Ñota, G. Pacini, G. Tortora, and M. Tucci. Relation grammars and their application to multidimentional languages. *Journal of Visual Languages and Computing*, 2:333–446, 1991.

[Che71] T. E. Cheatham. The recent evolution of programming languages. In *1971 IFIP Congress*, pages 298–313. IFIP, 1971.

[Che76a] P. P. Chen. The entity relationship model - toward an unified view of data. *ACM Trans. on Database Sys.*, 1(1):9, March 1976. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

[Che76b] P. P. Chen. "The Entity-Relationship Model". *ACM Trans. on Database Systems (TODS)*, 1:9–36, 1976.

[CLOT97] Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Genoveffa Tortora. A parsing methodology for the implementation of visual systems. *IEEE Transactions on Software Engineering*, 23(12):777–799, December 1997.

[CP00] Gennaro Costagliola and Giuseppe Polese. Extended positional grammars. In *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 103. IEEE Computer Society, 2000.

[Eck00] Bruce Eckel. *Thinking in Java.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 2000.

[Fok95] Jeroen Fokker. Functional parsers. *Lecture Notes in Computer Science*, 925:1–23, 1995.

[FPS+96] F. Ferrucci, G. Pacini, G. Satta, M. I. Sessa, G. Tortora, M. Tucci, and G. Vitiello. Symbol-relation grammars: A formalism for graphical languages. *Information and Computation*, 131(1):1–46, 25Ñovember 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley Professional Computing Series. Addison Wesley, 1995. http://www.aw.com.

[Gol91] E. J. Golin. Parsing visual languages with picture layout grammars. *Journal of Visual Languages and Computing*, 2:1–23, 1991.

[GR83] A. Goldberg and D. Robson. *Smalltalk 80. The Language and its implementation.* Addison–Wesley, 1983.

[HJW+92] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the programming language haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[HPF99] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98. October 1999.

[HU80] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, N. Reading, MA, 1980.

[HW96] V. Haarslev and M. Wessel. GenEd: An Editor with Generic Semantics for Formal Reasoning about Visual Notations. In *1996 IEEE Symposium on Visual Languages*, pages 204–211, Boulder, Colorado, USA, September 1996. IEEE Computer Society Press. Los Alamitos, September 1996.

[Joh92a] Ralph Johnson. HotDraw (abstract): A structured drawing editor framework for Smalltalk. In *Addendum to the Proceedings of the Conference on Object-Oriented Programming, Systems, Lanuages, and Applications*, page 232, 1992.

[Joh92b] Ralph E. Johnson. Documenting frameworks using patterns. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 63–72, New York, NY, 1992. ACM Press.

[KM00] O. Köth and M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In *Procee-*

dings International Workshop on Graph Transformation (GraTra 2000), Berlin, 2000.

[Lin01] Peter Linz. *An Introduction to Formal Languages and Automata.* Jones and Bartlett, 3rd edition, 2001.

[LMB92] J. Levine, T. Mason, and Doug Brown. *lex & yacc, 2nd Edition.* O'Reilly & Associates, 1992.

[LMH98] Daan Leijen, Erik Meijer, and James Hook. Haskell as an automation controller. In *Advanced Functional Programming*, pages 268–289, 1998.

[Mar94] K. Marriott. Constraint multiset grammars. In Allen L. Ambler and Takayuki Dan Kimura, editors, *Proceedings of the Symposium on Visual Languages*, pages 118–127, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[MC97] Erik Meijer and Koen Claessen. The Design and Implementation of Mondrian. In *Haskell Workshop.* ACM, June 1997.

[MCL98] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, pages 90–101, 1998.

[MLPR97] Pablo E. Martínez López, Pablo J. Pedemonte, and Federico C. Repond. Diagram grammars: A Structured Description of Diagrams. Technical report, LIFIA, Laboratorio de Formación e Investigación en Informática Avanzada, Universidad Nacional de La Plata, May 1997.

[MMS92] José Meseguer, Ugo Montanari, and Vladimiro Sassone. On the semantics of Petri nets. In W. R. Cleaveland, editor, *CONCUR '92: Third International Conference on Concurrency*

Theory, volume 630 of *Lecture Notes in Computer Science*, pages 286–301, Stony Brook, New York, 24–27August 1992. Springer-Verlag.

[MUT99] Hiroshi Maruyama, Naojiko Uramoto, and Kent Tamua. *XML and Java: Developing Web Applications.* Addison-Wesley, 1999.

[Nor99] Johan Nordlander. *Reactive Objects and Functional Programming.* PhD thesis, Dept. of Computing Science, Chalmers University of Technology, May 1999.

[PB99] Marian Petre and Alan F. Blackwell. Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51(1):7–30, 1999.

[Pos99] Kjell Post. Bottom-Up Evaluation of Attribute Grammars, 1999. URL: `http://www.idt.mdh.se/personal/kpt/research.htm`.

[PRML99] Pablo J. Pedemonte, Federico C. Repond, and Pablo E. Martínez López. Diagram Editor Generator: A Medium Size Haskell Application. In *3rd Latin American Conference on Functional Programing*, Recife, Brasil, March 1999.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, USA, 1 edition, 1999.

[RS96] J. Rekers and A. Schürr. A graph-based framework for the implementation of visual environments. In *Proceedings 12th of the IEEE Symposium on Visual Languages*, pages 148–155, Washington, September 3–6 1996. IEEE Computer Society Press.

[RV98] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[SSK00] Joao Saraiva, S. Doaitse Swierstra, and Matthijs F. Kuiper. Functional incremental attribute evaluation. In *Computational Complexity*, pages 279–294, 2000.

[Ste71] K. A. Steele. CPM/PERT. In *Second Canadian Man-Computer Communications Conference*, pages 81–84, May 1971.

[Str95] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley, Reading, MA, USA, second, reprinted with corrections August, 1995 edition, 1995.

[SV91] S. Doaitse Swierstra and Harald H. Vogt. Higher Order Attribute Grammars. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545, pages 256–296, New York–Heidelberg–Berlin, 1991. Springer-Verlag.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, Harlow, England, 1996.

[TVC94] Maurizio Tucci, Giuliana Vitiello, and Gennaro Costagliola. Parsing nonlinear languages. *IEEE Transactions on Software Engineering*, 20(9):720–739, September 1994.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *ACM SIGPLAN Notices*, 24(7):131–145, July 1989.

[Wit92] K. Wittenburg. Early-style parsing for relational grammars. In *Proceedings 8th of the IEEE Workshop on Visual Languages*, pages 192–199, Seattle, 1992. IEEE Computer Society Press.