Gramáticas de atributos, clasificación y aportes en técnicas de evaluación

Tesis de carrera de Magister en Ciencias de la Computación
Universidad Nacional del Sur

Bahía Blanca - Argentina

Marcelo Daniel Arroyo
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto - Argentina

e-mail: marroyo@dc.exa.unrc.edu.ar

Director: Prof. Jorge Aguirre
Supervisor: Dr. Guillermo Simari

18 de Diciembre de 2008

Agradecimientos

A mis padres, quienes lamentablemente no han podido ver el resultado final de éste trabajo. Ellos fueron quienes en definitiva forjaron mi vida profesional y personal, permitiendo que accediera a una educación superior haciendo inmensos esfuerzos, sabiendo que la educación es la herencia mas importante que me podían dejar.

A Verónica y Joaquín, quienes han sido extremadamente comprensivos por las largas horas durante muchas noches frente a la computadora, sin haber mostrado jamás una pizca de intolerancia.

Deseo expresar un especial y profundo agradecimiento a Jorge Aguirre, quien además de ser director de esta tesis, es mi mentor y amigo. Es quien permanentemente está estimulando y fomentando en forma continua la superación docente, profesional, científica y personal. Sin duda es una de las personas que, a quienes trabajamos día a día a su lado, nos enseña cosas en forma permanente, seguramente sin siquiera proponérselo.

A mis amigos mas allegados, muchos de ellos ex-alumnos, quienes no me atrevo a nombrar, porque afortunadamente son muchos y no deseo omitir a ninguno ya que saben que no puedo confiar en mi memoria volátil.

Finalmente a algunos docentes de la UNS, especialmente al Dr. Guillermo Simari, quienes han facilitado a muchos docentes y alumnos su formación de posgrado.

Resumen

Las gramáticas de atributos, desde que fueron propuestas por Knuth en 1966, se han utilizado ampliamente para el desarrollo de herramientas de procesamiento de lenguajes formales como compiladores e intérpretes de lenguajes de programación; como también para especificar la semántica de lenguajes.

Las gramáticas de atributos son un formalismo simple para la especificación de la semántica de lenguajes formales, como ser lenguajes de programación o de especificación. Integran la modularidad que brindan las gramáticas libres de contexto y la expresividad de un lenguaje funcional.

Si bien las gramáticas de atributos han sido ampliamente estudiadas no es fácil encontrar definiciones precisas y rigurosas.

Los principales motivos del desarrollo de esta tesis son en primer lugar, obtener un material autocontenido sobre sus definiciones, extensiones, implementación y aplicaciones, ya que prácticamente no existen libros actualizados en el tema.

En segundo lugar, se realiza un estudio profundo sobre nuevas clasificaciones propuestas y métodos de evaluación. En este último aspecto es donde se presentan los aportes más significativos de esta tesis.

En este trabajo se presentan las gramáticas de atributos en su forma clásica, mostrando definiciones más precisas que las que se pueden encontrar en la bibliografía tradicional. Se describen diferentes clasificaciones y métodos de evaluación secuenciales y concurrentes.

Entre los principales aportes de este trabajo, se propone un algorimo eficiente de evaluación dinámica bajo demanda, que puede aplicarse para la evaluación de cualquier gramática de atributos bien definida (no circular). Este algorimo se ha implementado en la herramienta *agcc*, la cual ha sido desarrollada en el marco de este trabajo.

Se analiza una nueva clasificación, la jerarquía *NC*, propuesta por Wuu Yang en 1999 y se relaciona con la clasificación tradicional.

Se describe *NCeval*, una herramienta desarrollada en el marco de esta tesis, la cual genera evaluadores estáticos para la familia NC(1). Un evaluador generado por NCEval realiza la evaluación disparando procesos o tareas concurrentes que no requieren sincronización, ya que operan sobre conjuntos independientes de instancias de atributos.

El método de particionado usado en NCEval está basado en las dependencias y se demuestra que la partición inducida sobre el conjunto de atributos es la mas fina posible.

Hasta el momento, no se conocen herramientas que utilicen este enfoque.

Finalmente se describe *agcc* (Attribute Grammars Compiler Compiler), herramienta desarrollada en el marco de esta tesis, la cual tiene un diseño totalmente modular permitiendo su extensibilidad tanto en los generadores de código como en los métodos de evaluación utilizados. Acepta la familia más amplia de gramáticas de atributos para las cuales pueden generarse evaluadores estáticos.

Índice general

Capítulo 1

Introducción

Desde que D. Knuth introdujo en 1966 las gramáticas de atributos (GA)[26], estas se han utilizado ampliamente para el desarrollo de herramientas de procesamiento de lenguajes formales como compiladores, intérpretes, traductores como también para especificar la semántica de lenguajes de programación. Las gramáticas de atributos son un formalismo simple para la especificación de la semántica de lenguajes formales, como los lenguajes de programación o de especificación. Integran la modularidad que brindan las gramáticas libres de contexto y la expresividad de un lenguaje funcional.

1.1. Gramáticas de atributos

En una gramática de atributos, se relaciona con cada símbolo de una gramática libre de contexto un conjunto de *atributos*. Cada regla o producción tiene asociados un conjunto de *reglas semánticas* que toman la forma de asignación a atributos de valores denotados por la aplicación de una función, la cual puede tomar como argumentos intancias de atributos pertenecientes a los símbolos que aparecen en la producción.

Las reglas semánticas inducen *dependencias* entre los atributos que ocurren en la producción. El orden de evaluación es implícito (si existe) y queda determinado por las dependencias entre las instancias de los atributos.

Una regla semántica se podrá evaluar cuando las instancias de los atributos que aparecen como sus argumentos estén evaluadas.

Un evaluador de gramáticas de atributos debe tener en cuenta las dependencias entre las instancias de atributos para seguir un orden consistente de evaluación de los mismos.

Si una GA contiene dependencias circulares no podría ser evaluada[1] ya que no podría encontrarse un orden de evaluación.

Existen numerosas herramientas basadas en este formalismo o en alguna de sus extensiones, entre las cuales podemos mencionar *yacc, Yet Another Compiler-Compiler*[29], desarrollado por AT&T, *AntLR*[53], *JavaCC*[54], *JavaCUP*[55], *ELI*[24] y muchas otras. Algunas de ellas se describen en el capítulo 8.

[1] Podría ser evaluada si existe un punto fijo sobre la relación de dependencia utilizando evaluación *normal o lazy*.

Posteriormente de su introducción, las investigaciones y los desarrollos de herramientas basadas en gramáticas de atributos se han concentrado en la búsqueda de métodos de evaluación eficientes.

Los métodos utilizados pueden clasificarse en *estáticos* y *dinámicos*. Los primeros generan la secuencia de pasos de evaluación, consistente con las dependencias entre las instancias de los atributos, en tiempo de construcción del evaluador, es decir que los órdenes de evaluación son determinados realizando un análisis estático de la gramática.

Los métodos dinámicos determinan la secuencia de pasos u orden de evaluación en base a las dependencias de cada instancia del árbol sintáctico construido por el parser.

1.2. Métodos de evaluación

Los métodos estáticos deben tener en cuenta todos los posibles árboles sintácticos posibles a ser generados por la gramática y calcular todas las posibles dependencias entre las instancias de los atributos. Además, se deberán detectar las posibles dependencias circulares, para informar la viabilidad de su evaluación.

Esto se conoce como el *problema de la circularidad*, el cual se ha demostrado ser intrínsecamente exponencial [20].

El problema de la circularidad ha motivado que muchos investigadores hayan realizado esfuerzos en la búsqueda e identificación de familias o subgrupos de gramáticas de atributos, para las cuales puedan detectarse circularidades con algoritmos de menor complejidad (polinomial o lineal).

Estas familias imponen restricciones sobre la gramática de atributos o sobre las dependencias entre sus atributos para garantizar que una GA no sea circular, con el costo de restringir su poder expresivo.

Las clases de familias de gramáticas de atributos que se han utilizado para el desarrollo de herramientas eficientes y que se encuentran ampliamente analizadas en la bibliografía especializada, encontramos las s-atribuidas[2], l-atribuidas, las gramáticas de atributos ordenadas (OAG) y las absolutamente no circulares (ANCAG)[2].

En 1980, Uwe Kastens[23] caracterizó las gramáticas de atributos ordenadas y propuso un método para su evaluación, denominado *secuencias de visita*. Estas son secuencias de operaciones que conducen el recorrido del árbol sintáctico atribuido y realizan la evaluación de las instancias de los atributos.

Kastens propone un método para generar las secuencias de visita en tiempo polinomial para la familia OAG.

Mas recientemente, en 1999, se han propuesto nuevas familias de GA para las que se pueden implementar evaluadores eficientes basado en métodos estáticos y con un mayor poder expresivo que las utilizadas tradicionalmente[44].

1.3. Aportes en esta tesis

Se presentan las gramáticas de atributos en su forma clásica, mostrando definiciones más precisas que las que se pueden encontrar en la bibliografía tradicional. Se

[2]Familia soportada por la popular herramienta yacc.

describen diferentes clasificaciones y modelos de evaluación secuenciales y concurrentes.

Se analiza una nueva clasificación: las jeraquía *NC* [44], propuesta por Wuu Yang [42] y se compara con la clasificación tradicional.

Se describen diferentes métodos de evaluación y se propone un algoritmo de evaluación dinámico bajo demanda mas eficiente que los encontrados en la bibliografía y en las herramientas tradicionales[2]. El algoritmo puede ser aplicado a la familia de GA bien definidas, la cual es la familia mas amplia de las GA no circulares.

Se muestra cómo podría ser extendido para algunas extensiones de GA como por ejemplo, las GA condicionales y las GA de alto orden (HAG).

Otro de los principales aportes de esta tesis es el desarrollo de un generador de evaluadores concurrentes para la familia NC(1), denominado *NCEval*, el cual produce evaluadores cuyos procesos de evaluación operan sobre conjuntos de atributos en regiones independientes, por lo que no requieren de ningún tipo de sincronización.

Los procesos evaluadores están basados en la idea de secuencias de visita de Kastens, lo que muestra que el método, en realidad, puede aplicarse para cualquier familia de GA no circulares.

El generador utiliza un algoritmo de generación de una partición de las instancias de los atributos para una GA dada y se demuestra que es la partición mas fina posible[45].

Hasta el momento no se conoce ninguna herramienta para la familia citada ni utilizando este método de evaluación.

1.4. Desarrollo

En el capítulo 2 se presenta la introducción a las GA en su forma clásica con definiciones más precisas que las encontradas en la bibliografía tradicional.

En el capítulo 3 se muestran dos clasificaciones de GA: ls clasificación tradicional de subfamilias de GA no circulares útiles para la implementación de evaluadores estáticos eficientes y la *Jerarquía NC*[44].

Se realiza además una clasificación basada en la estrategia de evaluación.

En el capítulo 4 se presentan diferentes métodos de evaluación, y su aplicabilidad a las las familias caracterizadas anteriormente.

Se presenta el algoritmo de evaluación bajo demanda desarrollado en esta tesis y se realiza un estudio de su performance.

Algunas técnicas de evaluación concurrente se describen en el capítulo 5.

En el capítulo 6 se describe **NCeval**, una herramienta de generación de evaluadores concurrentes de gramáticas de atributos basado en secuencias de visita extendido para la familia $NC(1)$, el cual utiliza varios conceptos y algoritmos analizados en los capítulos previos.

Se muestra que es posible generar evaluadores para la familia de gramáticas más amplia existente con una performance comparable a los evaluadores basados en secuencias de visita para subclases de gramáticas de atributos (como las ANCAG o OAG).

En el capítulo 7 se describen varias extensiones al formalismo clásico de gramáticas de atributos, como son las GA condicionales, GA de alto orden (HAG), y sus técnicas de evaluación.

Además se muestra cómo es posible extender el algoritmo de evaluación bajo demanda propuesto en esta tesis para cada una de las extensiones a las GAs.

En el capítulo 8 se hace una breve descripción del estado del arte actual con respecto a algunas de las principales herramientas existentes basadas en gramáticas de atributos. En este capítulo se describe *agcc* una herramienta desarrollada en el marco de esta tesis. *AGCC* soporta la familia mas grande de gramáticas de atributos no circulares y utiliza técnicas de evaluación desarrolladas en esta tesis.

En el capítulo 9 se presentan las conclusiones y trabajo futuro.

Capítulo 2

Definiciones

En este capítulo se definen formalmente las gramática de atributos y se introducen un conjunto de conceptos relacionados útiles para la caracterización de sus propiedades.

A modo de ejemplo introductorio de su utilización, en la figura 2.1 se muestra una especificación de una GA que computa el tipo y la memoria requerida por una declaración al estilo del lenguaje *C* y además computa el tipo y el desplazamiento para cada identificador.

---

nonterminals: decl, type, lid;
terminals: id, int, float;

semantic domains
TYPE type_id = {t_int, t_real};
attributes

SYNT= decl.mem, lid.mem:**int**; type.t:type_id;
INH= lid.t, id.t:type_id; id.off:**int**;
rules

p_0: *decl* → *type lid ';'*
attribution
lid.t=type.t
decl.mem=lid.mem
end

p_1: *type* → *int*
attribution
type.t=t_int
end

p_2: *type* → *float*
attribution
type.t=t_real
end

p_3: *lid* → *id*
attribution
id.t=lid.t
id.off=4
lid.mem=id.off
end

p_4: $lid_0 \rightarrow lid_1, id$
attribution
$id.t = lid_0.t$
$id.off = lid_1.mem$
$lid_1.t = lid_0.t$
$lid_0.mem = lid_1.mem$ + *size*($lid_0.t$)
end

Figura 2.1: Una GA que computa la memoria requerida por una declaración.

La sentencia **TYPE** permite definir tipos (o dominio de valores que puede tomar algún atributo) al programador. Las sentencias **SYNT** y **INH** declaran los atributos sintetizados y heredados asociados a cada símbolo de la gramática, respectivamente[1].

[1] En la sección 2.1 se definen formalmente estos conceptos.

Los atributos se clasifican en *heredados* y *sintetizados*. Intuitivamente, los atributos de un símbolo representan valores funcionales asociados a cada una de sus ocurrencias en el árbol sintáctico.

Los atributos heredados de un símbolo son aquellos en que sus valores se definen en una producción en la cual el símbolo aparece en la parte derecha, mientras que los sintetizados de un símbolo son los atributos que se definen en una producción en la que el símbolo aparece en la parte izquierda. Un atributo puede ser heredado o sintetizado, pero no de ambos tipos.

La figura 2.2 muestra un árbol atribuido para una declaración generada por la GA de la figura 2.1, en el cual se pueden apreciar las instancias de los atributos correspondientes. Las flechas gruesas representan el árbol de derivación y las flechas finas las dependencias entre las instancias de los atributos.

Si se analiza lo anterior en un árbol atribuido, (el cual es un árbol sintáctico cuyos nodos contienen las instancias de los atributos del símbolo correspondiente), se podría decir informalmente, que los valores de los atributos heredados se transmiten en el árbol en forma descendente (y posiblemente de izquierda a derecha en la misma producción) y los sintetizados en forma ascendente.

En la producción p_0 se pasa el valor del atributo sintetizado *type.t* (definido en las producciones p_1 o p_2) en el atributo heredado t del símbolo *lid* (denotado como *lid.t*), el cual será utilizado para asignar el tipo a cada identificador declarado (en el atributo *id.t*).

El atributo sintetizado *decl.mem* representa la cantidad total de memoria requerida por la representación de las variables definidas en la declaración.

En la producción p_3 se definen el tipo del identificador ($id.t$) y su desplazamiento ($id.off$) (dentro del registro de activación correspondiente).

El atributo sintetizado *lid.mem* representa la cantidad de memoria utilizada por los identificadores de su subárbol, es decir, se utiliza como un acumulador para finalmente definir el atributo *decl.mem*.

La función *size()* computa el tamaño de una variable dependiendo de su tipo.

2.1. Gramáticas y lenguajes

No existe una única definición de gramática de atributos, así que en esta sección se presenta una definición formal que no contiene las simplificaciones, omisiones y ambigüedades encontradas en la mayor parte de la bibliografía tradicional.

Informalmente, una gramática de atributos se basa en una gramática libre de contexto (CFG) en el cual cada símbolo de la gramática tiene asociado un conjunto de atributos y en la cual cada producción tiene asociada un conjunto de *reglas de evaluación*.

Cada atributo puede tomar valores sobre algún dominio y éstos valores se van computando sobre el árbol sintáctico atribuido[2].

[2]Un *árbol sintáctico atribuido* es un árbol sintáctico en el cual cada nodo contiene instancias de los atributos de la instancia del símbolo correspondiente al nodo. Su definición formal está en la sección 2.2.1.

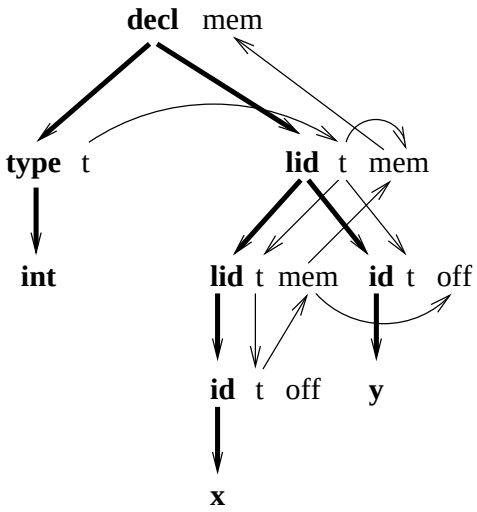

Figura 2.2: Árbol atribuido para la cadena ***int x,y;***.

2.1.1. Mecanismos de descripción de lenguajes

Existen varios formalismos para describir lenguajes. Entre los más utilizados se pueden clasificar en dos grandes grupos: formalismos de *generación* de lenguajes como las gramáticas, BNF's[3] y las expresiones regulares; por otra parte encontramos los formalismos de *aceptación* de lenguajes como los autómatas.

Tanto los autómatas como las gramáticas se han clasificado en base a ciertas restricciones, las cuales definen diferentes clases de lenguajes. La tabla 2.1 relaciona los formalismos con cada familia de lenguajes.

Los lenguajes libres de contexto se han utilizado para describir lenguajes de programación y se han desarrollado métodos de análisis sintáctico (parsing) determinísticos para ciertas subclases, como las gramáticas de precedencia, las *LL(k)*[4] y las *LR(k)* [5][6].

La mayoría de los lenguajes de programación contienen construcciones sintácticas dependientes del contexto, como por ejemplo, en un lenguaje con reglas de alcance estático, un identificador debe estar dentro del alcance de su declaración. Aunque se podrían utilizar formalismos dependientes de contexto para describir estas construcciones, la descripción de lenguajes por medio de éstas gramáticas se torna dificultosa y poco clara. Además, los métodos de análisis sintáctico para este tipo de lenguajes no se han desarrollado tanto debido a complejidades teóricas y prácticas. Las herramientas existentes producen analizadores menos eficientes que los métodos utilizados para los lenguajes independientes del contexto.

Las gramáticas de atributos permiten describir estas características extendiendo así el poder expresivo de las CFG ya que permiten, además, describir traductores

[3] Backus Naur Form.
[4] Análisis *Left to right* de la entrada que realiza *derivaciones de más a la izquierda.*
[5] Análisis *Left to right* que realiza *derivaciones de más a la derecha.*

Gramática	Autómata	Lenguaje
Regular	Finito	Regular
Libre de contexto	Pila no determinístico	Libre de contexto
L.C. determinístico	Aut. pila determinístico	L.C.determinístico
Dependiente de contexto	Maq. Turing lineal acotada	Dep. de contexto
Irrestricta	Máquina de Turing	Recursivamente numerable

Cuadro 2.1: Relación entre formalismos y lenguajes

dirigidos por sintaxis mediante la computación de las ecuaciones semánticas asociadas a las producciones.

2.1.2. Gramáticas libres de contexto

Las gramáticas libre de contexto se han utilizado ampliamente para describir lenguajes de programación, de especificación y aún para construcciones de lenguaje natural. Informalmente, una gramática describe el lenguaje por medio de *reglas o producciones* que al ser aplicadas *generan* las sentencias o "frases" del lenguaje.

Definición 2.1.1 *Una gramática libre de contexto (CFG) es una tupla* (V_N, V_T, S, P)*, donde* V_N *es el conjunto finito de símbolos no terminales,* V_T *es el conjunto finito de símbolos terminales,* $S \in V_N$ *es el símbolo de comienzo y* P *es un conjunto finito de producciones.*

Los conjuntos V_N *y* V_T *deben ser disjuntos (*$(V_N \cap V_T) = \emptyset$*) y denotaremos* $\Sigma = V_N \cup V_T$*.*

P *es un* conjunto de producciones*, donde una producción* $p \in P$*,* $P \subseteq V_N \times (V_n \cup V_T)^*$*. Así una producción* $p \in P$ *tiene la forma* (L, R)*, donde* $L \in V_N$ *es la parte izquierda (lhs) de la producción y* $R \in (V_N \cup V_T)^*$ *es la parte derecha (rhs).*

Por claridad, en lugar de describir las producciones como pares, a una producción rotulada p: (X_0, α), con $\alpha = X_1 \dots X_n$ se la denotará como:

$$p : X_0 \rightarrow X_1 \dots X_{np} \qquad (si\ \alpha \neq \epsilon) \tag{2.1}$$

Cuando la cadena α sea vacía (ϵ), se escribirá como:

$$p : X_0 \rightarrow \lambda \tag{2.2}$$

Definición 2.1.2 *Dada una producción* p *de la forma (2.1), cada* X_i*, con (*$0 \leq i \leq n_p$*), denota una ocurrencia en ella de un símbolo de* V_N *si i=0, y de* $V_N \cup V_T$ *para i>0.*

Si un símbolo X *ocurre* más de una vez en una producción p, se denotará cada ocurrencia como X_i con i variando desde cero de izquierda a derecha. Con esta notación, una ocurrencia de un símbolo se podrá identificar sin ambigüedad.

Definición 2.1.3 *Sea* p *una producción como en (2.1),* X^p *representa los símbolos que ocurren en la producción* p*:*

$$X^p = \bigcup_{i=0}^{n_p} \{X_i^p\}$$

donde X_i^p, $(1 \leq i \leq n_p)$, *son los símbolos que ocurren el la producción* p.

De aquí en adelante se asumirá que el símbolo de comienzo S aparece en la parte izquierda de una única producción y no puede aparecer en la parte derecha de ninguna producción[6].

Definición 2.1.4 *Sean* $\alpha, \beta \in (V_N \cup V_T)^*$ *y sea* $q : X \to \varphi$ *una producción de P, entonces* $\alpha X \beta \underset{G}{\overset{q}{\Longrightarrow}} \alpha\varphi\beta$

La relación $\underset{G}{\overset{q}{\Longrightarrow}}$ se denomina *relación de derivación* y se dice que la cadena $\alpha X \beta$ deriva directamente (por aplicación de la producción p) a $\alpha\varphi\beta$.

Cuando se omita la producción aplicada en un paso de derivación, se denotará como $\underset{G}{\Longrightarrow} q$.

Se denotará como $\underset{G}{\overset{*}{\Longrightarrow}}$ a la clausura reflexo-transitiva de la relación de derivación.

Definición 2.1.5 *Sea* $G = (V_N, V_T, S, P)$ *una gramática libre de contexto. Una cadena* α *obtenida por* $S \underset{G}{\overset{*}{\Longrightarrow}} \alpha$ *que contiene sólo símbolos terminales* ($\alpha \in {V_T}^*$), *se denomina una sentencia de G. Si la cadena* $\alpha \in (V_T \cup V_N)^*$ *se denomina forma sentencial.*

Definición 2.1.6 *El lenguaje generado por G, denotado como* $L(G)$, *satisface*

$$L(G) = \{w | w \in {V_T}^* \mid S \underset{G}{\overset{*}{\Longrightarrow}} w\}$$

Definición 2.1.7 *Sea el grafo dirigido* $ST = (K, D)$ *un árbol (K es un conjunto de nodos y D es una relación no reflexiva sobre K), con* k_0 *como raíz.*

Sea l *una función de rotulación de la forma* $l : K \to \sigma \cup \{\epsilon\}$ *y sean* $k_1, \ldots, k_n$, $(n > 0)$, *los sucesores inmediatos de* k_0.

El árbol $ST = (K, D)$ *es un árbol de derivación (o parse tree) correspondiente a* $G =< V_N, V_T, P, S >$ *si cumple con las siguientes propiedades:*

1. $K \subseteq (V_N \cup V_T \cup \epsilon)$
2. $l(k_0) = S$
3. $S \to l(k_1) \ldots l(k_n)$
4. *Si* $l(k_i) \in V_T \cup \{\epsilon\}$, $(1 \leq i \leq n)$, *entonces* k_i *es una hoja de ST.*
5. *Si* $l(k_i) \in V_N$, $(1 \leq i \leq n)$, *entonces* k_i *es la raíz del árbol sintáctico para la gramática libre de contexto* $< V_N, V_T, P, l(k_i) >$.

Definición 2.1.8 *Sea ST(G) un árbol de derivación para* $G =< V_N, V_T, S, P >$. *La frontera de ST(G) es la cadena* $l(k_1) \ldots l(k_n)$ *tal que* $k_1 \ldots k_n$ *es la secuencia formada por las hojas de ST(G) visitadas en un recorrido preorden.*

[6] Esta forma se denomina *forma normal.*

Teorema 2.1.1 *Sea* $G =< V_N, V_T, S, P >$ *una gramática libre de contexto,* $S \overset{*}{\underset{G}{\Longrightarrow}} \alpha$ *si y sólo si existe un árbol de derivación para* G *cuya frontera es* α*.*

Demostración: Probaremos que para cualquier no terminal A en V_N, $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha$ si y sólo si existe un subárbol con raíz A y frontera α. Haremos inducción en el número de nodos interiores del árbol cuya raíz es A y cuya frontera es α.

Si existe un único nodo interior, éste es A y por la definición de árbol de derivación, debe existir una producción en P de la forma $A \to \alpha$ y por lo tanto $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha$.

Ahora supongamos que el resultado es verdadero para un árbol con raíz A con *k-1 nodos* interiores. También suponemos que $\alpha = \alpha_1\alpha_2...\alpha_n$ es la frontera de un árbol con k nodos interiores. Consideremos los hijos de la raíz, $X_1X_2...X_n$, (en orden de izquierda a derecha). No todos ellos pueden ser hojas (por tener k-1 nodos interiores). Por la definición 2.1.7, $\exists p$ tal que $p : l(A) \to l(X_1)l(X_2)\dots l(X_n)$.

Si un nodo $X_i, (1 \leq i \leq n)$ no es una hoja, entonces $l(X_i) \in V_N$ y por hipótesis inductiva, $X_i \overset{*}{\underset{G}{\Longrightarrow}} \alpha_i$.

Si el nodo $l(X_i) \in V_T$, claramente, $l(X_i) \overset{*}{\underset{G}{\Longrightarrow}} \alpha_i$ (ya que $l(X_i) = \alpha_i$, o sea es una derivación trivial).

Si ordenamos las derivaciones, se puede ver claramente que

$$l(A) \overset{p}{\underset{G}{\Longrightarrow}} l(X_1)l(X_2)\dots l(X_n) \overset{*}{\underset{G}{\Longrightarrow}} \alpha_1 l(X_2)\dots l(X_n) \overset{*}{\underset{G}{\Longrightarrow}} \alpha_1\alpha_2\dots l(X_n) \overset{*}{\underset{G}{\Longrightarrow}} \alpha_1\alpha_2\dots\alpha_n = \alpha$$

Ahora supongamos que la derivación $X_0 \to \alpha_1 \dots \alpha_n$ toma k pasos. Sea la primer derivación $p_i : X_0 \to X_1 \dots X_n$, entonces $ST(G)/p_i$ es $l(A) \to l(X_1)l(X_2)\dots l(X_n)$.

Por hipótesis inductiva, por cada $l(X_i) \in V_N$ existe un árbol de derivación ST_i que tiene como raíz a X_i y como frontera a α_i. Por lo tanto existe un árbol de derivación con X_0 como raíz, $X_1, X_2, \dots, X_n$ como hijos de la raíz y que a su vez son raíces de $ST_i, 1 \leq i \leq n$, respectivamente.

Formalmente: $ST(G) =< \{A, \dots, X_n\} \cup \phi_1(ST_i), \{(A, X_1), \dots, (A, X_n)\} \cup \phi_2(ST_i) >$

donde $\phi_1(ST_i)$ y $\phi_2(ST_i)$ $(1 \leq i \leq n)$, denotan el conjunto de vértices y arcos de ST_i, respectivamente. □

Definición 2.1.9 *Se dice que una gramática libre de contexto* $G =< V_N, V_T, S, P >$ *es* ***reducida*** *si*

$\forall B \in V_N$, $\exists \alpha, \gamma \in \Sigma^* : S \overset{*}{\underset{G}{\Longrightarrow}} \alpha B \gamma$ *y* $\exists \tau \in V_T^* : B \overset{*}{\underset{G}{\Longrightarrow}} \tau$*.*

Definición 2.1.10 *Una gramática libre de contexto* $G =< V_N, V_T, S, P >$ *es* ***ambigua*** *si para una cadena* α*, derivada de* G*, existen dos o más árboles de derivación diferentes.*

2.1.3. Gramáticas de atributos

Definición 2.1.11 *Una gramática de atributos es una tupla* $GA = (G, A, V, Dom, F, R)$ *donde :*

- $G = (V_N, V_T, S, P)$ *es una gramática libre de contexto reducida y no ambigua*
- $A = \cup_{X \in (V_N \cup T)} A(X)$*, es el conjunto finito de atributos (*$A(X)$ *es el conjunto de atributos asociados al símbolo* X*)*

- *V es el conjunto finito de dominios de valores de los atributos*[7]
- *$Dom : A \to V$ asocia a cada atributo un dominio o conjunto de valores $d \in V$.*
- *F es un conjunto finito de funciones semánticas (con perfiles que se definen en el próximo punto).*
- *$R = \bigcup_{p \in P} R^p$ es el conjunto finito de reglas de atribución o ecuaciones asociadas a cada producción $p \in P$, donde*

$$R^p = \bigcup_{j=0}^{m^p} \{r_j^p\} \qquad (\#(R^p) = m^p \geq 0)$$

y cada regla $r_j^p \in R^p$, con $0 \leq j \leq m^p$ es una tupla de la forma

$$r_j^p = (X_0.a_0, f, X_1.a_1, \ldots, X_k.a_k) \tag{2.3}$$

con $X_i \in X^p$, $a_i \in A(X_i)$, $(0 \leq i \leq k)$ y $f \in F$.

Aclaración: los subíndices utilizados en los símbolos denotan ocurrencias de símbolos en la regla de atribución, es decir que la ocurrencia de X_i en la regla r_j^p no necesariamente se corresponde con el símbolo X_i^p.

Este abuso de notación simplifica el uso de super y subíndices y permite definiciones mas claras teniendo en cuenta el contexto correspondiente.

La función $f \in F$ debe ser de la forma:

$$f \subseteq (\bigotimes_{j=0}^{k} Dom(a_j)) \to Dom(a_0)$$

Nota: Las definiciones presentadas en la bibliografía básica (como por ejemplo en [26] , [2] y [6]), no imponen las restricciones que aquí se han agregado. Las restricciones utilizadas aquí permiten definir que el significado de una cadena sea único y la simplificación de enunciados de otras propiedades.

Se utilizará la notación $X.a$ para significar que el atributo a está asociado al símbolo X ($a \in A(X)$) y para denotar el valor de una ocurrencia del attributo a del símbolo X en una regla de atribución.

Por mayor claridad sintáctica escribiremos una regla de la forma 2.3 como:

$$r_j^p : X_0.a_0 = f(X_1.a_1, \ldots, X_k.a_k) \tag{2.4}$$

Definición 2.1.12 *En una regla r_j^p como en 2.4* ***ocurren*** *los atributos $X_i.a_i \in A$, $(0 \leq i \leq k)$ y* ***ocurre*** *la función semántica f.*

[7] En este caso se han tomado dominios de los atributos para presentar una definición de gramática de atributos pura, en las cuales se dispone sólo de las funciones semánticas de F. Se podría definir V como un conjunto finito de elementos de un sistema de tipos dado.

Definición 2.1.13 *Un atributo a está asociado al símbolo X si y sólo si* $a \in A(X)$.

Asumiremos que los conjuntos de atributos de cada símbolo son disjuntos, esto es

$$A(X_i) = A(X_j) \Rightarrow i = j \text{ y } i \neq j \Rightarrow A(X_i) \cap A(X_j) = \emptyset$$

Definición 2.1.14 *Los atributos asociados a una producción* $p \in P$, *donde p es de la forma* $X_0 \rightarrow X_1 X_2 \dots X_{np}$*:*

$$A(p) = \bigcup_{i=0}^{np} A(X_i)$$

Definición 2.1.15 *Sea una regla* $r \in R^p$ *como la dada en (2.4), la ocurrencia definida por la regla r u ocurrencia de salida es*

$$DO(r) = X_0.a_0$$

Las ocurrencias de uso en la regla r, también conocidas como ocurrencias de entrada u ocurrencias aplicadas.

$$UO(r) = \{X_i.a_i \mid 1 \leq i \leq k\}$$

Definición 2.1.16 *Las ocurrencias de atributos definidos en la producción p,*

$$DO(p) = \bigcup_{r \in R^p} \{DO(r)\}$$

Definición 2.1.17 *Las ocurrencias de uso de atributos en la producción p,*

$$UO(p) = \bigcup_{r \in R^p} UO(r)$$

Ahora podemos redefinir las *ocurrencias de los atributos* en una regla y en una producción.

Definición 2.1.18 *Los atributos que ocurren en la regla semántica* $r \in R^p$, *de una producción p es el conjunto:*

$$A(r) = DO(r) \bigcup UO(r)$$

Definición 2.1.19 *Los atributos que ocurren en la producción p es el conjunto*

$$A(p) = \bigcup_{r \in R^p} A(r)$$

El conjunto de atributos de un símbolo $X \in \Sigma$ está particionado en dos subconjuntos disjuntos: $S(X)$ y $I(X)$. El conjunto $S(X)$ representa los *atributos sintetizados del símbolo X* y $I(X)$ los *atributos heredados del símbolo X*.

Se asumirá que el símbolo de comienzo S no tiene atributos heredados.

Definición 2.1.20 *Un atributo* $X.a \in S(X)$ *si existe una producción* $p : X \rightarrow \chi$ *y* $X.a \in DO(p)$ *(*$\chi \in \Sigma^*$*).*

Definición 2.1.21 *Un atributo $a \in I(X)$ si existe una producción $p : Y \rightarrow \mu X \nu$ y $X.a \in DO(p)$ $(\mu, \nu \in \Sigma^*$ y $X \in (\Sigma - \{S\}))$.*

Para evitar circularidades obvias, se impone que la restricción que
$\forall X \in \Sigma : S(X) \cap I(X) = \emptyset$

Si un atributo podría ser sintetizado y heredado simultáneamente, podría definirse, por ejemplo, en una producción de la forma $X \rightarrow \alpha X \beta$, una regla semántica con la forma $X_0.a = f(\ldots, X_0.a, \ldots)$. En este caso la gramática de atributos será circular (como ya se verá en la sección 2.2), ya que un atributo dependerá (tal vez transitivamente) de sí mismo.

Con esta restricción, las definiciones anteriores inducen una partición en el conjunto A:

$$S(A) = \bigcup_{X \in \Sigma} S(X)$$

y

$$I(A) = \bigcup_{X \in (\Sigma - \{S\})} I(X)$$

a los que denominamos los *atributos sintetizados de* y los *atributos heredados de* de la GA, respectivamente.

Esta definición no se corresponde la definición original de Knuth (ver [26] quien establecía que los símbolos terminales sólo podían contener atributos heredados. Nuestra definición elimina esa restricción y se debe a conveniencias prácticas ya que en el desarrollo de sistemas de procesamiento de lenguajes basados en traducción dirigida por la sintaxis es preferible modularizar el problema en cuanto al reconocimiento de símbolos terminales [8] en el analizador lexicográfico, el cual generalmente reconoce un lenguaje regular, y a menudo es conveniente definir atributos (sintetizados) para los tokens los cuales se computan durante el análisis lexicográfico.

2.2. Gramáticas de atributos bien definidas

Definición 2.2.1 *Una gramática de atributos es completa si cumple con las siguientes propiedades:*

1. *Para toda producción $p \in P$, de la forma $p : X \rightarrow \chi$, todos los atributos sintetizados de X se definen en p.*
2. *Para toda producción $p \in P$, de la forma $p : Y \rightarrow \mu X \nu$, todos los atributos heredados de X se definen en p.*

Definición 2.2.2 *En una regla $r_j^p : X_i.a = f(\ldots, X_j.b, \ldots) \in R^p$, el atributo $\boldsymbol{a}$ depende (directamente) del atributo $\boldsymbol{b}$ y se denotará como $X_j.b \rightarrow X_i.a$.*

Definición 2.2.3 *La relación de dependencia directa entre los atributos de la producción p:*

$$DP(p) = \{(X_i.a, X_j.b) \mid X_j.b \rightarrow X_i.a \in R^p\}$$

Definición 2.2.4 *El grafo de dependencias de una producción p:*

$$DG(p) =< A(p), DP(p) >$$

De aquí en adelante se utilizarán indistintamente $DP(p)$ o $DG(p)$ cuando se refiera al conjunto o grafo de dependencias directas de la producción p.

[8] También denominados *tokens*.

2.2.1. Árbol sintáctico atribuido

Definición 2.2.5 *Un árbol atribuido T(GA)*[9] *para la gramática de atributos GA es un árbol de derivación (o árbol sintáctico) ST(G) cuyos nodos son tuplas de la forma* $(l(X), (l(X).a_1, l(X).a_2, \ldots, l(X).a_k))$*, donde* $l(X)$ *es el rótulo del nodo y* $\{l(X).a_1, l(X).a_2, \ldots, l(X).a_k\} = A(X)$.

Diremos que en un nodo $n = (l(X), (l(X).a_1, l(X).a_2, \ldots, l(X).a_k))$ *cada* $l(X).a_i$, $(1 \leq i \leq k)$*, es una instancia del atributo* $X.a_i$*, respectivamente.*

Definición 2.2.6 *Para un árbol sintáctico atribuido* $T(GA)$*, su grafo de dependencias* $GD(T)$ *es el grafo construido a partir de la composición de los grafos de dependencias* $GD(p)$ *de cada producción aplicada durante la construcción de* T*. Formalmente,*

$$GD(T) = (V_{GD}(T), E_{GD}(T))$$

donde:

1. $V_{GD}(T) = \{X.a \mid \exists n$*, un nodo de T(GA) con rótulo* $l(X)$ *y* $a \in A(X)\}$.
2. *la relación* $E_{GD}(T)$ *(arcos de* $GD(T)$*) cumple con la siguiente propiedad:*

   Sea n *un nodo de* $T(GA)$ *con rótulo* $l(X_0)$*, cuyos hijos son los nodos* $n_1, n_2, \ldots, n_k$*, con* $k \geq 0$ *con rótulos* $l(X_1)$*,* $l(X_2)$*, ...,* $l(X_k)$*, respectivamente, y sea* $p : X_0 \rightarrow X_1 X_2 \ldots X_k$ *una producción de GA, entonces,*

$$(X_i.a, X_j.b) \in E_{GD}(T) \Leftrightarrow (X_i.a, X_j.b) \in DP(p) \qquad 0 \leq i,\ j \leq k$$

Definición 2.2.7 *Una gramática de atributos GA es circular si y solo si existe un árbol sintáctico atribuido* $T(GA)$*, tal que su grafo de dependencias* $GD(T)$ *contiene al menos un ciclo.*

El algoritmo 1 describe el conocido *test de circularidad* propuesto por Knuth en [26].

Definición 2.2.8 *Un orden de evaluación consistente, con respecto a las dependencias entre los atributos de una gramática de atributos GA, es una secuencia (orden parcial) de instancias de atributos con la siguiente restricción:*

Dada una regla $r_j^p : X_0.a_0 = f(\ldots, X_i.a_i, \ldots)$ *en una producción p,* $X_i.a_i$ *deberá preceder a* $X_0.a_0$.

Definición 2.2.9 *Un orden topológico de un grafo dirigido acíclico* $G(V, E)$ *es una secuencia* S *de los vértices o nodos de* G *tal que:*

1. *Los vértices aparecen una única vez en* S.
2. *Por cada par de vértices diferentes,* v_i *y* v_j*, en* S*, si* $(v_i, v_j) \in G$*, entonces,* $v_i <_S v_j$ *(*v_i *precede a* v_j *en la secuencia* S*).*

Es fácil notar que la secuencia S define un *orden total* de los nodos o vértices de G.

[9] También conocido como *árbol de derivación atribuido.*

```
Δ(X) ← ∅, ∀X ∈ V
repeat
  change ← false
  for each production q: X_0 → X_1 X_2 ... X_k do
    G ← DP(q)
    for j = 1 to k do
      for each δ' ∈ Δ(X_j) do
        G ← G ∪ δ'
      end for
    end for
    δ ← G||_{X_0}
    if δ es circular then error
    if δ ∉ Δ(X_0) then
      changed ← true
      Δ(X_0) ← Δ(X_0) ∪ δ
    end if
  end for
until not changed
```

Algoritmo 1: Algoritmo de test de circularidad

Teorema 2.2.1 *Cualquier orden topológico S de $DG(T)$ de una gramática de atributos bien definida GA es un orden de evaluación consistente con respecto a las dependencias entre los atributos de GA.*

Demostración: Se procederá por inducción sobre la longitud del camino definido por la relación entre las dependencias de las instancias de los atributos en $GD(T)$.

Caso base: (dependecias directas) dada una regla de atribución de la forma $r_j^p : X_0.a_0 = f(X_1.a_1, \ldots, X_k.a_k)$, por la definición 2.2.3, se generan los siguientes arcos en $DG(T)$: $(X_1.a_1, X_0.a_0)$, $\ldots$, $(X_k.a_k, X_0.a_0)$ lo cual, por definición de orden topológico, $X_1.a_1 <_S X_0.a_0$, $\ldots$, $X_k.a_k <_S X_0.a_0$, por lo que los atributos $X_1.a_1$,$\ldots$,$X_k.a_k$ se evaluarán antes que $X_0.a_0$

Paso inductivo: Dada una dependencia indirecta (camino de longitud $n > 1$ entre las ocurrencias de los atributos $X.a$ y $Y.b$). Obviamente deberá existir un arco $(Z.c, Y.b) \in E_{GD(T)}$ y $Z.c$ es alcanzable de $X.a$ por un camino de longitud $n - 1$.

Por hipótesis inductiva, $X.a <_S Z.c$, y por definición de orden topológico, $Z.c <_S Y.b$, por lo que se respeta la definición de orden de evaluación consistente. □

Definición 2.2.10 *La evaluación de atributos sobre un árbol atribuido $T(GA)$ es un proceso que computa los valores de las instancias de atributos de $T(GA)$ de acuerdo a las reglas semánticas R – es decir que el valor de una instancia del atributo $X_0.a$ en un nodo rotulado X_0, se obtiene computando la regla semántica $r_j^p : X_0.a_0 = f(X_1.a_1, \ldots, X_k.a_k)$ – según un orden de evaluación consistente.*

Definición 2.2.11 *El árbol atribuido, sobre el cual las instancias de los atributos en cada nodo han sido definidas (es decir, cada instancia se ha asociado a un valor de su dominio), se denomina* ***árbol decorado****.*

Teorema 2.2.2 *El algoritmo de decisión de circularidad para una gramática de atributos requiere tiempo exponencial.*

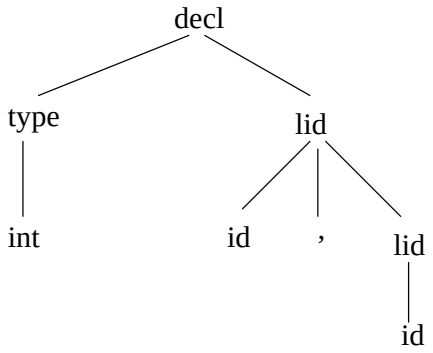

Figura 2.3: Árbol sintáctico para la cadena *int a,b*.

Demostración: Se verá en el capítulo 3, donde se hace un análisis de tiempo de ejecución del algoritmo de Knuth. Otra demostración (de Jazayeri) se puede encontrar en [20]. □

Definición 2.2.12 *Una gramática de atributos es* ***bien definida (WDAG)*** *si es completa y el algoritmo 1 no detecta circularidades.*

2.3. Semántica

Definición 2.3.1 *El significado de una cadena $\alpha \in \mathcal{L}(G)$, es el conjunto de valores $A(S)$, es decir, el conjunto de los valores de los atributos sintetizados asociados al símbolo de comienzo S.*

Las gramáticas de atributos bien definidas aseguran la existencia de una secuencia (consistente) finita de pasos para la evaluación de cada instancia de los atributos de cualquiera de sus árboles atribuidos. Además, también aseguran que cualquier secuencia elegida produce la misma valuación.

En la figura 2.3 se muestra un ejemplo de un árbol sintáctico para la GA de la figura 2.1 para la cadena *int a,b*.

En la figura 2.4 se muestran los grafos de dependencias de cada producción[10] $DP(p)$.

Las flechas gruesas denotan la producción aplicada y las flechas finas representan la relación de dependencia entre los atributos que ocurren en la instancia de la producción.

En la figura 2.5 se muestra el grafo de dependencias para el árbol de derivación de la cadena *int a, b;*.

El grafo de la figura 2.5 no contiene ningún ciclo, por lo tanto es posible encontrar una secuencia de evaluaciones de reglas de atribución consistente con las dependencias.

La secuencia <*type.t, type.size,* $lid^1.t$, $lid^1.size$, $id^2.t$, $id^2.size$, $lid^2.t$, $lid^2.size$, $id^3.t$, $id^3.size$, $lid^2.mem$, $lid^1.mem$, *decl.mem*> es un orden consistente para el grafo de dependencias de la figura 2.5.

En el próximo capítulo se definen clases o familias de gramáticas de atributos que permiten obtener métodos de evaluación estática eficientes y permiten que el test de circularidad para esas familias se pueda hacer en tiempo polinomial. Esto se logra con

[10] Se omiten los grafos de dependencias de *p1* y *p2*, ya que sus atributos son independientes.

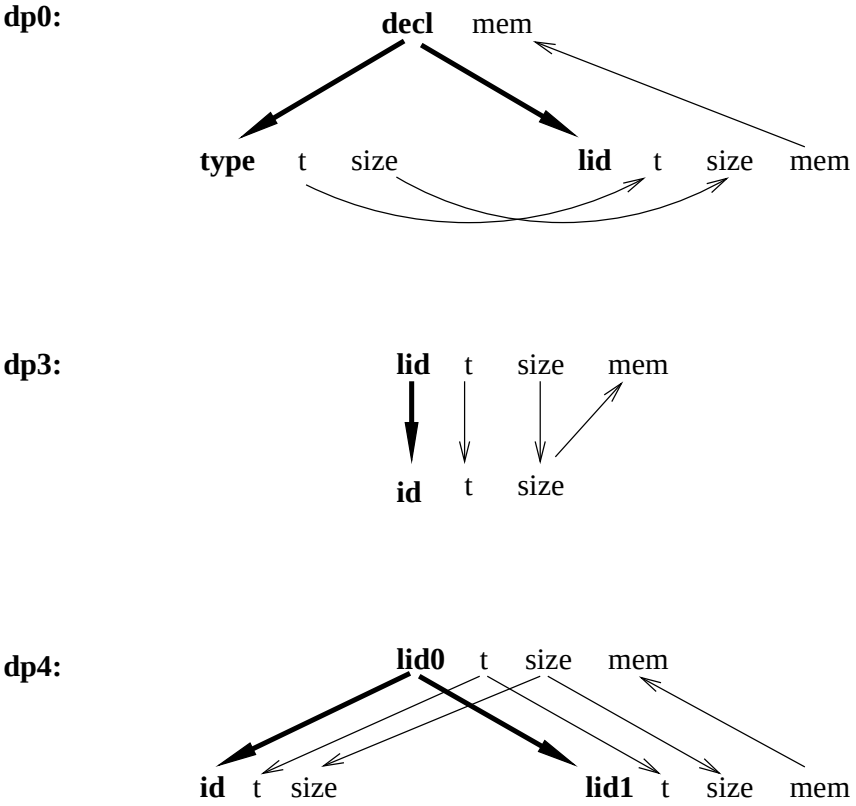

Figura 2.4: Grafos de dependencias de cada producción.

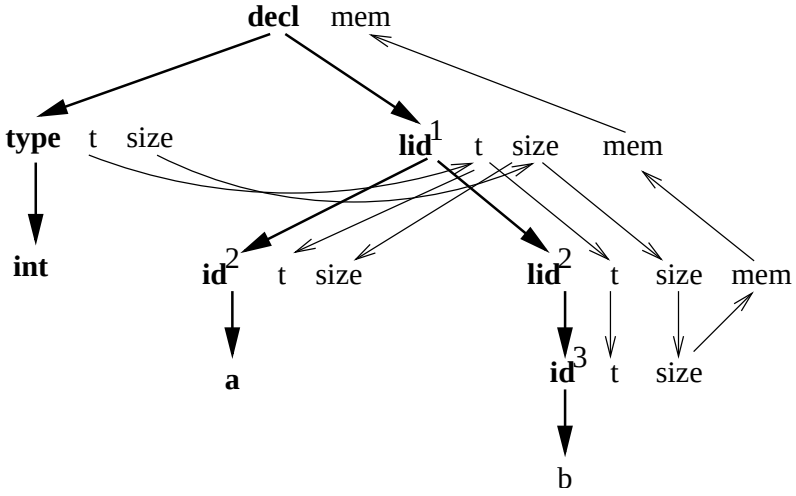

Figura 2.5: GD^T para el árbol atribuido para la cadena *int a, b;*.

el costo de restringir el poder expresivo del formalismo pero que han demostrado que son utilizables en la mayoría de las aplicaciones prácticas.

Teorema 2.3.1 *Sea GA_1 una gramática de atributos bien definida. Es posible obtener una gramática de atributos GA_2, equivalente a GA_1, tal que GA_2 no contiene atributos heredados.*

Demostración: los atributos heredados se utilizan para llevar información dependiente del contexto para computar atributos de símbolos en los contextos inferiores (subárboles).

Se presentan dos posibles enfoques:

1. **Modificación de la gramática libre de contexto**: la solución consiste en aplicar repetidamente a las producciones, reglas de factorización y eliminación de recursión a izquierda, para finalmente reemplazar los atributos heredados por sintetizados.
   Se ilustran los dos esquemas posibles de transformación:

   a) una producción de la forma $X \rightarrow \alpha Y \beta Z \gamma$, donde un atributo heredado de Y depende de atributos de Z, y donde $Y \underset{G}{\overset{*}{\Longrightarrow}} t$, se reemplaza por las producciones $X \rightarrow \alpha t Y$ e $Y \rightarrow \beta Z \gamma$.

   b) una producción de la forma $X \rightarrow \alpha Z \beta Y \gamma$, donde un atributo heredado de Y depende de atributos de Z, y donde $Y \underset{G}{\overset{*}{\Longrightarrow}} t$, se reemplaza por las producciones $X \rightarrow Y \gamma$ e $Y \rightarrow \alpha Z \beta t$.

   Claramente se puede ver que es posible reemplazar el atributo heredado de Y por uno sintetizado y el lenguaje generado por la gramática libre de contexto es equivalente.

2. **Modificación del dominio de los atributos**: Sea G una gramática de atributos con atributos heredados. Sea G' una gramática de atributos obtenida a partir de G en la cual se eliminan todos los atributos heredados y el dominio del conjunto de atributos heredados de A es reemplazado por un conjunto de funciones $Ops(A)$ que reflejan las dependencias de los atributos atributos heredados de G desde sus atributos sintetizados.

   El conjunto $Ops(A)$ contiene funciones de la forma $f_{(\sigma,x)}$, las cuales reflejan las dependencias funcionales del valor del atributo σ del símbolo x desde sus atributos heredados con respecto a G (la gramática original).

   De esta manera los valores de los atributos son funciones. La evaluación de atributos arroja como resultado la composición de las funciones. Como G' no es circular, eventualmente se computará una función constante la cual servirá como valor inicial para la aplicación de la composición funcional.

   Este enfoque tiene como ventaja que no es necesario modificar la gramática libre de contexto subyacente, pero como desventaja resulta que cada acceso a un atributo se transforma en la aplicación de una función.

La idea es similar al cambio del dominio de los atributos heredados por sintetizados con estructuras de árbol. De esta manera la nueva gramática obtenida sintetiza un árbol, sobre la cual se aplica una función de *decorado* (o computación de valores de los atributos) en cada nodo del árbol. La función deberá recorrer el árbol en un orden consistente con las dependencias.

Otro enfoque similar puede encontrarse en [33], donde realizan la evaluación durante el parsing descendente y se computan aproximaxiones a valores. Esas aproximaciones son equivalentes a valores parciales (expresiones aún no evaluadas), lo cual es equivalente a una implementación de evaluación lazy.

En esta demostración las aproximaciones son composiciones de funciones (aún no evaluadas).

□

$D \rightarrow L\ :\ T$	$D.t = T.type; L.t = T.type$
$L \rightarrow L\ ,\ id$	$L_1.t = L_0.t;\ id.t = L_0.t$
$L \rightarrow id$	$id.t = L.t$
$T \rightarrow int$	$T.type = INTEGER$

Figura 2.6: Una GA con atributos heredados ($\{L.t, id.t\}$).

$D \rightarrow id\ L$	$D.t = L.t$
$L \rightarrow,\ id\ L$	$L_0.t = L_1.t$
$L \rightarrow :\ T$	$L.t = T.type$
$T \rightarrow int$	$T.type = INTEGER$

Figura 2.7: Obtención de una GA sin atributos heredados (ej.1).

$D \rightarrow L\ :\ T$	$D.t = L.t_f(T.type)$
$L \rightarrow L\ ,\ id$	$L_0.t_f = \lambda t.(L_1.t_f)$
$L \rightarrow id$	$L.t_f = \lambda t.t$
$T \rightarrow int$	$T.type = INTEGER$

Figura 2.8: Una GA sin atributos heredados (ej.2).

La figura 2.6 muestra una gramática de atributos que computa el tipo de una declaración (al estilo Pascal) utilizando atributos heredados.

Las figuras 2.7 y 2.8 son ejemplos de transformación de una GA para que contenga sólo atributos sintetizados utilizando las dos técnicas usadas en la demostración del teorema 2.3.1, respectivamente.

En la figura 2.8 el atributo $L.t$ fue reemplazado por un atributo sintetizado $L.t_f$, el cual es una función que representa la relación funcional del atributo heredado $L.t$ con los atributos sintetizados en la GA original.

En este ejemplo la función define la identidad ($\lambda t.t$), ya que simplemente tiene que retornar su argumento.

Esta idea ilustra que es posible demorar la evaluación de aquellos atributos cuyas dependencias no hayan sido aún evaluadas. Estas técnicas se han usado en diferentes implementaciones como utilizando evaluación lazy o computando aproximaciones de valores como en [33].

Si bien el teorema 2.3.1 nos dice que es posible prescindir totalmente de los atributos heredados, éstos son muy útiles en la práctica para conseguir una mayor claridad y simplicidad de la gramática.

Una generalización de la segunda idea de transformación usada en la demostración del teorema 2.3.1 induce la idea de una *gramáticas de alto orden* (High Order Attribute Grammars), las cuales permiten que el dominio de los atributos sean árboles sintácticos derivado a partir de otra GA. Este tipo de GA se describen en el capítulo 7.

Capítulo 3

Clasificación de gramáticas de atributos

Es posible clasificar a las gramáticas de atributos en base a ciertas restricciones de las dependencias entre sus atributos. Cada familia tiene asociado un método propio de evaluación y aquellas que tienen más restricciones en sus dependencias permiten evaluadores más eficientes pero restringen su poder expresivo.

A continuación se presentan las diferentes familias (en orden creciente de poder expresivo) y se hace una discusión sobre el problema de decidibilidad sobre la pertencencia de una *GA* en particular a una famila determinada.

En la sección 3.1 se describe una clasificación basada en la estrategia de evaluación utilizada. En la sección 3.2 se realiza una clasificación basada en las dependencias y en la sección 3.3 se presenta una nueva clasificación, denominada *la jerarquía NC* propuesta recientemente por Wuu Yang en [44].

También se presenta un toerema que describe la relación entre las familias de GA descriptas en cuanto a su poder expresivo. En el próximo capítulo se describen métodos o estrategias de evaluación que se pueden aplicar a cada una de las clases que se describen a continuación.

3.1. Clasificación basada en la estrategia de evaluación

Es posible hacer una clasificación de gramáticas de atributos en base a procedimientos de evaluación previamente definidos. Uno de los primeros procedimientos propuestos en la literatura se basa en la idea de hacer varios recorridos del árbol atribuido e ir evaluando los atributos que sean posibles en cada uno de ellos. Luego de varios recorridos todos los atributos deberían estar evaluados (por supuesto si el grafo de dependencias es acíclico). Esta estrategia requiere que se tenga información sobre las dependencias entre los atributos durante la evaluación.

Es posible detectar estáticamente si a una GA dada se le puede aplicar un algoritmo de evaluación determinado y, en caso de ser necesario, generar la información sobre las dependencias para el evaluador.

A continuación de analiza una clasificación basada en la estrategia de evaluación partiendo desde la estrategia más simple a las de múltiples pasadas.

En el próximo capítulo se describirán algoritmos para evaluar AG que pertenezcan a alguna de las siguientes familias.

3.1.1. Gramáticas de atributos sintetizadas (S-AG)

Esta es la clase más restrictiva de GA ya que sólo permite atributos sintetizados y se conocen generalmente como *s-atribuidas*. Esta familia de GA's se pueden evaluar en un solo recorrido ascendente del árbol atribuido y es de aplicación directa en parsers ascendentes.

Definición 3.1.1 *Una* $GA = (G, A, V, Dom, F, R)$ *es una GA s-atribuida si y solo si el conjunto* $I(A) = \emptyset$ *y el grafo de dependencias directas de cada producción* $DP(p)$ *no contiene ciclos.*

3.1.2. Gramáticas de atributos l-atribuidas (L-AG)

También denominadas *1-PAGs (1 Pass Attribute Grammars)*. Pueden ser evaluadas en un solo recorrido descendente de izquierda a derecha. Esta famila de GA tienen como característica que en cada producción p, los atributos de un símbolo de la parte derecha de p, X_i, dependen sólo de atributos de los símbolos X_j que aparecen a la izquierda de X_i.

Definición 3.1.2 *Una GA es l-atribuida o L-AG si y solo si es bien definida y para cada producción* $p \in P$ *de la forma* $p : X_0 \rightarrow X_1 X_2 \dots X_{np}$ *se cumple:*

1. *Si* $(X_i.a, X_j.b) \in DG(p)$ *entonces* $(1 \leq j \leq np) \Rightarrow i < j$, *y*

2. $DP(p)$ *es acíclico.*

Esta familia de GA ha sido ampliamente utilizada en herramientas de generación de parsers ya que pueden evaluarse durante el parsing y es una familia que permite resolver problemas simples en la práctica. Sin embargo su limitación es bastante notable en muchas construcciones que se encuentran comunmente en lenguajes de programación, como por ejemplo en una declaración tipo Pascal:

decl: *id-list* ':' *type* ';'

El atributo *type* de cada identificador no puede ser computado hasta que se conozca el tipo de la declaración, por lo que una GA para esta construcción no sería *l-atribuida*. En la práctica estas limitaciones se resuelven generalmente sintetizando listas de identificadores en un atributo de *id-list* y se asigna el tipo de cada uno (en una tabla de símbolos global) luego de la reducción de la regla *decl*.

3.1.3. GA's evaluables en n-recorridos (n-PAG)

Obviamente existen gramáticas de atributos que no pueden ser evaluadas en un solo recorrido (ascendente ni descendente), pero que podrían ser evaluadas en varios recorridos (por ejemplo, descendentes, de izquierda a derecha). Cada recorrido evaluará los atributos que pueda (aquellos cuyos atributos de los cuales dependen se hayan evaluado en recorridos anteriores).

Si una gramática de atributos GA pertenece a esta familia, para cada $a \in A$ existe un número k_a, $1 \leq k_a \leq n$ tal que cualquier instancia de a se evalúa en el recorrido k-ésimo en cualquier árbol atribuido.

El número máximo de pasadas necesarias es $max(k_a)$, $\forall a \in A$.

3.1.4. GA evaluables en m pasos alternantes (m-APAG)

Una generalización o extensión del método anterior es alternar recorridos en cada pasada: en la pasada i-ésima se realiza un recorrido descendente de izquierda a derecha (derecha a izquierda) si i es par (impar). Por cada n-PAG existe un $m \leq 2n-1$ tal que es m-APAG.

3.1.5. Jerarquía de GA's según la estrategia de evaluación

En esta sección se analiza la inclusión de las familias de GA caracterizadas arriba según su poder expresivo.

Teorema 3.1.1 *Se mantiene la siguiente relación entre las familias $S-AG$, $L-AG$, $n-PAG$ y $m-PAG$, según su poder expresivo.*

$$S-AG \subseteq L-AG \subseteq n-PAG \subseteq m-APAG$$

Demostración: Una familia de GA tiene mayor poder expresivo si impone menos restricciones en las dependencias entre los atributos.

Es obvio determinar la inclusión de una familia en otra a partir de la definición del modo de evaluación. Una $S-AG$ es una $L-AG$ con el conjunto $S(X) = \emptyset$, para todo símbolo X de la GA. Una $n-PAG$ es obviamente una $L-AG$ ya que estas últimas son equivalentes a una $1-PAG$ utilizando un recorrido ascendente de izquierda a derecha. Finalmente una $m-PAG$ es una $n-PAG$ sin alternar los recorridos. Obviamente una $m-PAG$ impone menos restricciones que una $n-PAG$ ya que estas últimas imponen que las dependencias ocurran en todo árbol atribuido en el sentido del recorrido preestablecido, mientras que en una $m-PAG$ si ocurren dependencias en otro sentido podrían ser evaluadas en algún paso que utilize el recorrido adecuado. □

A continuación se presenta otra clasificación basada en las dependencias que resulta del análisis de los posibles contextos inferiores de una instancia de una producción. La clasificación es mas homogénea ya que se basa en el mismo concepto, el cual se va extendiendo en cada nueva familia.

3.2. Clasificación basada en las dependencias

En ésta sección se describen familias de gramáticas de atributos en las que las dependencias entre sus atributos tienen ciertas características que permiten determinar

estáticamente[1] un orden total de sus instancias de atributos para cada producción que sea consistente con sus dependencias.

3.2.1. Gramáticas de atributos ordenadas (OAG)

En una gramática de atributos de esta clase, los atributos de un símbolo $X \in \Sigma$ pueden ser evaluados en un orden (total) determinado, es decir que es posible determinar un orden de evaluación independientemente de su contexto. Esta información se puede utilizar para producir evaluadores para esta familia que sean muy eficientes, tanto en tiempo como en espacio y el algoritmo de verificación de circularidad tiene tiempo polinomial.

---

p_0: $S \rightarrow A$
attribution
$A.i1 = S.in$
$A.i2 = A.s1$
$S.out = A.s2$
end

p_1: $A \rightarrow A\ A$
attribution
$A_1.i1 = A_0.i1$
$A_1.i2 = A_1.s1$
$A_0.s1 = A_1.s2$
$A_2.i1 = A_0.i2$
$A_2.i2 = A_2.s1$
$A_0.s2 = A_2.s2$
end

p_2: $A \rightarrow \lambda$
attribution
$A.s1 = A.i1 + 1$
$A.s2 = A.i2 + 2$
end

Figura 3.1: Una gramática de atributos OAG (ordered Attribute Grammar)

La figura 3.1 muestra una gramática de atributos *ordenada*. Un orden total consistente para A es $\{i1 \rightarrow s1 \rightarrow i2 \rightarrow s2\}$. Esta gramática de atributos no se puede evaluar en un número fijo de pasadas. La figura 3.2 muestra el grafo de dependecias aumentado para obtener un orden total.

Esta familia es una subclase de las *l-ordered AG*, definidas también por Kastens (ver [23]), para las cuales la decisión de que si una GA pertence a ésta familia no puede ser determinado en tiempo polinomial. Una OAG es una *l-ordered* a la que se le han adicionado dependencias denominadas *dependencias aumentadas*. Una GA *l-ordered* a la que se le han aumentado sus dependencias para ser transformada en OAG, se denomina *arreglada apropiadamente*.

[1] Analizando las reglas de atribución de las producciones.

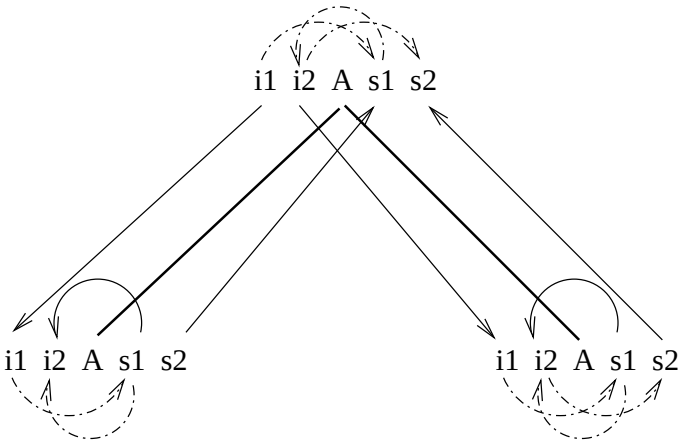

Figura 3.2: Grafo de dependencias de la producción $p1$ aumentado (arcos con línea de puntos) con un orden total sobre las dependencias de los atributos del símbolo A.

Definición 3.2.1 *Dada una GA, una familia de órdenes totales $O = \{O(X)\}$, donde cada $O(X)$ es un orden total sobre los atributos del símbolo $X \in \Sigma$, es consistente con el grafo de dependencias de la producción $p : X_0 \to X_1 X_2 \dots X_{np}$, si el grafo de dependencias aumentado con dichos órdenes es no circular:*

$$DP_{ord}(p) \equiv DP(p) \cup \{(X.a, X.b) \in O(X) \mid X \text{ ocurre en } p\}$$

Definición 3.2.2 *Una gramática de atributos es (ordenada) (OAG) si para cada producción existe un orden total, es decir que cada grafo $DP_{ord}(p)$ no es circular.*

En 1980, Uwe Kastens[23] propuso un algoritmo para computar los ordenes totales en tiempo polinomial, el cual se describe a continuación.

3.2.2. Determinación de que una gramática de atributos es OAG

Definición 3.2.3 *Dada una GA, la relación de dependencias inducidas entre las ocurrencias de atributos para la producción p está determinado como:*

$$IDP(p) \equiv DP(p) \cup \{(X.a, X.b) \mid \exists q \in P : (X.a, X.b) \in (IDP(q))^+\}$$

para todo símbolo X que ocurre en las producciones p y q.

$$IDP \equiv \bigcup_{p \in P} IDP(p)$$

Definición 3.2.4 *La relación de dependencias inducidas entre los atributos del símbolo $X \in \Sigma$:*

$$IDS(X) \equiv \{(X.a, X.b) \mid \exists p \in P \wedge (X.a, X.b) \in IDP(p)\}$$

$$IDS \equiv \bigcup_{X \in N} IDS(X)$$

Si $(X.a, X.b) \in IDS(X)$, entonces $(X.a, X.b) \in IDP(p)$ para cada ocurrencia de X en la regla p y si existe una dependencia entre las instancias de atributos $K_X.a$ y $K_X.b$ en un árbol atribuído para alguna sentencia del lenguaje entonces $(X.a, X.b) \in IDS$.

El hecho que $(X.a, X.b) \in IDS$ no implica que necesariamente existirá una dependencia entre las instancias correspondientes en algún árbol atribuido. En este sentido la relación $IDS(X)$ es pesimista. Lo anterior es fácil de ver ya que es posible que dos dependencias de $IDS(X)$ nunca puedan darse en forma simultánea en ningún árbol de derivación.

Kastens sugiere un método para construir las dependencias aumentadas para una *m-PAG* eficientemente, el cual se describe a continuación. Antes de describir el proceso, son necesarias algunas definiciones.

Definición 3.2.5 *Sea IDS acíclico: Para cada $X \in N$ definimos los siguientes conjuntos recursivamente:*

$$\begin{array}{lcl} A_{X,1} & = & \{X.a \in S(A) \mid \nexists X.b : (X.a, X.b) \in IDS^+\} \\ A_{X,2n} & = & \{X.a \in I(A) \mid \forall X.b : (X.a, X.b) \in IDS^+ \Rightarrow \\ & & \quad X.b \in A_{X,m}, m \leq 2n\} - \bigcup_{k=1}^{2n-1} A_{X,k} \\ A_{X,2n+1} & = & \{X.a \in S(A) \mid \forall X.b : (X.a, X.b) \in IDS^+ \\ & & \quad \Rightarrow X.b \in A_{X,m}, m \leq 2n+1\} - \bigcup_{k=1}^{2n} A_{X,k} \\ \ldots A_{X,m_x}: & & \textit{hasta que cada } X.a \in A(X) \textit{ esté en algún } A_{X,k}. \end{array}$$

Intuitivamente, cada $A_{X,k}$ contiene los atributos que contribuyen a la computación de los atributos contenidos en $A_{X,k-1}$.

El proceso de construcción de los conjuntos $A_{X,i}$ finaliza ya que el conjunto de atributos del símbolo X de una GA es finito.

Los conjuntos $A_{X,k}$, con $1 \leq k \leq m_x$, forman una partición disjunta de $A(X)$ (los atributos del símbolo X). Esto es

$$A(X) = \bigcup_{k=1}^{m_x} A_{X,k} \qquad \text{para } m_x \geq 1$$

$$A_{X,k} \cap A_{X,j} \neq \emptyset \Rightarrow k = j$$

Definición 3.2.6 *Sea IDS acíclico. La relación de dependencia completa se define como:*

$$DS \equiv \bigcup_{X \in N} DS(X)$$

$$DS(X) \equiv IDS(X) \cup \{(X.a, X.b) \mid X.a \in A_{X,k}, X.b \in A_{X,k-1}, \textit{tal que } 2 \leq k \leq m_x\}$$

También es necesario completar la relación de dependencia IDP de acuerdo a DS para verificar que no se produzcan ciclos.

Definición 3.2.7 *La relación extendida de dependencia, EDP:*

$$EDP \equiv \bigcup_{p \in P} EDP(p)$$

$$EDP(p) \equiv DP(p) \cup \{(X.a, X.b) \mid (X.a, X.b) \in DS(X)\}$$

Definición 3.2.8 *la relación extendida DS es compatible con las dependencias de atributos si cada grafo EDP(p), $\forall p \in P$ es acíclico.*

Ahora es posible redefinir a la famila de gramáticas de atributos ordenadas.

Definición 3.2.9 *Una GA es una OAG si y sólo si la relación de dependencia DS existe y es compatible con las dependencias de los atributos.*

En ocasiones, puede suceder que la aplicación del este algoritmo a una AG dada, haga que su DS no sea compatible con las dependencias de sus atributos, causado por la extensión (completitud) de la relación.

Si algún grafo de $EDP(p)$ es cíclico, se puede ver que cada ciclo contiene al menos dos arcos $(X.a, X.b)$ y $(Y.c, Y.d)$ que pertenecen a DS/IDS, donde X e Y corresponden a distintas ocurrencias de símbolos en p. Por otra parte $X.a, X.b, Y.c$ e $Y.d$ son independientes en IDS^+. Los arcos están en DS porque los atributos involucrados pertenecen a diferentes subconjuntos de $A(X)$ y $A(Y)$.

Si existen atributos independientes en IDS^+, se puede imponer una partición disjunta de A agregando dependencias adicionales a las que surgan de las dependencias directas la atribución. En este caso, este conjunto arbitrariamente definido, $ADS \subseteq (A \times A)$ se denomina *conjunto aumentado de dependencias.*

Definición 3.2.10 *Sea DP el grafo de dependencias directas de una GA y sea ADS un conjunto de dependencias aumentadas. Sea DS′ el grafo computado desde DP′, donde*

$$DP'(p) \equiv DP(p) \cup \{(X_i.a, X_i.b) \mid (X.a, X.b) \in ADS\}$$

Una GA está adaptada ordenadamente por ADS si es una OAG, teniendo en cuenta DS′.

Es interesante hacer notar que una GA es adaptada ordenadamente si ADS tiene la siguiente propiedad: contiene arcos $(X.b, X.a)$ tales que, en cada ciclo del grafo EDP original, al menos un arco $(X_i.a, X_i.b)$ puede ser reemplazado por $(X_i.b, X_i.a)$ sin que se introducan nuevos ciclos.

La computación de un ADS tiene una gran complejidad computacional ya que es un problema combinatorio. Afortunadamente en la práctica se presentan muy pocos casos donde la complejidad combinatoria lo haga intratable.

A continuación se presenta una extensión que permite aumentar las dependencias mediante un algoritmo, el cual tiene complejidad polinomial y que permite caracterizar a una familia mas amplia que las OAG.

3.2.3. OAG extendidas (EOAG)

En 1996, Wuu Yang[41] caracterizó una extensión a las OAG que permite que el algoritmo de decisión de pertenencia a esta nueva familia sea de tiempo polinomial. A ésta extensión se la conoce como *Extended OAG.*

La observación realizada permite determinar conjuntos de atributos de un símbolo que sean *promovibles*, los cuales pueden moverse a una partición mas alta de la cual podría ser asignados por el algoritmo visto en la sección anterior.

El algoritmo propuesto, el cual se describe en esta sección, introduce un paso adicional, previo a la construcción del particionado, que consiste en la construcción de los conjuntos de dependencias aumentadas de los símbolos ($ADS(X)$) y redefine el particionado para que se tengan en cuenta estas dependencias.

Si se realiza un análisis de las causas de ciclos en algún grafo de dependencias ($EDP(p)$) es posible observar que los ciclos incluyen atributos *significativos* y *co-significativos*.

Intituivamente, un atributo es *significativo* si tiene un arco saliente hacia un atributo de otro símbolo en $DP(p)$ y un atributo es *co-significativo* si tiene un arco entrante desde un atributo de otro símbolo en $DP(p)$, para alguna producción p.

Forzando un orden (correcto) de evaluación entre los atributos *significativos* y *co-significativos* es posible encontrar un orden de evaluación correcto para algunas *l-ordered AG* que no son *OAG*.

A continuación se enuncian definiciones para llegar a la caracterización de las *EOAG*.

Definición 3.2.11 *Dada G una GA, sean X e Y dos ocurrencias de símbolos*[2] *en la producción p de G:*
$Out(X, Y, p) = \{X.a \mid X.a, X.b \text{ ocurren en } p \wedge (X.a \rightarrow Y.b) \in DP(p)\}$
$In(X, Y, p) = \{X.a \mid X.a, X.b \text{ ocurren en } p \wedge (Y.b \rightarrow X.a) \in DP(p)\}$

Dado que la GA está en forma normal, todas las ocurrencias de atributos en $Out(X, Y, p)$ deben ser ocurrencias de atributos sintetizados de X si X aparece en la parte derecha de la producción p u ocurrencias de atributos heredados de X si X aparece en la parte izquierda de p. Similarmente, todas las ocurrencias de los atributos de $In(X, Y, p)$ serán ocurrencias de atributos heredados de X si X aparece en la parte derecha de p u ocurrencias de atributos sintetizados de X si X aparece en la parte izquierda de p.

Definición 3.2.12 *Sean X e Y dos símbolos que ocurren en la producción p. Y es alcanzable desde X en p si* $(X.a, Y.b) \in DP(p)$.

Definición 3.2.13 *Sean X e Y dos símbolos que ocurren en la producción p. Y es objetivo de X si X alcanzable*$^+$ *Y.*

Definición 3.2.14 *Un atributo X.a es significativo si una ocurrencia de* $X.a \in Out(X, Y, p)$, *para las ocurrencias de los símbolos X e Y en p. Un atributo X.b es co-significativo si una ocurrencia de* $X.b \in Out(X, Y, p)$. *Un atributo significativo X.a corresponde a un atributo co-significativo X.b si en alguna producción p,* $X.a \in Out(X, Y, p)$, $X.b \in In(X, Z, p)$ *y Z es objetivo de Y en p.*

En un grafo de EDP puede haber dos clases de ciclos: entre atributos de un mismo símbolo y ciclos en los cuales aparecen atributos de diferentes símbolos. En estos últimos tipos de ciclos, existe un camino entre la ocurrencia de un atributo co-significativo

[2]Notar que X e Y pueden ser dos ocurrencias del mismo símbolo.

a una ocurrencia de un atributo significativo. La idea es agregar arcos en la dirección inversa: desde un atributo significativo a uno no-significativo. Sin embargo, esta forma de agregado de arcos también podría crear ciclos que no podrían ser creados por la definición 3.2.5, por lo que sólo deberían agregarse sólo aquellos que cumplan con ciertas condiciones.

Definición 3.2.15 *Un atributo significativo de X es promovible si*

1. *no es alcanzable desde ningún atributo co-significativo de $IDS(X)$ y*
2. *no es un atributo co-significativo.*

Intituivamente, un atributo *promovible* puede ser promovido a una partición más alta de la que pueda ser asignado por el algoritmo *OAG*.

Definición 3.2.16 *Sea $IDS(X)$ acíclico, para cada símbolo X,*
$ADS(X) = IDS(X) \cup \{(X.a \rightarrow X.b) | X.a$ *es promovible y* $X.b$ *es co-significativo y corresponde a* $X.a\}$

$$ADS = \bigcup_{X \in \Sigma} ADS(X)$$

Las particiones de atributos $A_{X,k}$, los grafos $DS(X)$ y los grafos de dependencias extendidas de cada producción ($EDP(p)$), ahora se definen en base a $ADS(X)$, en lugar de $IDS(X)$ como en la sección anterior.

Definición 3.2.17 *Sea ADS acíclico ($\forall X \in V$): se definen los siguientes conjuntos recursivamente:*

$$\begin{array}{lcl} A_{X,1} & = & \{X.a \in S(A) \mid \nexists X.b : (X.a, X.b) \in ADS(X)^+\} \\ A_{X,2n} & = & \{X.a \in I(A) \mid \forall X.b : (X.a, X.b) \in ADS(X)^+ \Rightarrow \\ & & \quad X.b \in A_{X,m}, m \leq 2n\} - \bigcup_{k=1}^{2n-1} A_{X,k} \\ A_{X,2n+1} & = & \{X.a \in S(A) \mid \forall X.b : (X.a, X.b) \in ADS(X)^+ \Rightarrow \\ & & \quad X.b \in A_{X,m}, m \leq 2n+1\} - \bigcup_{k=1}^{2n} A_{X,k}, \\ \ldots & & \textit{hasta que cada } X.a \in A(X) \textit{ esté en algún } A_{X,k}. \end{array}$$

Definición 3.2.18 *La relación de dependencia completa DS se define como:*

$$DS = \bigcup_{X \in V} DS(X)$$

$$DS(X) = ADS(X) \cup \{(X.a \rightarrow X.b) \mid X.a \in A_{X,k}, X.b \in A_{X,k-1}\}$$

Definición 3.2.19 *La relación extendida de dependencias, EDP:*

$$EDP \equiv \bigcup_{p \in P} EDP(p)$$

$$EDP(p) \equiv DP(p) \cup \{(X.a, X.b) \mid (X.a, X.b) \in ADS(X)\}$$

Definición 3.2.20 *Una GA es EOAG si y sólo si DS existe y EDP es acíclico.*

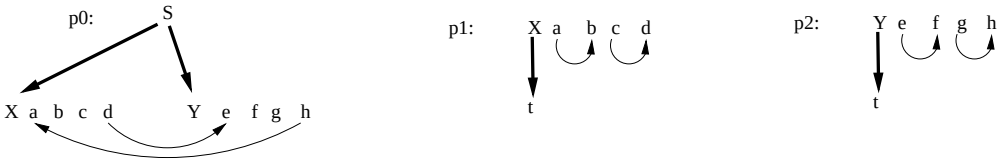

Figura 3.3: Ejemplo de una gramática de atributos EOAG

Ejemplo: en la figura 3.3 se muestra una GA con tres producciones: $p_0 : S \rightarrow X\,Y$, $p_1 : X \rightarrow t$ y $p_2 : Y \rightarrow t$. El símbolo X tiene asociados los atributos *a,b,c* y d e Y tiene los atributos *e,f,g* y *h*. Las flechas delgadas muestran las dependencias entre los atributos (grafos $DP(p)$).

En la figura 3.4 se pueden apreciar los grafos de $EDP(p_0)$. La gramática de la figura 3.3 no es OAG ya que $EDP(p_0)$ a) contiene un ciclo ($a \rightarrow d \rightarrow e \rightarrow h \rightarrow a$). El grafo $EDP(p_0)$ b) no contiene ciclos, por lo tanto es $EOAG$.

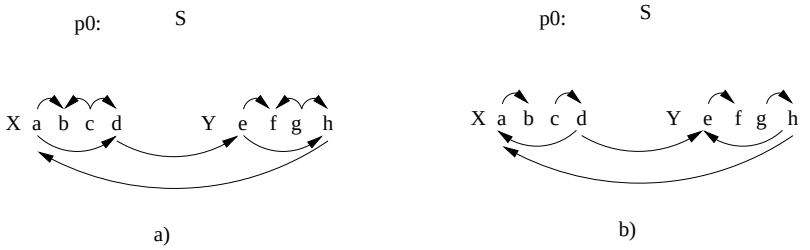

Figura 3.4: a) EDP(p0) (OAG) y b) EDP(p0) (EOAG)

Teorema 3.2.1 *La familia EOAG incluye propiamente a la familia OAG.*

Demostración: En primer lugar se demostrará que toda OAG G es $EOAG$. Supóngase que $EDP(p)$ es cíclico (para alguna producción p). Como G es OAG, entonces IDS es acíclico, por lo que ADS también es acíclico. Si ADS no fuese acíclico, el ciclo se habría producido al adicionar un arco ($a \rightarrow b$), tal que a es promovible y b debe ser un atributo correspondiente no-significativo. Como a y b están incluidos en el ciclo, deberá haber un camino de un atributo co-significante b' a a en IDS, pero esto contradice la definición 3.2.15. Así ADS debe ser acíclico si IDS lo es.

Como $EDP(p)$ (en $EOAG$) es cíclico y $EDP(p)$ (en OAG) no lo es, deberá existir un camino entre $X_i.a$ y $X_i.b$, es decir $X_i.arightarrow^+ X_i.b$, (denominado un segmento o camino *intra-símbolo*), y $X_i.a$ deberá tener un arco entrante de una ocurrencia de un atributo de otro símbolo y $X_i.b$ un arco saliente a una ocurrencia de un atributo de algún otro símbolo. Además, $X.i.a$ y $X_i.b$ no pueden ser ambos atributos sintetizados o heredados. Así, $X_i.a$ y $X_i.b$ aparecerán en diferentes particiones, por ejemplo en $A_{X_i,m}$ y $A_{X_i,n}$ (en OAG) y en $A_{X_i,m'}$ y $A_{X_i,n'}$ (en $EOAG$), respectivamente.

Como existe un camino ($X_i.a \rightarrow^+ X_i.b$) en $EDP(p)$ (en EOAG), $X_i.a$ deberá estar en una partición más alta que $X_i.b$, esto es, $m' > n'$. Como no existe tal camino en $EDP(p)$ (en OAG), $X_i.a$ deberá estar en una partición más baja que $X_i.b$ (en $EOAG$),

esto es, $m < n$. Pero por la definición 3.2.17, claramente $n' \geq n$ y $m' > m$[3]. Pero $X_i.a$ es un atributo co-significativo y por lo tanto deberá estar en la misma partición computada por 3.2.5 (OAG), así $m' = m$, por lo que llegamos a una contradicción.
Como un grafo $EDP(p)$ ($EOAG$) no puede contener ciclos si no los tiene el grafo $EDP(p)$ correspondiente a OAG, G debe ser una $EOAG$.

Finalmente, la inclusión propia se muestra en el ejemplo de la GA dada en las figuras 3.3 y 3.4, que demuestra que existen $EOAG$ que no son OAG. □

3.2.4. Gramáticas de atributos absolutamente no circulares (ANCAG)

Esta familia de gramáticas de atributos, denominada también *fuertemente no circulares*) no comprende las GAs cuyas dependencias entre sus atributos es tal que cada atributo sintetizado del símbolo de la parte izquierda de cada producción, depende (directa o indirectamente) de un conjunto de sus atributos heredados.

Esta relación de dependencias se conoce como $IS(X)$ (dependencias entre atributos heredados y sintetizados del símbolo X).

Es posible definir formalmente que una GA es $ANCAG$ si se extiende el grafo de dependencias de cada producción $p : X_0 \rightarrow X_1 X_2 \ldots X_{np}$:

$$IDP_{ANCAG} = \bigcup_{p \in P} IDP_{ANCAG}(p)$$

$$IDP_{ANCAG}(p) = DP(p) \cup \{(X.a, X.b) \mid \exists q : X \rightarrow \alpha \in P$$

$$\wedge (X.a, X.b) \in IDP_{ANCAG}(q)\}$$

Definición 3.2.21 *Una gramática de atributos es ANCAG si ningún grafo $IDP_{ANCAG}(p)$, para toda producción p, es circular.*

Se puede observar que la relación de dependencias inducidas para cada producción, es una estimación conservadora de las dependencias entre sus atributos. Nuevamente la relación $IDP - ANCAG(p)$ es pesimista ya que podría contener dependencias que nunca se podrían dar simultáneamente en un árbol sintáctico. Se debe notar que un grafo $IDP-ANCAG(p)$ contiene todas las dependencias transitivas entre los atributos que ocurren en p para la unión de todos los posibles contextos inferiores.

3.2.5. Gramáticas de atributos bien definidas (WDAG) (o no circulares)

El la sección 2.2 del capítulo anterior se presentó la definición de las GA bien definidas. Esta familia excluye a las gramáticas de atributos circulares de la clase de gramáticas de atributos irrestrictas. Los evaluadores para esta familia tienen que enfrentar el problema de la determinación de secuencias (planes) de evaluación consistentes con sus dependencias.

[3]La adición de un arco $X.a \rightarrow X.b$ puede sólo incrementar el número de partición de $X.a$ y todos los atributos $X.c$ de los cuales se pueda alcanzar $X.a$.

El teorema 2.2.2 establece que la determinación si una GA en particular es circular, es decir si pertenece a ésta familia o no, es intrínsecamente exponencial. Esta complejidad en el algoritmo de decisión hace que la mayoría de los evaluadores de $WDAG$ no incluyan la prueba de circularidad, y utilicen alguno de los siguientes enfoques:

1. Para cada árbol atribuído T se construye su grafo de dependencias $GD(T)$ y se verifica que éste sea acíclico. Luego se produce la secuencia (plan) de evaluación computando un orden topológico de $GD(T)$. Posteriormente se puede utilizar un evaluador basado en secuencias de visita como en que se verá mas adelante.

   Este enfoque tiene un gran costo en tiempo de ejecución y de uso de memoria ya que debe construirse y mantenerse el grafo de dependencias durante la evaluación (run time). Este enfoque podría aplicarse a la familia de GA irrestrictas, dando un error si $GD(T)$ contiene al menos un ciclo.

2. Computar estáticamente todos los posibles planes de evaluación para la GA dada para que el evaluador seleccione (en tiempo de ejecución) el plan correspondiente para cada árbol atribuído.

   En el próximo capítulo se presenta un evaluador con estas características para la familia de gramáticas $NC(\infty)$, la cual se corresponde con las $WDAG$.

3. Ignorar el problema y comenzar a evaluar inicialmente los atributos que no dependan de otros atributos, marcarlos como computados, luego evaluar los atributos que dependan solamente de ellos y así sucesivamente. En el caso de existir dependencias cíclicas, el proceso no podrá continuar con la evaluación (en cuyo caso emitirá un mensaje de error en caso de que se realice detección de ciclos, o se producirá un deadlock).

   Esta estrategia de evaluación se conoce comúnmente como *evaluación bajo demanda o por necesidad*.

En [4] se presenta un evaluador concurrente, orientado a objetos que sigue esta estrategia, en cuyo diseño e implementación ha participado el autor de esta tesis.

La mayoría de los generadores de compiladores se basan en esquemas de traducción, produciendo la evaluación de atributos de esta forma (ej: yacc, javacc, java

En el próximo capítulo se mostrará que a pesar de la complejidad de la determinación que una GA pertenece a la familia de las $WDAG$. Se presenta un algoritmo que aún siendo exponencial, el exponente es una constante pequeña (generalmente no mayor que 5) en la práctica, lo que hace que el algoritmo sea perfectamente aplicable en una herramienta corriendo en una PC actual.

3.2.6. Gramáticas de atributos irrestrictas

Esta es la familia de gramáticas de atributos más general, sin ninguna restricción en las dependencias de sus atributos. Las GAs que pertenecen a esta familia son aquellas gramáticas que podrían generar árboles atribuídos cuyo grafo de dependencias sea circular, para el cual no existirá un orden consistente de evaluación. De todos modos, si existe un punto fijo para las dependencias circulares (recursivas) se podrían evaluar bajo demanda utilizando evaluación lazy como se descibe en [1] y [16].

Un resumen de varios métodos de evaluación se puede encontrar en [2].

3.3. La familia NC

En 1999 Wuu Yang[44] propone una nueva clasificación de *Gramáticas de atributos no circulares basadas en información de contextos inferiores (lookahead bahavior)*. Desde este punto de vista, las gramáticas de atributos pueden clasificarse en una nueva jerarquía, denominada la jerarquía ***NC***.

La jerarquía NC confirma el resultado de Riis y Skyum ([35]), quienes establecieron que es posible evaluar cualquier GA bien definida por medio de evaluadores estáticos multi-visita (como se verá en el próximo capítulo).

La clase *NC(0)* se corresponde con la familia ANCAG y la familia $NC(\infty)$ con las WDAG.

La idea básica en la jerarquía *NC* es tratar de computar grafos de dependencias inducidos para cada producción y cada símbolo no terminal, que sean aproximaciones mas reales (refinamientos) a los grafos utilizados en la caracterización de otras clases como las descriptas anteriormente.

Es posible encontrar aproximaciones mas exactas si se tienen en cuenta los posibles contextos inferiores de una instancia de aplicacación de una producción.

---

p_0: $S \rightarrow XYZ$
attribution
S.s0 := f(X.s1,Y.s2,Y.s3,Z.s4)
X.i1 := Y.s3
Y.i2 := X.s1
Y.i3 := Y.s2
end

p_1: $Y \rightarrow m$
attribution
Y.s2 := Y.i2
Y.s3 := 1
end

p_2: $Y \rightarrow n$
attribution
Y.s2 := 2
Y.s3 := Y.i3
end

p_3: $X \rightarrow m$
attribution
X.s1 := X.i1
end

p_4: $Z \rightarrow Y$
attribution
Z.s4 := Y.s3
Y.i2 := 3
Y.i3 := Y.s2
end

Figura 3.5: Una gramática de atributos no ANCAG

Definición 3.3.1 *Dada G, una GA El grafo de dependencias transitivas hacia abajo de un símbolo X para el árbol atribuido T(G) es el grafo* $DGC(X) = (A(X), E(X))$ *donde* $E(X) \equiv \{(X.a, X.b)\}$ *tal que* X_b *depende (transitivamente) de* $X.a$ *en* $T(G)$.

Los grafos $DGC(X), \forall X \in N$ son costosos de calcular, por lo cual se han propuesto en la literatura *aproximaciones seguras*(ver [2]). Una definición que ha sido ampliamente utilizada (para caracterizar las $GD_{ANCAG}(p)$) es la siguiente:

Definición 3.3.2 *Las dependencias Down(X) para un símbolo* $X_0 \in N$ *de una GA :*

Sea $p : X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \ldots X_k \alpha_k$, $(\alpha_i \in V_T^*, X_i \in V, 0 \leq i \leq k)$ *una producción en P, y sean* $q_1, q_2, \ldots, q_k$ *producciones con parte izquierda* $X_1, X_2, \ldots, X_k$, *respectivamente:*

$$IDP_{ANCAG}(p) = DP(p) \bigcup_{i=1}^{k} Down(X_i)$$

$$Down(X) = \{(X.a, X.b) \in IDP_{ANCAG}(q)^+ \mid X = lhs(q)\}$$

$$(lhs(q) = X_0 \text{ si y sólo si } q : X_0 \rightarrow \alpha)$$

Los grafos $DG(X)$ tienen en cuenta todas las posibles dependencias entre los atributos de un símbolo para cualquier árbol atribuído correspondiente a la GA. Esto permite que puedan existir algunas dependencias que no pueden ocurrir en forma simultánea en ningún árbol atribuido. Si se tienen en cuenta los posibles subárboles cuya raíz es X y se calculan las dependencias transitivas hacia abajo es posible obtener grafos de dependencias entre los atributos de un símbolo con mayor precisión.

La figura 3.5 muestra una GA que no es ANCAG.

La figura 3.6 muestra los grafos de dependencias de las producciones y la figura 3.7. Como se puede apreciar, la gramatica no es *ANCAG* ya que el grafo $IDP_{ANCAG}(P_0)$ contiene un ciclo.

Se puede notar que el ciclo es producto de considerar la unión de todas las posibles dependencias (transitivas) en cualquier árbol. Por ejemplo las dependencias $i2 \rightarrow s2$ y $i3 \rightarrow s3$ nunca podrán ocurrir simultáneamente en ningún árbol de derivación, sin embargo ambas aparecen, ocasionando el ciclo, en $IDP_{ANCAG}(P0)$.

3.3.1. Gramáticas de atributos NC(1)

En ésta sección se caracterizan las gramáticas $NC(1)$ y se mostrará que la familina NC(1) contiene a las ANCAG.

Definición 3.3.3 *Sea q una producción de una GA de la forma:* $X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \ldots X_k \alpha_k$ *y sea* $p_i, (1 \leq i \leq k)$ *una producción cuya parte izquierda es* X_i.

$$DCG_X(q) = \bigcup \{(X_0.a, X_0.b) \mid (X_0.a, X_0.b) \in ADP(q \mid p_1, p_2, \ldots, p_k)^+\}$$

donde $p_1, p_2, \ldots, p_k$ *son producciones cuyo símbolo de la parte izquierda es* X_i, $(1 \leq i \leq k)$ *y*

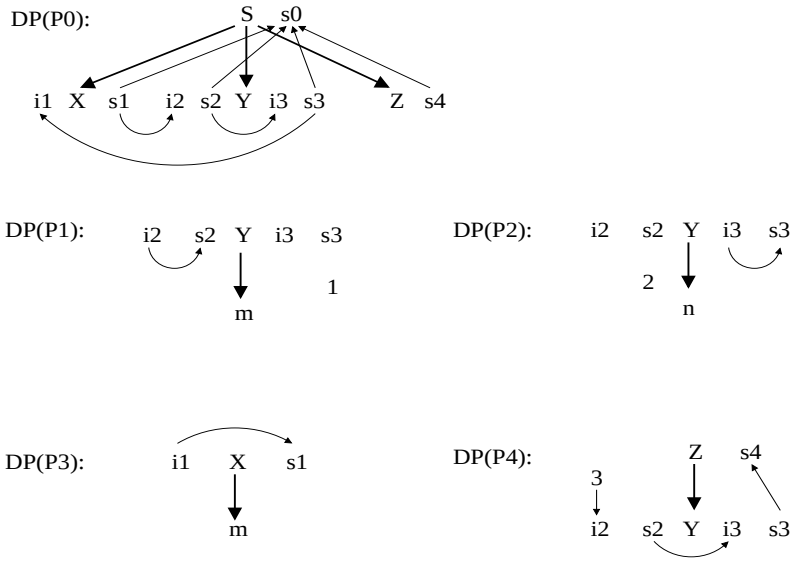

Figura 3.6: Grafos de dependencias directas (DP(p))

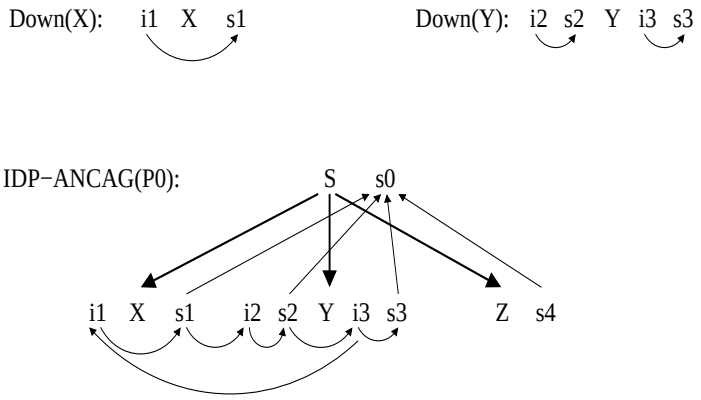

Figura 3.7: Grafos Down(X), Down(Y) e $IDP_{ANCAG}(P0)$

$$ADP(q \mid p_1, p_2, \ldots, p_k) = DP(q) \bigcup_{i=1}^{k} DGC_{Xi}(p_i)$$

$DCG_X(q)$ se denomina el grafo característico hacia abajo del símbolo X en los subárboles derivados por q.

$ADP(q \mid p_1, p_2, \ldots, p_k)$ es el grafo de dependencias aumentado de la producción q con subárboles derivados por $p_1, p_2, \ldots, p_k$.

Teorema 3.3.1

$$\bigcup_{X=lhs(q)} DCG_X(q) \subseteq DG(X)$$

Demostración: de las definiciones anteriores se puede ver que si una dependencia $(X.a, X.b)$ pertence a algún $DCG_X(q)$, deberá pertenecer en $DG(X)$ ya que éste último tiene en cuenta absolutamente todas las producciones en que X aparece en su parte izquierda, en particular q. □

Cada grafo $DCG_X(q)$ es una representación más precisa que $DG(X)$, ya que el primero contiene menos dependencias espúreas, ya que sólo podrán aparecer algunas, que pueden resultar del tener en cuenta los posibles subárboles a partir de la aplicación de la producción q.

Definición 3.3.4 *Sea q una producción de GA de la forma: $X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \ldots X_k \alpha_k$ y sea $p_i, (1 \leq i \leq k)$ una producción cuya parte izquierda es X_i.*

$$SADP(q) = \bigcup_{q \in P} ADP(q \mid p_1, p_2, \ldots, p_k)$$

Una GA es $NC(1)$ si y sólo si cada grafo de $SADP(q)$ es acíclico, $\forall q \in P$.

Como se verá en el próximo capítulo, los grafos $ADP(q \mid p_1 p_2 \ldots p_k)$ permiten generar *planes de evaluación* para la producción q, los cuales pueden evaluarse con un evaluador basado en *secuencias de visita*.

3.3.2. Gramáticas de atributos NC(m)

Los grafos $ADP(q \mid p_1 p_2 \ldots p_k)$ definidos en la sección anterior, representan las dependencias (transitivas) de los atributos de la producción q aplicando las producciones $p_1 p_2 \ldots p_k$ a los símbolos no terminales $X_1 X_2 \ldots X_k$ que ocurren en la parte derecha de q. Ellos tienen en cuenta una generación de descendentes (lookahead behavior).

Analizando mas generaciones de descendientes, se puede caracterizar una jerarquía de clases. La clase $NC(m)$ consiste de aquellas GA's para las cuales existe un conjunto de planes de evaluación para cada producción, los que quedan determinados analizando m niveles de producciones descendientes.

Para definir con mayor precisión la clase $NC(m)$ es necesario extender las definiciones de los grafos característicos descendentes.

Definición 3.3.5 *Sea $X \in V_N$ y m un entero no negativo, un árbol de frase (phrase tree), denotado $\tau_{<X,m>}$, es un árbol de derivación (posiblemente incompleto) el cual cumple con:*

1. *X es el rótulo de la raíz*
2. *La altura[4] del árbol es m*
3. *La longitud del camino desde la raíz hasta cualquier hoja rotulada con un no terminal es exactamente m*

[4]El camino más largo desde la raíz hasta alguna de las hojas.

Dado un árbol T, la *restricción de T hasta el nivel m*, denotado como $T_{<m>}$, es el subgrafo de T obtenido por la eliminación de todos los nodos por debajo del nivel m. Así, $T_{<1>}$ consiste de la raíz y sus hijos, la cual es escencialmente, la producción aplicada a la raíz.

Definición 3.3.6 *Sea $\tau_{<X,m>}$ un árbol de frase. El grafo característico descendente del no terminal X con m generaciones, con respecto a τ, denotado $DGC_X(\tau)$, es el grafo $G =< A(X), E(X) >$ donde*

$$E(X) = \{(X.b, X.a) \mid X.b \rightarrow X.a\}$$

es una dependencia (transitiva) en algún subárbol T derivado de X y $T_{<m>} = \tau$.

$DCG_X(\tau)$ puede definirse mas precisamente como:
Sea

$$q : X_0 \rightarrow \alpha_0 X_1 \alpha_1 X2 \ldots X_k \alpha_k$$

y sea τ_i un $< X_i, m >$ un árbol de frase ($1 \leq i \leq k$).

$$DCG_X(\tau) = \{(X.b, X.a) \mid (X.b, X.a) \in ADP_{<m>}(q \mid \tau_1 \tau_2 \ldots \tau_k)^+\}$$

tal que $(q \bigcup_{i=1}^{k} \tau_i) = \tau$ [5]

$$ADP_{<m>}(q \mid \tau_1 \tau_2 \ldots \tau_k) = DP(q) \bigcup_{i=1}^{k} DCG_{Xi}(\tau_i)$$

Los grafos $ADP_{<m>}(q \mid \tau_1 \tau_2 \ldots \tau_k)$ se denominan *grafo de dependencias aumentado de m-generaciones de la producción q con árboles de frase $\tau_1 \tau_2 \ldots \tau_k$.*

Definición 3.3.7 *Sea*

$$SADP_{<m>}(q) = \{ADP_{<m>}(q \mid \tau_1 \tau_2 \ldots \tau_k)\}$$

tal que τ_i ($1 \leq i \leq k$), es un $< X_i, m >$ árbol de frase.
Una gramática de atributos GA es $NC(m)$ si y sólo si $\forall q \in P$, cada grafo $g \in SADP_{<m>}(q)$ es acíclico.

3.3.3. Transformación de una gramática NC(m) a NC(0)

La unión de una producción q y los árboles de frase correspondientes a los símbolos no terminales de la parte derecha de q ($(q \bigcup_{i=1}^{k} \tau_i = \tau)$, se puede ver como una forma extendida de una producción. Es decir un árbol de frase de profundidad m puede *aplanarse* en una única producción.

Desde este punto de vista, una GA $NC(m)$ es equivalente a una GA' $NC(0)$.

Teorema 3.3.2 *Sea $AG = (G, A, V, Dom, F, R)$ una gramática de atributos $NC(m)$, existe una gramática de atributos AG' $NC(0)$ equivalente.*

[5] La notación $(q \bigcup_{i=1}^{k} \tau_i) = \tau$ denota el árbol obtenido a partir de la producción q y los τ_i como subárboles de los nodos rotulados X_i ($1 \leq i \leq k$).

Demostración: Sea AG' una GA $NC(m)$ (G', A, V, Dom, F, R) donde
$G' = (V_N, V_T, S, P')$ y P' contiene las producciones obtenidas como se describe a continuación.

Para cada q una producción de la forma 3.3.6 y sea $T = q \bigcup_{i=1}^{k} \tau_k$ de G,
la producción $q' : X_0 \rightarrow \alpha$, donde α está formada por la frontera de T es una producción de AG'.

Obviamente AG' es $NC(0)$ ya que las dependencias de cada producción q' representan las dependencias transitivas entre los atributos de X_0 y los atributos de las hojas de T. □

Aunque cada gramática de aributos $NC(m)$ se puede transformar en una $NC(0)$, el número de producciones se incrementa enormemente, el cual podría ser hasta $O(\mid P \mid\mid V_N \mid^{lm})$, donde $\mid P \mid$ es el número de producciones, $\mid V_N \mid$ es el número de no terminales y l la longitud máxima (el número máximo de no terminales del rhs) de una producción de la gramática $NC(m)$.

El número de ocurrencias de atributos en cada producción se incrementa en forma similar.

3.3.4. La familia $NC(\infty)$

Existen gramáticas de atributos que no pertenecen a la familia $NC(m)$ para ningún m. Sucede cuando la gramática tiene producciones que son mutuamente recursivas. Un árbol sintáctico podría contener derivaciones de la forma $X \rightarrow Y \rightarrow X$ para cualquier número de repeticiones y si existiera un ciclo en las dependencias no podría determinarse estáticamente para ningún número finito de descendientes m. El hecho que una gramática no sea $NC(m)$ no significa necesariamente que sea circular, como se muestra en la figura siguiente (la cual es la gramática de la figura 3.8 con dos producciones adicionales: $P_5 : Y \rightarrow O$ y $P_6 : Q \rightarrow Y$).

Un evaluador para las GA $NC(\infty)$ necesita tener en cuenta un número *potencialmente (no realmente)* infinito de descendientes. En términos de un árbol sintáctico, lo anterior significa que el evaluador necesitará "mirar" hasta las hojas para seleccionar un plan para una instancia de una producción.

Para realizar la selección, el evaluador necesitará realizar un recorrido ascendente del árbol sintáctico para *marcar* los nodos con la información sobre las dependencias *actuales* transitivas en base al subárbol.

A continuación de caracterizará la familia $NC(\infty)$, para lo cual es necesario definir algunos conceptos.

Definición 3.3.8 *Sea T_X un subárbol cuya raíz es una instancia del símbolo X, el conjunto de dependencias transitivas entre los atributos de X en T_X se define como:*

$\delta = \{(X.a, X.b) \mid X.b \text{ depende}^{+} \text{ de } X.a \text{ en } T_X\}$

Definición 3.3.9 *Con cada no terminal X se asocia un conjunto $\Delta(X)$ de todos los posibles grafos de dependencias (teniendo en cuenta diferentes contextos inferiores) entre sus atributos:*

Sean $\delta_1, \delta_2, \ldots, \delta_k$ grafos de dependencias para todos los posibles subárboles cuya raíz es una instancia del símbolo X,

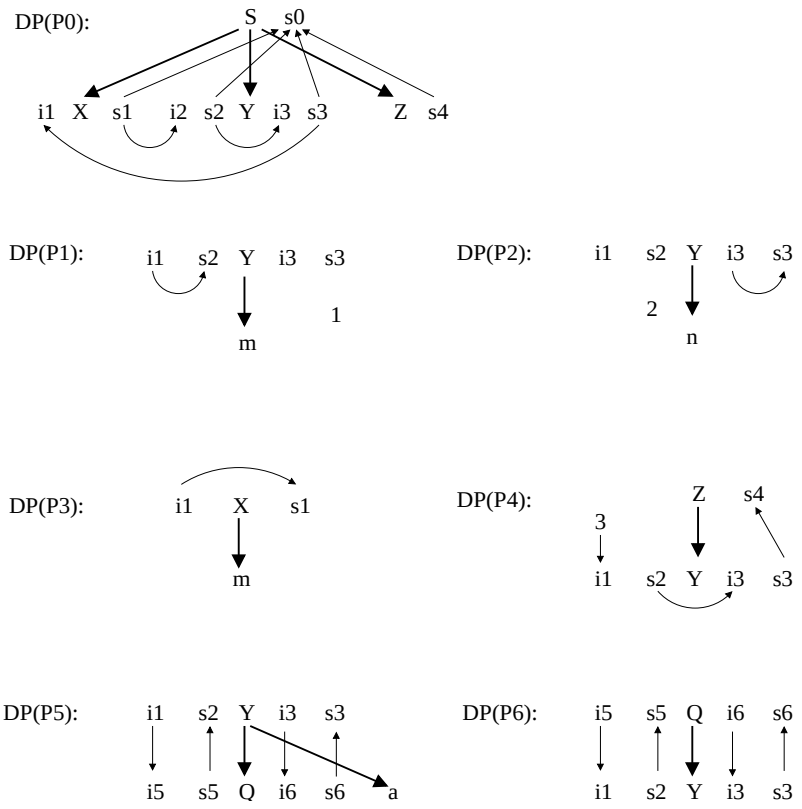

Figura 3.8: Grafos DP de una gramática $NC(\infty)$

$$\Delta(X) = \bigcup_{i=1}^{k} \delta_i$$

Teorema 3.3.3 *Los conjuntos de grafos de dependencias $\Delta(X)$ son finitos.*

Demostración: Dado que $\#A(X)$ es finito, el número total de posibles relaciones diferentes es finito. □

Para poder computar los posibles ordenes de evaluación para una producción, es necesario definir un grafo de dependencias para lo cual son necesarias las siguientes definiciones.

Definición 3.3.10 *Sea la producción $q : X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \dots X_k \alpha_k$ y sea $\delta_i \in \Delta(X_i)$, para $i = 1, \dots, k$, el grafo de (look-ahead) dependencias de la producción q con respecto a $\delta_1 \delta_2 \dots \delta_k$*

$$LDP(q|\delta_1, \delta_2, \dots, \delta_k) = DP(q) \bigcup_{j=1}^{k} \delta_j$$

Similarmente, se define el conjunto de todos los conjuntos de dependencias de la producción q como

$$SLDP(q) = \{LDP(q \mid \delta_1, \delta_2, \ldots, \delta_k) \mid \delta_i \in \Delta(X_i), i = 1, 2, \ldots, k\}$$

Definición 3.3.11 *Una GA es (o pernecece a la familia) $NC(\infty)$ si y sólo si cada grafo de $SLDP(p)$ es acíclico, para cada producción p.*

Es posible calcular estáticamente todos los posibles contextos inferiores de un símbolo analizando sucesivamente las producciones y ecuaciones semánticas de la GA.

En el próximo capítulo se presentará un algoritmo para computar los conjuntos de grafos $\Delta(X)$ (para cada X), se hará un análisis de su complejidad y se presentará un algoritmo para la generación de planes de evaluación.

Teorema 3.3.4 *La familia $NC(\infty)$ se corresponde con la familia de gramáticas de atributos bien definidas (WDAGs).*

Demostración: ($G \notin WDAG \Rightarrow G \notin NC(\infty)$):

Si G es una GA circular ($GA \notin WDAG$), entonces existe un árbol sintáctico T que contiene al menos una dependencia circular. Sea $q : X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$ la instancia de la producción aplicada en T que involucra las dependencias circulares y es la mas cercana a la raíz de T. Sea T_i el subárbol cuya raíz es X_i, $(i = 1, 2, \ldots, k)$. Sean δ_i, $(i = 1, 2, \ldots, k)$ las dependencias transitivas de los atributos de los símbolos X_i, respectivamente. Claramente, $\delta_i \in \Delta(X_i)$, por la definición 3.3.9.

Por lo tanto $DP(q) \bigcup_{i=1}^{k} \delta_i$ está contenido en $SLDP(q)$, el cual deberá ser circular, por lo tanto $GA \notin NC(\infty)$.

Conversamente, si se asume que $G \in NC(\infty)$. Sea T cualquier árbol sintáctico derivado de G y sea $q : X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$ cualquier instancia de una producción en T. Sea T_i un subárbol cuya raíz es X_i y δ_i contiene las dependencias transitivas de las instancias de los atributos de X_i $(i = 1, 2, \ldots, k)$. Nuevamente por la definición 3.3.9, $\delta_i \in \Delta(X_i)$, $DP(q) \bigcup_{i=1}^{k} \delta_i$ está contenido en $SLDP(q)$ y es acíclico, lo que implica que no pueden existir depedencias circulares en ningún árbol sintáctico derivado de G. Así G es una GA bien definida. □

3.4. Clasificación jerárquica de gramáticas de atributos

A continuación se presenta un teorema que describe la relación entre las familias de GA presentadas en esta seción, en base a su poder expresivo. Las familias por debajo de la línea divisoria se corresponden con estrategias de evaluación preestablecidas.

Es fácil ver desde sus correspondientes definiciones que $IDP - ANCAG(p) \subseteq EDP(p)$, para cada producción $p \in P$ de una GA, ya que cada grafo $IDP-ANCAG(p)$ está definido en base a las dependencias directas de los atributos de los símbolos que ocurren en p mas las dependencias (transitivas) de las producciones que tienen a esos símbolos como parte izquierda, no en cualquier parte como en el caso de los grafos $EDP(p)$.

Por lo tanto, los grafos de la familia $ANCAG$ nunca contendrán arcos no contenidos en los grafos correspondendientes de la familia $EOAG$, lo que implica claramente

AG	Gramáticas de atributos
WDAG	Gramáticas de atributos bien definidas
ANCAG	GA absolutamente no circulares
EOAG	GA Ordenadas Extendidas
OAG	GA Ordenadas
m-APAG	GA evaluables en m pasadas alternantes
n-PAG	GA evaluables en n pasadas
L-AG	GA l-atribuidas
S-AG	GA s-atribuidas

Cuadro 3.1: Jerarquía de GA

la inclusión de las familias, ya que si una GA es $ANCAG$ también es $EOAG$.

La inclusión de las familias definidas en base a una estrategia de evaluación es obvia. Sólo cabe aclarar que una $L-AG$ es una $n-PAG$ con $n=1$ con una pasada descendente de izquierda a derecha.

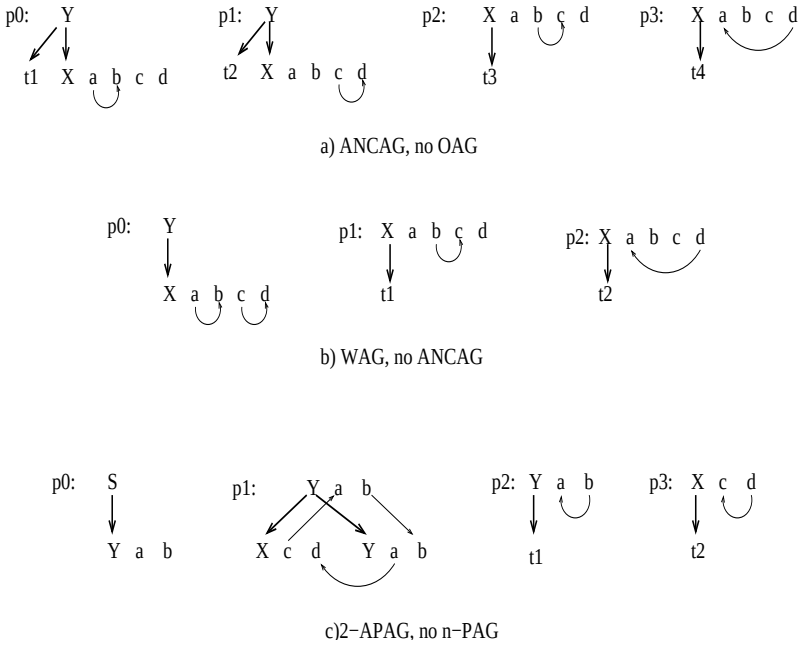

Figura 3.9: Ejemplos de GAs de diferentes familias

La figura 3.4 muestra algunos ejemplos de GA que pertencen a una familia y no a una familia inferior en la jerarquía.

Ya se mostró en la sección 3.2.3 que la familia $EOAG$ extiende a las OAG. También en la sección 3.3 se relacionó la jerarquía NC con las familias mencionadas en esta clasificación.

El próximo capítulo describe métodos de evaluación y de generación de evaluadores

para las familias de GA descriptas en este capítulo. En particular se hace incapié la evaluación de atributos por medio de *secuencias de visita*, método que se utilizará para realizar el diseño de un generador de evaluadores de GA descripto en el capítulo 7.

Capítulo 4

Métodos de evaluación

En este capítulo se analizan algunos métodos de evaluación de GA's. El problema de diseñar e implementar un evaluador para GAs consiste en realizar la evaluación de los atributos en un orden que sea consistente con sus dependencias.

El primer problema a resolver es la verificación que la GA sea evaluable, es decir, que no contenga dependencias circulares.

Como ya se ha visto en los capítulos anteriores, el *test de circularidad* es un problema intrínsecamente exponencial, lo que ha motivado la caracterización de algunas subfamilias analizadas en el capítulo anterior que permitan detectar circularidades en tiempo polinomial (en base a imponer restricciones en las dependencias).

En primer lugar se analiza la evaluación de atributos durante (en forma simultánea con) el parsing.

Luego se estudian métodos para realizar la evaluación dinámicamente, es decir, determinando el orden de evaluación en tiempo de ejecución.

Posteriormente se presenta un evaluador para $WAG's$ con una estrategia bajo demanda, que se basa solamente en el conocimiento de las dependencias directas entre las ocurrencias de los atributos en cada equación.

Luego se analizan métodos de evaluación estática. Este enfoque requiere que se computen los ordenes[1] de evaluación en tiempo de construcción del evaluador. En particular se analizará una técnica de evaluación basado en *secuencias de visita* introducido por Kastens en [23] y se analiza su aplicación a algunas familias analizadas en el capítulo anterior.

En el próximo capítulo se presentan algunos métodos de evaluación concurrente.

4.1. Evaluación durante el parsing

La idea de ir realizando la evaluación de los atributos durante el parsing es muy atractiva ya que es eficiente tanto en tiempo como en espacio (no se requiere almacenar el árbol sintáctico, el menos en forma completa).

El problema es que un parser generalmente construye (parcialmente) el árbol sintáctico o en forma descendente (top-down parsing) o ascendente (bottom-up parsing)[2] y la evaluación de ciertas clases de gramáticas de atributos puede requerir que ciertos nodos del árbol deba ser visitado varias veces, lo cual generalmente no es compatible

[1] También denominados *planes* de evaluación.

[2] Se asume que el parsing analiza la cadena de izquierda a derecha en una sola pasada.

con el parsing.

Una dificultad de la evaluación durante el parsing surge cuando se requiere computar una instancia de un atributo a en un nodo n_i que depende de una instancia b que se encuentra en un nodo n_j que aún no ha sido creado por no haberse realizado aún el análisis sintáctico de la producción correspondiente.

Como se vio en el capítulo anterior, una *L-AG* permite que sea evaluada durante el parsing en un parser descendente, ya que éste construye el árbol sintáctico desde la raíz y de izquierda a derecha, lo que es consistente con las dependencias de una *L-AG*.

Con un parser ascendente es un poco mas complicado, ya que la información necesaria (atributos heredados) para evaluar una *L-AG* no se encuentra disponible, aunque es fácil emular el parsing *LL* durante el parsing LR, como se verá en la sección 4.1.2.

Para evaluar durante el parsing la famila *WAG*, el evaluador debería computar los atributos que puedan ser evaluados durante un paso de parsing y para los demás atributos se podrían computar *aproximaciones* de las instancias de los atributos que residan en nodos aún no construidos, como lo proponen Noll y Vogler en [33].

La técnica utilizada por Noll y Vogler realiza la evaluación durante el parsing de aquellas instancias de los atributos que son consistentes con la construcción del árbol sintáctico y aquellas instancias que dependan de instancias aún no evaluadas son reemplazadas con *aproximaciones de términos*[3]y cada vez que se evalúa un atributo se recomputan las aproximaciones dependientes.

La desventaja es que si bien no es necesario mantener el árbol sintáctico completo, se deben mantener las aproximaciones y por cada nuevo nodo creado es necesario refinar las aproximaciones de los nodos vecinos, lo que en la práctica consiste en cambios de punteros en las representaciones de los términos.

A continuación se presentan dos técnicas para realizar la evaluación para gramáticas $LL(1)$[4] y para las gramáticas $LR(1)$[5]. Cabe aclarar que la mayoría de las herramientas de generadores de parsers generan uno de estos dos tipos mencionados.

4.1.1. Evaluación durante el parsing descendente (*LL(1)*)

En esta sección se presenta la construcción de un evaluador de una pasada determinístico para la familia *LL-AG*, es decir las GA *L-AG* cuya gramática subyacente son *LL(1)*.

Un parser clásico *LL(1)* utiliza una tabla de parsing y una pila para almacenar prefijos de formas sentenciales izquierdas[6].

Otra implementación posible es por medio de procedimientos recursivos. Por cada no terminal existe un procedimiento correspondiente encargado del parsing de la subgramática que éste produce. El cuerpo de los procedimientos reconocen los terminales de la producción e invoca al procedimiento correspondiente por cada no terminal. Es fácil de extender el parser para la evaluación de atributos, donde cada procedimiento

[3]Un término se puede representar con un grafo de punteros a nodos que contienen los valores de las instancias correspondientes.

[4]Una subfamilia de gramáticas libres de contexto que pueden reconocerse con un parser descendente determinístico.

[5]Subfamilia de las CFG que puede reconocerse con un parser ascendente determinístico.

[6]Cadenas de la forma $\alpha X_1 \dots X_n$, donde cada $\alpha \in T^*$ y $X_i (i = 1 \dots n) \in N$.

correspondiente al símbolo X tiene los atributos heredados de X como parámetros de entrada y los atributos sintetizados de X como parámetros de salida.

A continuación se presenta un evaluador basado en una implementación del método descendente recursivo utilizando una pila, adeacuada para la evaluación de atributos, tal como se presenta en [2] por Rieks op den Akker, B. Melichar y J. Tarhio.

```
var stack:(STATE × I(A) × S(A)) ;
stack := ([S → .α], ∅, ∅)
t := next_token(input)
repeat
  act := action(top(stack).state, t)
  if act = shift st then
    top(stack).state := st
    t := next_token(input)
  else if act = expand to A → γ then
    ia := eval(I(A),p), where p : X → αAβ ∈ P and top(stack).state = [X → α.Aβ]
    push([A → .γ], ia, ∅)
  else if act = reduce by A → γ then
    sa := eval(S(A), p), where p : A → γ ∈ P
    s = goto(top(stack).state,t)
    ia = top(stack).inh ; sa_γ = top(stack).syn
    pop(stack)
    top(stack) := (s, ia, sa ∪ sa_γ)
  end if
until act = accept or act = error
```

Algoritmo 2: Evaluador de $L - AG$ durante el parsing LL

El algoritmo de parsing se basa en la construcción de items LR(0), de la forma $[A \rightarrow \alpha.\beta]$, donde $A \rightarrow \alpha\beta$ es una producción de la gramática.

La pila de parsing contiene los *estados de producciones* de la forma $([A \rightarrow \alpha.\beta], i, s)$, donde la primer componente (estado o item LR(0)) describe la parte ya reconocida o *expandida* de la producción (α) y la parte aún por reconocer (β). En un estado de la forma $([A \rightarrow \alpha.X\beta])$, el elemento i representa el conjunto de valores de las instancias de los atributos heredados de X (X es un no-terminal) y s representa el conjunto de instancias de atributos sintetizados que ocurren en α.

La tabla de parsing es una matriz cuyas filas corresponden a estados y las columnas a Σ (conjunto de símbolos terminales). En el algoritmo de parsing de la figura 2 consulta la tabla de parsing mediante la función *action(state,token)*. Cada elemento de la matriz contiene una de las tres operaciones que se describen a continuación:

1. **shift** $s = [A \rightarrow \alpha t.\beta]$**:** dado el estado $[A \rightarrow \alpha.t\beta]$, $(t \in V)$, y si el caracter corriente de la entrada coincide con el símbolo terminal t, el estado del tope de la pila se reemplaza por el estado s.

2. **Expand to** $s = [X \rightarrow .\gamma]$**:** dado el estado $[A \rightarrow \alpha.X\beta]$, $(X \in N)$, en el tope de la pila, se apila el estado s. En este momento se evalúan los atributos heredados de X.

3. **Reduce by** $s = [A \rightarrow \alpha]$**:** dado un estado s de la forma $[A \rightarrow \alpha.]$ en el tope de la pila, este se reemplaza por el estado dado por la función $goto(s, t)$, donde t es el

terminal corriente en la cadena de entrada. Se evalúan los atributos sintetizados de A.

El algoritmo 2 muestra un evaluador basado en un parser *LL(1)*. Cada elemento de la pila es una tupla $(state, inh, syn)$ para mantener el estado y los conjuntos de valores de las instancias de atributos heredados y sintetizados de los símbolo correspondientes. La tabla de parsing está implementada en la función $action(state, token)$, la cual generalmente se representa como una matriz.

---

p_0: $S' \rightarrow S\ \$$
attribution
$S.h = 1$
end

p_1: $S \rightarrow (S)$
attribution
$S[0].s = S[1].s \times S[0].h$
$S[1].h = S[0].h + 1$
end

p_2: $S \rightarrow number$
attribution
$S.s = number.value$
end

Figura 4.1: Una GA LL(1).

La evaluación de los atributos toma lugar en cada operación *expand to* $A \rightarrow \alpha.X\beta$, en la cual se evalúan las instancias de los atributos heredados de X (y se almacenan en el tope del stack) y en la operación *reduce by* $A \rightarrow \alpha.$ donde se evalúan las instancias de los atributos sintetizados que ocurren en α (y se almacenan en el estado del tope del stack).

Estados	(	)	**number**	$
$s_1 = [S' \rightarrow .S\$]$	exp s_2	–	exp s_3	–
$s_2 = [S \rightarrow .(S)]$	shf s_4	–	–	–
$s_3 = [S \rightarrow .number]$	–	–	shf s_5	–
$s_4 = [S \rightarrow (.S)]$	exp s_2	–	exp s_3	–
$s_5 = [S \rightarrow number.]$	–	red 2, goto s_6	–	red 2, goto s_7
$s_6 = [S \rightarrow (S.)]$	–	shf s_8	–	–
$s_7 = [S' \rightarrow S.\$]$	–	–	–	accept
$s_8 = [S \rightarrow (S).]$	–	red 1, goto s_8	–	red 1, goto s_7

Figura 4.2: Tabla de parsing LL(1) para la GA 4.1

La función $goto(state, terminal)$ define las transiciones a nuevos estados en cada reducción, esto es, $goto([X \rightarrow \alpha.], t) = [Y \rightarrow .\beta]$ donde $t \in SD(Y)$.

Para mayor información sobre el cómputo de los *símbolos directrices* (SD) de un símbolo, ver [19].

La construcción de las tablas de parsing y de la función *goto* van mas allá del alcance de este trabajo. Para detalles sobre técnicas de parsing ver [6].

La AG de la figura 4.1, se usará como ejemplo para mostrar los pasos de evaluación de una LL-AG durante el parsing LL-1.

Stack *(tope a la izquierda)*	**Acción**
$[(s_1, \emptyset, \emptyset)]$	$expand\ s_2$
$[(s_2, \{S.h = 1\}, \emptyset), (s_1, \emptyset, \emptyset)]$	$shift\ s_4$
$[(s_4, \{S.h = 1\}, \emptyset), (s_1, \emptyset, \emptyset)]$	$expand\ s_3$
$[(s_3, \{S.h = 2\}, \emptyset), (s_4, \{S.h = 1\}, \emptyset), (s_1, \emptyset, \emptyset)]$	$shift\ s_5$
$[(s_5, \{S.h = 2\}, \emptyset), (s_4, \{S.h = 1\}, \emptyset), (s_1, \emptyset, \emptyset)]$	$reduce\ 2, goto\ s_6$
$[(s_6, \{S.h = 2\}, \{S.s = 5\}), (s_1, \emptyset, \emptyset)]$	$shift\ s_8$
$[(s_8, \{S.h = 2\}, \{S.s = 5\}), (s_1, \emptyset, \emptyset)]$	$reduce\ 1, goto\ s_7$
$[(s_7, \{S.h = 2\}, \{S_0.s = 10, S_1.s = 5\})]$	$accept$

Figura 4.3: Pasos de ejecución del algoritmo 2 para la entrada "(5)$".

La tabla de parsing $LL(1)$, correspondiente a la gramática de la figura 4.1 se muestra en la tabla 4.2. La tabla 4.3 muestra la secuencia de pasos del parser (y evaluador) para la entrada "(5)$".

A menudo se define un parser $LL(k)$ por medio de un conjunto de procedimientos recursivos. Cada procedimiento P_X es responsable del parsing de una producción de la forma $X \rightarrow \alpha \mid \beta \mid \ldots$.

El cuerpo de cada procedimiento P_X reconoce las cadenas que corresponden a terminales y el reconocimiento de lo generado por un no-terminal se invoca al procedimiento correspondiente.

Para realizar la evaluación de atributos durante el parsing descendente recursivo, cada procedimiento P_X toma como argumentos los atributos heredados correspondientes al símbolo X y retorna los valores de los atributos sintetizados de X, ya sea como parámetros de salida o los procedimientos se transforman en funciones. Los atributos heredados de los símbolos de la parte derecha se computan antes de la invocación al procedimiento (o función) correspondiente y son pasados como parámetros (por valor). Los atributos sintetizados se pueden computar inmediatamente antes del retorno. Para mayor información sobre parsing $LL(k)$ y evaluación de atributos, ver [2] y [6].

Un algoritmo similar se presenta en [3], el cual no utiliza items LR(0), sino que se basa en la emulación de un parser ascendente.

4.1.2. Evaluación durante el parsing ascendente (*LR(1)*)

Un parser *LR* utiliza un algoritmo *shift-reduce* para emular la construcción del árbol sintáctico desde las hojas hasta la raíz. La idea básica es ir apilando símbolos hasta que en la pila se detecte una parte derecha de una producción, para luego reducirla o reemplazarla con el símbolo no terminal de su parte izquierda. Mas detalladamente, el algoritmo de parsing *LR* se basa en la propiedad que los prefijos viables (cadenas que pueden aparecer en la pila durante el parsing de una sentencia), definen un conjunto

regular. Por esta propiedad se puede definir un autómata finito determinístico M tal que el conjunto de prefijos viables sea $\mathcal{L}(M)$.

A partir de él se puede implementar un algoritmo de parsing dirigido por M, cuyas trazas siguen las secuencias de transiciones de M para aceptar cada prefijo viable.

```
stack := (S, ∅, ∅)
t := next_token(input)
repeat
  state := top(stack).state
  act := action(state, t)
  if act = Shift a then
    push((t,∅, ∅), stack)
    t := next_token(input)
  else if act = Reduce by p : A → α then
    sa := eval(S(A), p)
    pop | α | symbols of stack
    s:= goto(state, A)
    push((s,∅, ∅), stack)
    top(stack).inh := eval(I(B), q) where q : X → γBβ y s = [X → γ.Bβ]
    top(stack).syn := sa
  end if
until act = error or act = accept
```

Algoritmo 3: Evaluador de $L-AG$ durante el parsing LR

Como ya se ha mencionado, es natural evaluar una *S-AG* en un parser ascendente, ya que la construcción (implícita) del árbol es compatible con las dependencias. Sin embargo, la evaluación de una $L-AG$ también es posible durante el parsing ascendente si se emula el comportamiento de un parser LL.

Dada una gramática $LL(1)$, si se introducen símbolos no terminales adicionales (los cuales producen la cadena vacía) al comienzo de la parte derecha de cada producción, esos no terminales adicionados, en un parser LR, se reconocerán en el mismo orden en el que las producciones son aplicadas durante el parsing LL. Además esta transformación no introduce conflictos LR.

De esta forma es posible evaluar durante el parsing LR cada gramática de atributos que pueda ser evaluada durante el parsing LL. Además, como las gramáticas LR incluyen (propiamente) a las LL, es posible evaluar una familia mayor de GA durante el parsing LR.

Un parser LR (shift-reduce) está basado en una *pila de parsing* cuyo cada elemento es una tupla $(state, i, s)$ como en la sección anterior. El control está dirigido por la *tabla de parsing* (representada por las funciones *action(state,token)*) ($token \in T$) y *goto(state,symbol)* ($symbol \in N$).

El algoritmo 3 muestra un evaluador basado en un parser LR.

En cada operación *reduce by* $A \to \alpha$ se evalúan los atributos sintetizados del símbolo A y los heredados de los símbolos que pueden seguir a A en una forma sentencial izquierda.

Si los símbolos terminales (tokens) contienen atributos sintetizados, éstos se deberían evaluar en cada operación **shift** y se deberían almacenar en el stack.

4.2. Evaluadores dinámicos

El primer enfoque para la evaluación de GA surge naturalmente de la idea de realizar la evaluación basándose en un *orden topológico* del grafo de dependencias [26].

El grafo de dependencias se puede construir sobre un árbol atribuido en base a las dependencias directas de las producciones.

Un evaluador dinámico tiene como ventajas su simplicidad y que es posible evaluar cualquier $WDAG$ (GA bien definida) o aún GA's irrestrictas utilizando evaluación lazy (siempre y cuando exista el mínimo punto fijo en el álgebra de términos denotado por las ecuaciones de atribución) [33]. Además se pueden detectar ciclos en el grafo de dependencias antes o durante la evaluación.

Las principales desventajas son que generalmente es necesario mantener el árbol sintáctico y el grafo de dependencias. La construcción del grafo de dependencias consume tiempo y memoria. Para una GA del tamaño comúnmente usado en la práctica los requerimientos de memoria pueden ser considerables.

El tiempo de procesamiento insumido en la construcción del grafo de dependencias puede ser mayor que el proceso de evaluación en sí mismo.

Los evaluadores dinámicos no han tenido mucho interés en el desarrollo de herramientas de generación de procesadores de lenguajes como por ejemplo los compiladores, porque uno de los principales requisitos de un compilador es que sea eficiente, ya que durante un desarrollo un programador generalmente necesita recompilar un número muy importante de veces hasta obtener una versión final del programa requerido.

Razón por la cual en el presente trabajo se pone énfasis en métodos de evaluación estática, y no se profundizará en los métodos dinámicos.

4.3. Evaluación bajo demanda

Otra forma de evaluar los atributos en el árbol atribuido es hacerlo en un esquema bajo demanda. Este enfoque permite que no sea necesaria la construcción del grafo de dependencias, sino que por cada instancia de cada atributo[7] se incluye un *flag* (boolean) que permite determinar si el atributo fue definido (evaluado) o no.

Una instancia de un atributo se puede evaluar si todas sus dependencias directas están evaluadas. Una vez definido el valor de un atributo se activa el *flag*.

El algoritmo de evaluación puede usar diferentes estrategias para realizar los recorridos del árbol sintáctico.

En esta sección se presenta un algoritmo de evaluación para GA bien definidas (WAG), desarrollado en el marco de esta tesis, con detección de circularidades. Se describen las estructuras de datos utilizadas y los detalles del funcionamiento del algoritmo, el cual implementa un autómata pila.

El algoritmo *DemandWAG* (algoritmo 4) utiliza una pila de pares de la forma (n, e) tal que n es el nodo que es una instancia de la aplicación de una producción p y e es una equación que define una instancia de un atributo en p.

La idea consiste básicamente en construir un evaluador iterativo que computa los atributos por necesidad. El algoritmo es en teoría óptimo con respecto a tiempo de

[7] En cada atributo de cada nodo el árbol sintáctico.

ejecución[8].

Una desventaja es que el tamaño del stack utilizado no está limitado por la altura del árbol sintáctico atribuido, sino por el camino mas largo en las cadenas de dependencias entre los atributos. Cada elemento del stack es muy compacto con lo que se logra una mejora en los requerimientos de memoria comparado con otros algoritmos de evaluación bajo demanda, como por ejemplo el algoritmo propuesto por Grosch en [17], el cual está basado en este enfoque pero utilizando procedimientos recursivos.

Los valores de las instancias de los atributos se almacenan en los nodos correspondientes a los símbolos de la gramática.

Cada nodo del árbol contiene los siguientes campos:

1. el número de regla (producción) al que corresponde el nodo[9].
2. un vector de punteros a sus nodos hijos.
3. un puntero a su nodo padre.
4. un campo por cada atributo del símbolo correspondiente al nodo.
5. un vector de bits *COMPUTED*, en el cual se mantienen marcados las instancias de atributos ya computados del nodo.

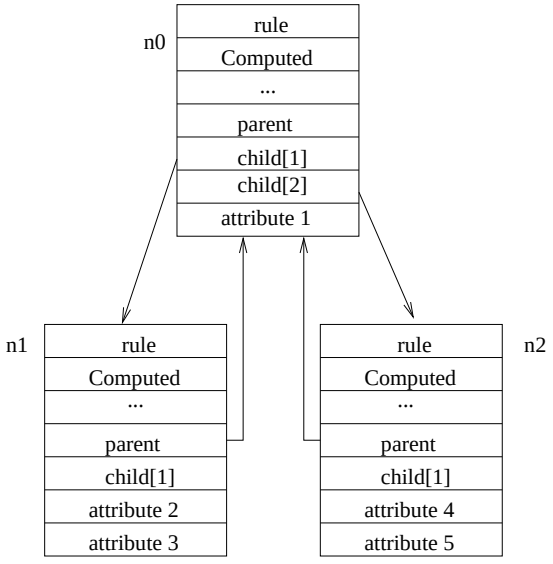

Figura 4.4: Estructura del árbol sintáctico

La figura 4.4 muestra una instancia de un árbol sintáctico para una producción de la forma $X \rightarrow YZ$ donde X tiene 1 atributo e Y y Z tienen dos.

[8]Ya que sólo computa los atributos que dependen de los atributos de la raíz y en cada iteración se computa el valor de una instancia de un atributo.

[9]Los símbolos terminales no se almacenan en el árbol.

```
Procedure wageval( t:TREE )
s := new stack()
push_root_attributes(s,t)
while not empty(s) do
  (t, eq) := top(stack)
  do := lhs(eq)
  if computed(rhs(eq)) then
    compute(t,eq)
    mark_as_computed(container_node(t,do), do)
    pop(stack)
  else
    for all a_i in no_computed(rhs(eq)) do
      dc := defining_context(t,a_i)
      de := defining_equation(dc,a_i)
      if (dc, de) in stack then
        error("Circular AG")
      else
        push(stack, (dc,de))
      end if
    end for
  end if
end while
```

Algoritmo 4: DemandWAG: Algoritmo de evaluación de WAG's por necesidad

El algoritmo 4 sólo necesita conocer las dependencias directas entre los atributos de cada producción y utiliza una pila que contiene información sobre los atributos a evaluar[10].

Sea $rule(t) = r : X_0 \rightarrow X_1 \dots X_k$, la regla aplicada correspondiente al nodo t, la función $container\_node(t, at)$ retorna el nodo que contiene la instancia del atributo at, la cual puede definirse como:

$$container\_node(t, at) = \begin{cases} t & \text{if } at \in A(X_0) \\ child_i(t) & \text{if } at \in A(X_i),\ (1 \leq i \leq k) \end{cases}$$

La función $defining\_context(t, at)$ retorna el nodo (contexto) que es una instancia de la producción en que se define el atributo at, definida como:

$$defining\_context(t, at) = \begin{cases} parent(t) & \text{if } at \in I(X_0) \\ t & \text{if } at \in (S(X_0) \cup I(X_i)),\ (1 \leq i \leq k) \\ child_i(t) & \text{if } at \in I(X_i),\ (1 \leq i \leq k) \end{cases}$$

La rutina $compute(t, eq)$ ejecuta la evaluación de la equación eq en la regla aplicada en t. La implementación de esta rutina puede hacerse de la forma

[10] En realidad se apilan pares que contienen el número de ecuación que define a la instancia del atributo a evaluar y el nodo que será su contexto de evaluación.

```
compute(t,eq)
{
  switch(t->rule) {
    case r1:
             eval_rule_r1(t,eq);
             break;
    case r2:
             eval_rule_r2(t,eq);
             break;
    ...
  }
}
```

Cada rutina de evaluación de las reglas de atribución asociadas a la producción r_i se pueden implementar como:

```
eval_rule_ri(treenode,rule)
{
  switch(rule) {
    case 1: ... /* code for evaluation of equation 1 of production ri */
            break;
    case 2: ...
  }
}
```

Esto puede hacerse de manera mas elegante si se implementa en un lenguaje orientado a objetos, en el cual cada producción tendría una correspondiente clase con un método eval(eq).

Los atributos podrían ser directamente atributos de la clase. De esta manera cada objeto (instancia) sería una instancia de una producción y se correspondería a un nodo del árbol sintáctico decorado.

En lenguaje orientado a objetos, además se aprovecharía del polimorfismo y del binding dinámico para la invocación al método eval(eq) correspondiente al nodo corriente.

La rutina $mark\_computed(n, at)$ marca el atributo at como computado en el nodo n (set de un bit).

En cada iteración se toma, del tope del stack, un nodo t del árbol y la ecuación eq que define el atributo do. El atributo do (defined occurrency) es el atributo a evaluar y t es el nodo que es una instancia de una aplicación de una regla (producción) que define a do. El cuerpo del ciclo invoca a la función semántica f que define el valor de la instancia de do (por medio de $compute()$) si las instancias de las que do depende directamente ya se han computado. Posteriormente do se marca como computado y se elimina el elemento del tope del stack. En caso que alguna dependencia directa a_i de do en eq, no esté aún computada, se apila el par (dc, de) para su posterior evaluación, donde dc (defining context) es el nodo del árbol correspondiente a una instancia de la aplicación de una producción la que contiene la ecuación de (defining equation) que define (computa) a a_i.

El algoritmo 4 detecta posibles circularidades cuando una instancia de un atributo es demandada para su evaluación y ya se encontraba en la pila. Para evitar una

búsqueda completa en la pila, una implementación podría agregar otro vector de bits *DEMANDED* (un bit por cada instancia de los atributos) en cada nodo del árbol, el cual se debe activar en cada push. De esta manera la condición de la búsqueda en la pila se reduce a verificar en el nodo si el bit correspondiente a la instancia está activado.

Una de las principales ventajas de este algoritmo es su simplicidad, bajos requerimientos de memoria (sólo una pila de referencias) y puede aplicarse o extenderse a varios formalismos que extienden las GAs, como por ejemplo las GA condicionales y las GA de alto orden (HAGs o *High order Attribute Grammars*) que se describen en el capítulo 5.

Como desventaja, requiere mantener información en tiempo de ejecución sobre las dependencias directas de cada regla de atribución y la clase de los atributos (sintetizados o heredados).

Sin embargo, esta información se puede representar en forma muy compacta. Las dependencias directas de cada regla de atribución pueden representarse como un vector v de pares $(socc, at)$, donde cada par representa la ocurrencia del símbolo en la producción a la que pertenece y el atributo (su identificador).

Los elementos $v[i]$, con $1 \leq i \leq k$, representan las dependencias directas (u ocurrencias de uso) del elemento $v[0]$, el cual es la ocurrencia definida. Es necesario un vector por cada regla de atribución.

La información si un atributo es heredado un sintetizado se puede representar como un vector de bits, donde cada bit se corresponde con un atributo.

Se debe notar que las funciones *container_ node(t,at)* y *defining_ context(t,at)* pueden implementarse fácilmente a partir de esta información.

4.4. Evaluación estática

En esta sección se analizan métodos de evaluación en que el orden de evaluación se determina estáticamente, esto es, en tiempo de generación del evaluador.

Un generador de evaluadores de GA podría determinar el orden de evaluación de los atributos que se definen en cada producción para luego generar código que realize la evaluación en un orden consistente con las dependencias.

Algunas de las familias de GA estudiadas en el capítulo anterior han sido definidas con una estrategia de evaluación asociada. Así por ejemplo, vimos que una L-AG se puede evaluar en un recorrido ascendente de izquierda a derecha, lo que permite que se evalúe durante el parsing si este construye el árbol sintáctico de una manera consistente (ej: LR-parsing).

Como ya se ha analizado anteriormente, éstas familias están caracterizadas por la imposición de restricciones en las dependencias, quitándoles poder expresivo.

Para las familias $ANCAG$ e inferiores en la jerarquía dada en la sección 3.4 existe un único orden asociado a cada producción. Pero para las familias superiores pueden existir diferentes ordenes de evaluación[11] para cada producción, dependiendo del contexto de su aplicación.

Un plan de evaluación $plan(p)$ es una secuencia de las ocurrencias de atributos de p $(A(p))$.

[11] También denominados *planes de evaluación.*

En el caso de las GA $NC(k)$ un orden de evaluación dependerá del contexto inferior (hasta k niveles inferiores) y en el caso de las $NC(\infty)$ (o WDAG) puede depender de su contexto superior y de su contexto inferior (desde la raíz hasta las hojas del árbol sintáctico).

Solamente las familias $ANCAG$ o inferiores en la jerarquía mencionada permiten encontrar los ordenes de evaluación en tiempo polinomial. El resto requiere algoritmos de mayor complejidad computacional, como ya se vio en el capítulo anterior.

Un plan de evaluación para una producción puede obtenerse por la aplicación de un orden topológico sobre el (los) grafo(s) de dependencias inducidas de cada producción. En el caso de las familias superiores a las $ANCAG$, se requiere que para cada árbol sintáctico se seleccione (en cada nodo) uno de los posibles planes, en base a su contexto, como paso previo a la evaluación.

Una vez obtenidos los planes de evaluación para cada producción queda el problema de obtener un algoritmo eficiente que evalúe las instancias de los atributos dichos planes. A continuación se presenta un método propuesto inicialmente para la evaluación de las OAG's y que se puede extender a cualquier familia de GA no circular.

4.4.1. Evaluación de la famila ANCAG

En esta sección se describe la manera en que se puede evaluar una *Gramática de Atributos Absolutamente no circular*.

Como ya se ha visto en el capítulo 2, el único factor exponencial del *test de circularidad* de Knuth es el número de grafos construidos para cada símbolo de la gramática[12].

Para obtener un algoritmo de tiempo polinomial, se ha definido la familia $ANCAG$ en las cuales se construye un único grafo de dependencias entre los atributos de un símbolo[13], dando lugar a un único grafo de dependencias por producción (como se ha visto en el capítulo anterior, en la caracterización de ésta famila).

El algoritmo que computa los grafos $IDP_{ANCAG}(p)$ (derivado en forma obvia desde su definición) para cada producción $p \in P$, tiene complejidad polinomial en cuanto al tamaño de la gramática. Mas precisamente tiene complejidad $O(\mid V \mid \mid P \mid K^4X^5)$, donde V es el conjunto de símbolos no terminales, P es el conjunto de producciones, K es la longitud máxima de las producciones y X es el número máximo de atributos de los símbolos de la gramática.

Es posible generar un evaluador a partir de los grafos de dependencias inducidas de cada producción ($IDP_{ANCAG}(p)$) para generar la estrategia (recorrido o tree-walk) del árbol sintáctico.

A continuación se presenta un enfoque propuesto por Riis[35].

Para describir el método propuesto por Riis, es necesario definir la noción de *secuencia de compitación*.

Definición 4.4.1 *Una secuencia de computación es una evaluación de atributos mediante un recorrido del árbol atribuido que cumple con las siguientes condiciones:*

1. *cada instancia de un atributo en cada nodo del árbol se computa sólo una vez.*

[12] Llamados comúnmente *grafos-is*, ya que representan las dependencias desde los atributos heredados a los sintetizados, representados en la función $Delta(X)$, en el algoritmo 1.

[13] Se realiza la unión de todos los posibles contextos inferiores de un símbolo.

2. el orden de evaluación es compatible con las dependencias.

Sea T un árbol atribuido de una AG G. Sea N_0 un nodo y $p : X_{p0} \rightarrow X_{p1} \dots X_{pn}$ la producción aplicada en N_0. Sean $N_1 \dots N_n$ los hijos (de izquierda a derecha) de N_0 en T.

Un recorrido de T/N_0 es una secuencia de símbolos (N, A), donde N es un nodo de T/N_0 y A es un conjunto de atributos de N. La secuencia se define como:

1. La secuencia vacía es un recorrido de T/N_0.

2. Si s_i es un recorrido de $T/N_{0_{ji}}$, para $1 \leq i \leq r > 0$ y $1 \leq j \leq n_p$, $AI \subseteq I(X_{p0})$ y $AS \subseteq S(X_{p0})$, entonces $s = (N_0, AI)s_1 \dots s_r(N_0, AS)$ es un recorrido de T/N_0.

3. Si s y s' son recorridos de T/N_0, entonces también lo es la secuencia ss'.

Los pares (N_i, A) pueden denotarse como (i, A).

Las una secuencia $s(N_0, p)$ tiene la forma $(h_1, A_1) \dots (h_r, A_r)$, con $0 \leq h_i \leq n_p$ y $A_i \subseteq I(X_{p_{h}i} \cup S(X_{x_{p}i})$, para $1 \leq i \leq r$.

Se denotará como $s(N, p)(j)$ (o también $s(N)$) a la secuencia de subconjuntos restringida a los atributos $A(X_{pj})$ en el mismo orden que aparecen en la secuencia $s(N, p)$.

Para poder establecer un recorrido que satisfaga los requemientos mencionados arriba, se definen particiones ordenadas de los atributos de los símbolos no terminales de las producciones.

Definición 4.4.2 *Una partición de los atributos de un símbolo X es una secuencia finita no vacía phi $=< A_1, \dots, A_{2_m} >$, que satisface:*

1. $A_{2i-1} \subseteq I(X)$ y $A_{2i-1} \subseteq S(X)$, $(1 \leq i \leq m)$,

2. $A_1 \cup \dots \cup A_{2m} = I(X) \cup S(X)$,

3. $A_i \cap A_j = \emptyset$, $(i \neq j, 1 \leq i, j \leq 2m)$.

Definición 4.4.3 *Sea T un árbol atribuido de una AG G. Sea N_0 un nodo y $p : X_{p0} \rightarrow X_{p1} \dots X_{pn}$ la producción aplicada en N_0. Sean $N_1 \dots N_n$ los hijos (de izquierda a derecha) de N_0 en T. Un recorrido s de T/N_0 es una secuencia de computación si*

1. $s(N_0, p)$ es una partición de los atributos de p que satisface $DG(p)$, y

2. $s(T/N_j)$ es una secuencia de computaciones para T/N_j, $(1 \leq j \leq n_p)$.

Cada secuencia de computación $s(N, p)$ representa un orden de evaluación de los atributos del símbolo X_n en el nodo N. Se debe notar que cada secuencia en un nodo es independiente de las secuencias de los nodos hijos, ya que cada $DG(p)$ de la familia $ANCAG$ refleja las dependencias entre los atributos que ocurren en p independientemente de sus contextos inferiores.

A partir de las secuencias de computaciones obtenidas un evaluador puede generar *secuencias de acciones*, posteriormente llamadas *secuencias de visita* por U. Kastens[23], las cuales se describen en la próxima sección.

4.4.2. Secuencias de visita

En 1980, Uwe Kastens propone un método de evaluación para la familia OAG, denominado *secuencias de visita*[23]. En esta sección se analizará una generalización de secuencias de visita aplicable no sólo a las OAG sino también para WDAGs.

Sea n un nodo de un árbol atribuido T. Una secuencia de visitas en el nodo n es una secuencia de tres operaciones u acciones: *visit(child,i)*, *compute(at)* y *leave(i)*.

La acción *visit(child,i)* indica que el evaluador debe moverse (visitar) el nodo hijo *child* de n y corresponde a la *i-ésima* visita al nodo hijo.

La acción *compute(at)* indica que debe evaluarse la ecuación que define at en la producción p aplicada correspondiente al nodo n.

La acción *leave(i)* indica que ha finalizado la visita *i-ésima* en el nodo corriente y que se debe visitar al nodo padre[14].

Un evaluador basado en secuencias de visita, comienza ejecutando las operaciones en el nodo raíz del árbol. En cada visita a un nodo se ejecutan las operaciones en secuencia hasta la ejecución de una operación *leave(i)*. En su próxima visita $(i+1)$, se continúa con la operación que sigue a *leave(i)*.

Cada secuencia de visitas finaliza con una operación *leave*. La evaluación termina cuando se ejecutaron todas las operaciones del nodo raíz del árbol sintáctico.

Los evaluadores basados en secuencias de visita pertenecen a una familia denominada llamados *evaluadores multivisita*, ya que el proceso de evaluación puede requerir múltiples visitas a cada nodo.

Se pueden generar secuencias de visita a partir de los grafos de dependencias inducidas de cada producción. Cada subsecuencia de instancias de atributos heredados del símbolo de la parte izquierda de la producción se reemplazan por una operación *leave*, cada ocurrencias de un atributo definido en la producción (sintetizados del símbolo de la parte izquierda o heredados de un símbolo de la parte derecha) se reemplazan por la correspondiente operación *compute* y cada subsecuencia de instancias de atributos sintetizados de el símbolo X_i de la parte derecha de la producción se reemplaza por una operación *visit(i)*.

Las operaciones *leave* al comienzo de una secuencia de visitas se pueden eliminar.

4.4.2.1. Implementación de secuencias de visita

Existen diferentes formas de implementar las secuencias de visita. Una de las técnicas que mas se ha utilizado se implementa por medio de procedimientos recursivos. En este enfoque, una secuencia de visita VS_r, correspondiente a una producción de la forma $r = X_0 \rightarrow X_1 \dots X_k$, se divide en m partes, cada una finalizada por una operación *leave*, es decir

$$VS_r = VS_{r,1}, VS_{r,2}, \dots, VS_{r,m}$$

Por cada $VS_{r,i}$, $1 \leq i \leq m$, se construye un procedimiento r_i que toma como argumento el nodo del árbol sintáctico correspondiente a una instancia de aplicación de la producción r.

El cuerpo de r_i contiene una traducción directa de las secuencias de las operaciones de $VS_{r,i}$. Cada $compute(at)$ se reemplaza por un llamado a la función semántica

[14] En [23] se proponen solo dos operaciones, donde *visit(0,i)* corresponde a *leave(i)*.

correspondiente que define at en la producción r. Cada $visit(child, k)$ por una invocación a la subrutina q_k tal que q es la producción correspondiente al nodo hijo $child$.

Una visita a un ancestro al final de cada $VS_{r,i}$ se implementa por el retorno de la subrutina correspondiente.

```
eval_VS(n:TREE)
while ¬root(n) ∧ n.vs[n.vs_current] ≠ LEAVE do
  node ← n
  if node.vs[node.vs_current] = VISIT(i) then
    n ← node.child[i]
  else if node.vs[node.vs_current] = COMPUTE(j) then
    compute(node, j)
  else {operation LEAVE}
    n ← node.parent
  end if
  node.vs_current ← node.vs_current + 1
end while
```

Algoritmo 5: Intérprete para la evaluación de secuencias de visita

A continuación se describe otra posible implementación de un evaluador de secuencias de visita. El evaluador actúa como un intérprete de las acciones realizando la operación correspondiente.

El algoritmo 5 evalúa los atributos de un árbol sintáctico en el cual cada nodo n contiene la secuencia de visita[15] (vector vs) correspondiente a la regla $n.rule$.

El algoritmo es muy simple y tiene como ventaja que no demanda memoria como otros algoritmos propuestos, ya que es iterativo y sólo requiere mantener un índice en cada nodo del árbol para tener en cuenta la porción ya ejecutada de una secuencia de visita (variable $vs\_current$).

Las secuencias de visita pueden representarse como un vector (arreglo unidimensional) $vs[]$ de enteros, donde un valor de $v[i] = k$, con $k > 0$, representa una operación $visit(k)$ (visita al k-ésimo nodo hijo), un valor $k < 0$ representa la operación $compute(j)$ y un valor $k = 0$ representa una operación $leave$.

NCEval, una de las herramientas desarrolladas en el marco de esta tesis, utiliza un evaluador de secuencias de visita basado en esta última idea desarrollada por el autor de esta tesis.

4.4.3. Generación de evaluadores para GA bien definidas

Como se vio con anterioridad en una GA bien definida (o $NC(\infty)$), una producción puede tener asociada mas de un plan, dependiendo del contexto de su aplicación. Un evaluador para esta familia deberá realizar un previo de selección del plan correspondiente a cada nodo del árbol sintáctico.

[15] En una implementación real será una referencia (o puntero) a un vector.

Un evaluador para las GA $NC(\infty)$ necesita tener en cuenta un número *potencialmente (no actualmente)* infinito de descendientes. En términos de un árbol sintáctico, lo anterior significa que el evaluador necesitará "mirar" hasta las hojas para seleccionar un plan para una instancia de una producción.

Para realizar la selección el evaluador necesitará realizar un recorrido ascendente del árbol sintáctico para *marcar* los nodos con la información sobre las dependencias *actuales* transitivas en base al subárbol.

A cada no terminal X se le asocia un conjunto $\Delta(X)$ de posibles grafos de dependencias entre sus atributos ($\Delta(X)$ fue definido en la sección 3.3.4) del capítulo anterior. El algoritmo 6 computa todos los posibles grafos. Si algún $\delta \in \Delta(X)$ es cíclico, entonces la GA no es $NC(\infty)$.

Sea p una producción de la forma $p : X_0 \rightarrow X_1 \dots X_k$, el algoritmo *InfiniteLookAhead*, además, construye la función $\Lambda[q|\delta_1, \delta_2, \dots, \delta_k]$, la cual representa el grafo de dependencias entre los atributos de X_0 cuando se aplica la producción p y las dependencias transitivas de los símbolos X_i son δ_i, $(1 \leq i \leq k)$, respectivamente.

```
Δ(X) ← ∅, ∀X ∈ V
repeat
  change ← false
  for cada q: X0 → X1X2 ... Xk do
    G ← DP(q)
    for j = 1 to k do
      for cada δ' ∈ Δ(Xj) do
        G ← G ∪ δ'
      end for
    end for
    δ ← G||X0
    if δ es circular then error
    Λ[q|δ1, ..., δk] ← δ
    if δ ∉ Δ(X0) then
      changed ← true
      Δ(X0) ← Δ(X0) ∪ δ
    end if
  end for
until not changed
```

Algoritmo 6: InfiniteLookAhead

La información denotada por Λ será utilizada por el evaluador para la selección de los planes en cada nodo del árbol sintáctico. El algoritmo 6 es básicamente el *test de circularidad* de Knuth[26] con la adición del cómputo de la función Λ y los conjuntos $\Delta(X)$.

La complejidad del algoritmo es $O(|N||P|(h!)^{l+1}l^3h^3)$, donde h es el número máximo de atributos de los símbolos y l es el número máximo de no terminales de la parte derecha de las producciones. El análisis de complejidad se basa en que $|\Delta(X)| = O(h!)$. Dado que hay $|N|$ no terminales, la suma de todos los $|\Delta(X)|$ es $O(|N|h!)$ (el cual es el número de veces máximo que se ejecuta el ciclo *repeat*, ya que en cada iteración se debe agregar al menos un nuevo δ). la sentencia *for* más externa se ejecuta $|P|$ veces y el *for* más interno se ejecuta a lo sumo $(h!)^l$. La operación de unión $G \leftarrow DP(q) \bigcup_{i=1}^{k} \delta_i$

toma $O(l)$ $(k \leq l)$ y la operación de proyección ($\|_{X_0}$ tiene $O(h^3(l+1)^3)$, ya que se debe realizar la clausura transitiva.

Si bien el orden de complejidad es exponencial con respecto a l, éste valor generalmente no supera a 5 en la práctica (una GA con numerosos no terminales en la parte derecha de las producciones es muy inusual, ya que sería poco legible e indicaría que no se ha modularizado como corresponde). Si bien se pueden llegar a realizar un gran número de operaciones y las funciones computadas pueden tener un número considerable de entradas, el problema es perfectamente tratable con la potencia computacional que hoy brinda cualquier PC de escritorio.

incialmente Π indefinida
$\omega_0 \leftarrow$ cualquier orden de $A(S)$ (atributos de S)
$WL \leftarrow \{(S, \omega_0)\}$
repeat
  $(X_0, \omega) \leftarrow$ un elemento de WL
  $WL \leftarrow WL - \{(X_0, \omega)\}$
  sea $q : X_0 \rightarrow X_1 X_2 \ldots X_k$
  for cada $ADP(q|\delta_1, \ldots, \delta_k) \in SADP(q) \wedge app(q, \omega, \delta_1, \ldots, \delta_k)$ **do**
    $\psi \leftarrow TopologicalSort(ADP(q|\delta_1, \ldots, \delta_k) \cup \omega)$
    $\Gamma[q, \omega, \delta_1, \ldots, \delta_k] \leftarrow \psi$
    for cada $X_i \in rhs(q)$ **do**
      $\omega_i \leftarrow \psi\|_{A(X_i)}$
      $\Theta[q, \omega, i, \delta_1, \ldots, \delta_k] \leftarrow \omega_i$
      if $\Pi(X_i, \omega_i) = undefined$ **then**
        $\Pi(X_i, \omega_i) \leftarrow defined$
        $WL \leftarrow WL \cup \{(X_i, \omega_i)\}$
      end if
    end for
  end for
until $WL = \emptyset$

Algoritmo 7: Generación de planes de evaluación

Una vez que se aplica el algoritmo 6 y se ha detectado que la GA es $NC(\infty)$, se generan los planes de evaluación asociados a cada producción.

El algoritmo 7 realiza la generación de planes y es una modificación del algoritmo propuesto por Wuu Yang en [44].

Las funciones[16] computadas, Γ y Θ, serán utilizados por el evaluador para realizar la selección del plan correspondiente para cada producción, en cada nodo del árbol sintáctico.

Un valor de $\Gamma[q, \omega, \delta_1, \ldots, \delta_k]$ denota el grafo de dependencias de la producción q con un orden de evaluación de los atributos de X_0 (impuestos por el contexto superior) y con las dependencias de su contexto inferior $\delta_1, \ldots, \delta_k$.

La función $\Theta[q, \omega, i, \delta_1, \ldots, \delta_k]$ representa el orden de evaluación impuesto a los atributos del símbolo X_i (hijo i-ésimo en la producción q).

[16] Representadas como matrices.

Una de las diferencias con respecto al algoritmo presentado por Wuu Yang en [44] es la lista de trabajo WL, que contiene pares de símbolos y ordenes en lugar de sólo ordenes. Otra de las diferencias es el uso de un *filtro* para evitar la generación de planes espúreos que pudieran ser generados[17](aunque algunos de ellos posteriormente descartados) inútilmente por el algoritmo original. El filtro es la función (boolean) *app*, la cual se define como la expresión lógica:

$$app(q, \omega, \delta_1, \ldots, \delta_k) \triangleq \begin{array}{l} \Lambda[q|\delta_1, \ldots, \delta_k] = \delta'_i \wedge \exists p, \omega', i \mid \\ (defined(\Theta[p, \omega', i, \ldots, \delta'_i, \ldots]) \Rightarrow \Theta[p, \omega', i, \ldots, \delta'_i, \ldots] = \omega) \end{array}$$

El predicado $app(q, \omega, \delta_1, \ldots, \delta_k)$ establece que la producción q elejida es aplicable con orden de evaluación ω de los atributos del símbolo de la parte izquierda de q (contexto superior) y al contexto inferior $\delta_1, \ldots, \delta_k$, previniendo la computación de planes espúreos cuando se dispone de la información sobre el origen de ω.

El uso de la función *app* reduce significativamente el número de planes generados evitando la generación de planes espúreos. En [44], W. Yang plantea descartar los planes que contienen ciclos, ya que obviamente son espúreos (se debe recordar que en ésta etapa la AG obviamente no es circular), pero el método no detecta planes espúreos no circulares.

Las secuencias de visita pueden generarse como se vio anteriormente desde los planes de evaluación $\Gamma[q, \omega, \delta_1, \ldots, \delta_k]$.

```
procedure mark(n)
for i=1 to k do
  mark(n.child[i])
end for
n.mark ← Λ[n.rule|n.child[1].mark, ..., child[k].mark]

procedure select(n,ω)
n.plan ← Γ[n.rule, ω, m_1.mark, ..., m_k.mark]
for i=1 to k do
  select(n.child[i], Θ[n.rule, ω, i, n.child[1].mark, ..., n.child[k].mark])
end for

procedure eval(T)
mark(T)
select(T,ω_0) {ω_0 es el orden elejido de los atributos de S}
vs_eval(T)
```

Algoritmo 8: eval(T): el evaluador de atributos

El procedimiento *eval(T)* del algoritmo 8 evalúa los atributos de un árbol sintáctico T. La primer fase (*mark* y *select*) seleccionan el plan a utilizar en cada nodo del árbol utilizando la información computada estáticamente por los algoritmos enteriores y luego invoca al intérprete de secuencias de visitas.

[17]Un plan espúreo puede surgir por tener en cuenta ordenes que han surgido de diferentes contextos que el que se está teniendo en cuenta actualmente.

La selección de un plan requiere dos pasadas sobre el árbol sintáctico[18]. En la primer pasada (procedimiento *mark*) se realiza un recorrido ascendente para "marcar" o seleccionar los contextos inferiores de cada nodo.

En la segunda pasada (procedimiento *select*), se realiza un recorrido ascendente, donde se selecciona el plan correspondiente en base a los contextos superiores e inferiores.

Es posible realizar alguno de estos dos pasos (o ambos) durante la generación del árbol sintáctico (parsing).

Una vez que se ha seleccionado el plan correspondiente en cada nodo, se aplica un algoritmo de evaluación basado en secuencias de visita.

4.5. Evaluación de GA circulares

Si bien las GA tradicionales requieren que no sean circulares, es posible evaluar GA con dependencias circulares bajo ciertos condiciones.

Si la relación de dependencias define un reticulado finito y que funciones semánticas son monótonas, el problema se torna en un caso especial del problema de solución del valor X dada la ecuación $X = f(X)$.

Comenzando por un valor para X igual al ínfimo del reticulado, es posible obtener su valor final computando sucesivas aproximaciones por medio de proceso iterativo $X_{i+1} = f(x_i)$, el cual convergerá al menor punto fijo para el cual se satisfacerán todas las reglas semánticas.

Una GA de atributos circular que cumple con las condiciones enunciadas arriba se denominan *pseudo-circulares*.

El algoritmo 9 evalúa gramáticas de atributos con dependencias circulares. Los argumentos de la función semántica f_i son los valores de los atributos de los cuales depende x_i computados en la iteración anterior.

```
Initialize all attributes x_i with bottom values
repeat
  for each x_i do
    x_i = f_i(...)
  end for
until (none x_i changes)
```

Algoritmo 9: Evaluación de atributos con dependencias circulares.

En el próximo capítulo se describen algunas técnicas de evaluación concurrente.

[18] Es posible realizarlo en una pasada, pero es mejor analizarlo en dos pasadas por claridad.

Capítulo 5

Técnicas de evaluación concurrente

En este capítulo se analizarán algunas técnicas que han sido utilizadas para realizar la evaluación concurrente de una GA.

La idea de evaluación de GA en forma concurrente ha sido atractiva desde la aparición de los sistemas de multiprocesamiento y sistemas de tiempo compartido.

Un evaluador de GA podrá distribuir las computaciones sobre varios *procesos* o *tareas* que corran concurrentemente. Aún sigue siendo necesario que el proceso de evaluación sea consistente con el orden impuesto por las dependencias. Es preferible minimizar los puntos de sincronización entre los diferentes procesos o tareas para maximizar el paralelismo. El objetivo de cualquier método de evaluación paralela será encontrar particiones del conjunto de instancias de atributos en un árbol sintáctico que tengan la mayor *independencia posible* entre ellas.

Kuiper en [28], contribuye al estudio de métodos de evaluación paralela en dos areas: define el concepto de *distribuidor* como un modelo de asignación estática de instancias de atributos a procesos de evaluación y da un algoritmo para detectar estáticamente todos los pares de atributos independientes de una GA dada.

Un distribuidor debe encontrar un buen balance entre los dos aspectos siguientes:

- instancias de atributos independientes deberán ser asignados a diferentes procesos para maximizar la concurrencia.

- instancias de atributos dependientes deberán ser, tanto como sea posible, ser asignados al mismo proceso con el fin de minimizar la comunicación y sincronización entre los procesos.

Se analizan estrategias de particionado del árbol sintáctico y se presenta una estrategia basada en las dependencias que permite particionar el árbol en regiones independientes, por lo que cada proceso o tarea de evaluación no necesitará ninguna sincronización con los demás procesos.

5.1. Distribuidores de Kuiper

Kuiper en [28] define dos tipos de distribuidores: basados en el árbol sintáctico y basados en los atributos.

Los primeros asignan todas las instancias de atributos de un nodo al mismo proceso. Los distribuidores mas útiles de este tipo son los que particionan el árbol en regiones conexas (ya que el flujo de las dependencias siguen el árbol, ésto reduce la comunicación entre procesos). El criterio de selección estática mas comunmente utilizado es basado en la produción aplicada en cada nodo.

Un distribuidor basado en los atributos asigna todas las instancias de un atributo (o un conjunto de ellos) a un mismo proceso.

Un distribuidor *combinado* consiste en un distribuidor basado en el árbol seguido de un distribuidor basado en atributos: el primero divide el árbol en regiones y el último asigna instancias de atributos de una región a diferentes procesos.

Kuiper no introduce el estudio de distribuidores basados en las dependencias, esto es, en regiones conexas del grafo de dependencias.

5.2. Métodos exhaustivos

Es posible evaluar los atributos en forma concurrente mediante dos enfoques: dinámicos o estáticos.

5.2.1. Métodos dinámicos

Este método dispara un nuevo proceso por cada instancia a evaluar. El proceso espera hasta que todas las instancias de atributos de las que depende estén computadas. Cuando una instancia se computa se envía un mensaje (signal) a los procesos que esperan por ella.

Fueron uno de los primeros métodos en utilizarse para la evaluación concurrente de GA (FOLDS desarrollado por Fang en 1972). Durante el desarrollo de esta tesis, en [4] se describe un evaluador concurrente orientado a objetos (JAPLAGE) que utiliza éste método implementado utilizando Java threads.

El método tiene el inconveniente que se crean un gran número de procesos, donde cada uno ejecuta unas pocas instrucciones (o ninguna) y se bloquea esperando por que se computen las instancias requeridas por sus dependencias.

5.2.2. Métodos estáticos

La idea subyacente en los métodos estáticos es detectar el paralelismo mediante el análisis de las dependencias entre los atributos.

Uno de los enfoques que se han utilizado consiste en la extensión de la idea de secuencias de visita, como el propuesto por el autor de esta tesis en [8]. Un evaluador basado en secuencias de visita puede extenderse fácilmente a un evaluador concurrente particionando una secuencia de visitas $vs = op_1 op_2 \dots op_n$ utilizando constructores de la forma $vs_{par} = \dots par\{< s_i > \dots < s_{i+k} >\} \dots$ donde cada segmento $< s_j >$ $(i \leq j \leq k)$ es una secuencia de visitas. Las instancias de atributos en cada segmento

será computados por procesos independientes. Un segmento $< s_i >$ computa instancias de atributos independientes de instancias computadas por un segmento s_j $(i \neq j)$ del mismo constructor.

Los segmentos inducen una partición en una secuencia de visitas de una producción.

Este método no requiere de mecanismos de comunicación entre procesos ni sincronización, ya que está garantizado que:

- todas las instancias de los atributos necesarios para cada nuevo proceso creado ya están computadas.
- dados dos procesos cualesquiera, ambos no acceden a las mismas instancias que computan (escriben en diferentes áreas de memoria).

Como desventaja, en cada operación *par* se debe ejecutar una operación *fork* (que dispara un nuevo proceso o tarea). Al final de cada constructor *par* se debe ejecutar una operación *join*.

En [25], Klein define las *Gramáticas de atributos ordenadas paralelas* quien propone que los grafos de dependencias de cada producción sean divididos en segmentos. Durante la evaluación los segmentos de diferentes instancias de producciones en el árbol sintáctico se combinan para establecer las particiones del árbol. Los segmentos no son independientes, por lo tanto es necesario algún tipo de sincronización entre los procesos o tareas evaluadores.

A continuación se describe un método basado en las dependencias que genera particiones independientes. De esta forma los diferentes procesos evaluadores no requieren sincronización.

5.3. Un método estático basado en las dependencias

En esta sección se analiza un método de particionado del grafo de dependencias teniendo en cuenta las dependencias entre las instancias de los atributos y tiene como objetivo determinar estáticamente particiones independientes (disjuntas) que podrían ocurrir en un árbol sintáctico generado a partir de la GA.

El algoritmo original fue propuesto por Wuu Yang en [45] quien demuestra que obtiene la partición mas fina posible. El evaluador podrá estar basado en secuencias de visita que previo a la evaluación selecciona la partición de instancias de atributos correspondiente para cada nodo del árbol, lo cual particiona de las instancias de los atributos del árbol y finalmente dispara un proceso evaluador sobre cada partición.

Un árbol sintáctico se construye a partir de la aplicación sucesiva de producciones de la gramática. Una instancia de una producción en un árbol sintáctico tiene como *contexto inferior* a las instancias de las producciones aplicadas a los no terminales de la parte derecha.

Para determinar una partición del grafo de dependencias de una producción p se deben tomar en cuenta tres tipos de dependencias:

1. las dependencias directas obtenidas por las ecuaciones de p
2. las dependencias impuestas por el contexto superior

3. las dependencias impuestas en el contexto inferior

Instancias diferentes de una producción p tendrán las mismas dependencias directas, pero podrán tener diferentes dependencias impuestas por los contextos inferiores y superiores. Es necesario entonces, determinar todas las *dependencias posibles* entre las instancias de los atributos que ocurren en una producción para luego poder determinar todas las *particiones posibles* entre los atributos que ocurren en una producción.

Dado un árbol sintáctico en particular, se podrá *seleccionar* la partición adecuada dado los contextos superiores e inferiores de la instancia de una producción.

Para describir el algoritmo propuesto por Yang es necesario introducir algunas definiciones y propiedades.

Definición 5.3.1 *Una partición posible de un grafo de dependencias $G_T =< V_N, A >$ es una partición sobre V_N tal que dados dos instancias de atributos $X.a$ y $Y.b$, si $(X.a, Y.b) \in A$, entonces $[X.a] = [X.b]$ (están en la misma clase).*

La partición mas fina posible de un grafo de dependencias G_T es obviamente dada por la relación reflexiva, simétrica y transitiva (relación de equivalencia) de A.

Las particiones son independientes, por lo que las instancias de los atributos del árbol sintáctico T de cada partición pueden evaluarse concurrentemente por procesos evaluadores que no requerirán ningún tipo de sincronización ni comunicación.

Definición 5.3.2 *Sea π una partición posible de un grafo de dependencias $G_T =< V_N, A >$. Sea V_{N_X} un nodo rotulado X en el árbol sintáctico T. La proyección de π en las instancias de los atributos de X es una partición posible de los atributos de X.*

Asimismo es conveniente definir una partición posible de una producción.

Definición 5.3.3 *Sea π una partición posible de un grafo de dependencias $G_T =< V_N, A >$. Sea p una instancia de una producción en el árbol sintáctico T. La proyección de π en las instancias de los atributos que ocurren en p es una partición posible de los atributos que ocurren en p.*

El algoritmo que computa todas las posibles particiones posibles (en la función Π) propuesto por Yang en [45] (mostrado en la figura 10) se basa en realizar dos recorridos por el conjunto de producciones de la gramática.

En el primer recorrido ascendente computa las particiones posibles para cada producción de la forma $p : X_0 \rightarrow \alpha_0 X_1 \ldots X_k \alpha_k \in P$ teniendo sólo en cuenta sus posibles contextos inferiores[1] y las dependencias directas de la producción.

Las particiones posibles son almacenadas en la función $\Sigma(p|\sigma_1, \sigma_2, \ldots, \sigma_k)$, la cual asocia una combinación de particiones plausibles (σ_i, $1 \leq i \leq k$) de los atributos de X_i a una partición plausible de los atributos de X_0.

Obviamente una partición plausible es una refinamiento de una partición posible, dado que las restricciones impuestas por algún contexto superior puede que dos clases

[1] Denominadas por Yang *particiones plausibles*.

$\Xi(X) := \Theta(X) := \emptyset$, $\forall X \in V_N$
$\pi_p :=$ *finest partition based on* $DP(p)$, $\forall p \in P$
repeat {bottom-up pass}
  changed := false
  for each $p : X_0 \to \alpha_0 X_1 \ldots X_k \alpha_k \in P$ **do**
    for each $\sigma_i \in \Xi(X_i)$, $(1 \leq i \leq k)$ **do**
      $\pi := merge(\ldots merge(merge(\pi_q, \sigma_1), \sigma_2, \ldots, \sigma_k)$
      $\Sigma(q \mid \sigma_1, \ldots, \sigma_k) := \sigma := \pi\|_{A(X_0)}$
      if $\sigma \notin \Xi(X_0)$ **then**
        changed := true; $\Xi(X_0) := \Xi(X_0) \cup \{\sigma\}$
      end if
    end for
  end for
until not changed
$\Theta(S) := \Xi(S)$, where S is the start symbol
repeat {top-down pass}
  changed := false
  for each $p : X_0 \to \alpha_0 X_1 \ldots X_k \alpha_k \in P$ **do**
    for each $\sigma_i \in \Theta(X_i)$, $(0 \leq i \leq k)$ **do**
      if $\Pi(q \mid \sigma_0, \ldots, \sigma_k)$ is undefined **then**
        $\pi := merge(\ldots merge(merge(\pi_q, \sigma_0), \sigma_1, \ldots, \sigma_k)$
        $\Pi(q \mid \sigma_0, \ldots, \sigma_k) := \sigma := \pi$
        for each $i := 1$ to k **do**
          $\sigma := \pi\|A(X_i)$
          $\Phi(q, \pi, i) := \sigma$
          if $\sigma \notin \Theta(X_i)$ **then**
            changed := true; $\Xi(X_i) := \Xi(X_i) \cup \{\sigma\}$
          end if
        end for
      end if
    end for
  end for
until not changed

Algoritmo 10: Algoritmo de generación de particiones posibles de cada producción

de una partición plausible se unan en una partición posible (en el caso de una dependencia transitiva entre dos instancias de atributos impuesta por el contexto superior).

Para el símbolo de comienzo la partición plausible coincide con una partición posible (ya que no tiene contexto superior).

En el segundo recorrido (descendente) se computan las particiones posibles para cada producción combinando la partición base de cada producción q, una partición de los atributos de X_0 (σ_0) y ua partición plausible de los atributos de X_i, ($i > 0$). El resultado de esta combinación se almacena en $\Pi(q \mid \sigma_0, \ldots, \sigma_k)$, la cual será usada durante la evaluación.

La función $merge(\pi, \sigma)$ computa la partición mas fina compatible con las particiones π y σ.

La computación, tanto de las particiones posibles como de las secuencias de visita, pueden incluirse en el algoritmo del test de circularidad, ya que tienen la misma complejidad computacional (exponencial).

A modo de ejemplo, la figura 5.1 muestra dos árboles sintácticos para la gramática de atributos de la figura 5.2.

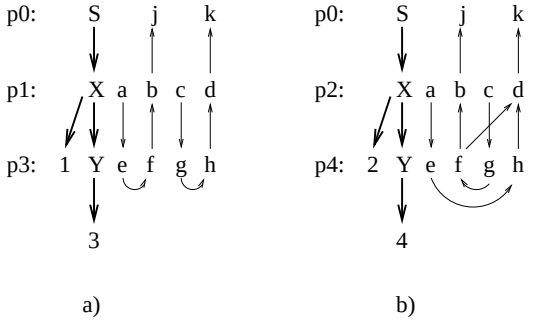

Figura 5.1: Dos árboles sintácticos de la GA de la figura 5.2

Las particiones posibles para la producción p_0 son $\pi_1 = \{< a, b, j >, < c, d, k >\}$ (figura 5.1 a)), $\pi_2 = \{< a, d, k >, < b, c, j >\}$ $\pi_3 = \{< a, b, c, d, j, k >\}$ (figura 5.1 b)).

Las de p_1 son $\pi_4 = \{< a, b, e, f >, < c, d, g, h >\}$ (figura 5.1 a)), $\pi_5 = \{< a, b, c, d, e, f, g, h >\}$ y $\pi_6 = \{< a, d, e, h >, < b, c, f, g >\}$.

La única partición posible de p_2 es π_5 (figura 5.1 b)).

Las de p_3 son $\pi_7 = \{< e, f >, < g, h >\}$ (figura 5.1 a)), y $\pi_8 = \{< e, f, g, h >\}$.

La única partición posible de p_4 es π_8 (figura 5.1 b)).

p_0: $S \rightarrow X$
attribution
$S.j = X.b$
$S.k = X.d$
$X.a = 0$
$X.c = 1$
end

p_1: $X \rightarrow 1\ Y$
attribution
$Y.e = X.a$
$X.b = Y.f$
$Y.g = X.c$
$X.d = Y.h$
end

p_2: $X \rightarrow 2\ Y$
attribution
$Y.e = X.a$
$X.b = Y.f$
$Y.g = X.c$
$X.d = f(Y.h, Y.f)$
end

p_3: $Y \rightarrow 3$
attribution
$Y.f = Y.e$
$Y.h = Y.g$
end

p_4: $Y \rightarrow 4$
attribution
$Y.h = Y.e$
$Y.f = Y.g$
end

Figura 5.2: Gramática de ejemplo

Un evaluador basado en este enfoque deberá seleccionar una de las particiones posibles en cada nodo de un árbol sintáctico. Yang propone hacerlo en dos recorridos (aunque se puede realizar en uno solo, aún en forma incremental, a medida que el árbol sintáctico es generado por el parser).

El evaluador (tiempo de ejecución) sólo tiene que realizar el proceso de selección de las particiones. Toda la información sobre las particiones se realizó en tiempo de compilación-compilación. Una vez seleccionadas las particiones en cada nodo es posible lanzar un proceso evaluador para cada partición.

La figura 11 muestra el algoritmo de selección de particiones para cada nodo de un árbol sintáctico T.

El evaluador deberá disparar el proceso de selección y luego podrá aplicar un proceso de evaluación por cada partición inducida en T.

Procedure choose(n)
choose_plausible(n)
choose_partition(n,partition_choosen(n))
end procedure

Procedure choose_plausible(n)
let $p : X_0 \rightarrow \alpha_0 X_1 \ldots X_k \alpha_k$ the production applied at node n
for each $c = child_i(n)$, $(1 \leq k)$ **do**
  choose_plausible(c)
end for
let $\sigma_1, \ldots, \sigma_k$ chosen at $child_i(n)$, $(1 \leq k)$
choose $\Sigma(p \mid \sigma_1, \ldots, \sigma_k)$ for node n
end procedure

Procedure choose_partition(n, σ)
let $p : X_0 \rightarrow \alpha_0 X_1 \ldots X_k \alpha_k$ the production applied at node n
let $\sigma_1, \ldots, \sigma_k$ chosen at $child_i(n)$, $(1 \leq k)$
choose $\pi = \Pi(q \mid \sigma, \sigma_1, \sigma_2, \ldots, \sigma_k)$ for node n
for each $child_i(n)$, $(1 \leq k)$ **do**
  choose_partition$(child_i, \Phi(p, \pi, i))$
end for
end procedure

Algoritmo 11: Algoritmo de selección de particiones en un árbol sintáctico T

En el próximo capítulo se describe una herramienta para la generación estática de evaluadores concurrentes de gramáticas NC(1) basado en un algoritmo de particionado como el analizado en esta sección. Hasta el momento de la finalización de este trabajo no se conocen otras herramientas similares.

Capítulo 6

NCEval

En este capítulo se describe una herramienta denominada *nceval*, que genera evaluadores concurrentes, con mínima comunicación entre procesos para gramáticas de atributos de la familia $NC(1)$. La herramienta se ha desarrollado en el marco de la presente tesis y es una de sus principales contribuciones.

---

p_0: $S \rightarrow X$
attribution
S.j:=X.b
S.k:=X.d
X.a:=0
X.c:=1
end

p_1: $X \rightarrow 1Y$
attribution
Y.e:=X.a
X.b:=Y.f
Y.g:=X.c
X.d:=Y.h
end

p_2: $X \rightarrow 2Y$
attribution
Y.e:=X.a
X.b:=Y.f
Y.g:=X.c
X.d:=plus(Y.h,Y.f)
end

p_3: $Y \rightarrow 3$
attribution
Y.f:=Y.e
Y.h:=Y.g
end

p_4: $Y \rightarrow 4$
attribution
Y.h:=Y.e
Y.f:=Y.g
end

Figura 6.1: GA1: Un ejemplo de una GA NC(1)

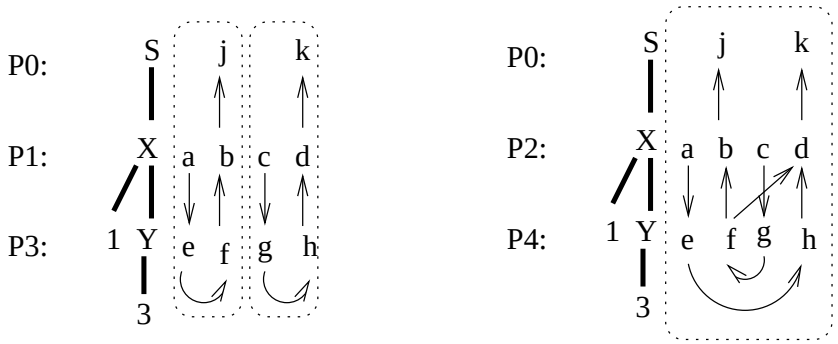

Figura 6.2: Dos instancias de árboles derivados de ga1

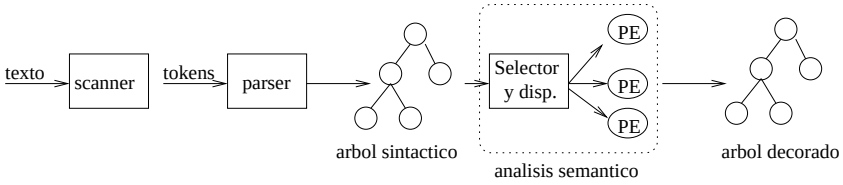

Figura 6.3: Estructura conceptual de un compilador concurrente

Como se vio en los capítulos anteriores, la familia elejida contiene a las *absolutamente no circulares* (ANCAG) y por lo tanto permite trabajar con una familia lo suficientemente expresiva para la mayoría de los casos en la práctica.

En evaluador genera evaluadores concurrentes cuyo particionamiento de los atributos está basado en las dependencias y se generan particiones disjuntas. De esta manera los procesos evaluadores no requieren ningún tipo de sincronización.

La generación del evaluador y la detección de posibles ciclos en algún árbol sintáctico, se implementó en base a los algoritmos vistos en la sección 3.3 del capítulo 2.

En la figura 6.1 se muestra una GA NC(1) y en la figura 6.2 se muestran dos instancias diferentes de árboles atribuidos (las flechas indican las dependencias entre los atributos).

6.1. Esquema general del sistema

En la figura 6.3 se muestra el esquema de un compilador concurrente que utilizaría un analizador semántico generado por *NCEval*.

NCEval es responsable por la generación del evaluador de atributos (denominado analizador semántico en la figura 6.3.

NCEval toma como entrada una especificación de una gramática de atributos como la de la figura 6.1 y luego realiza los siguientes pasos:

1. Generar los planes de evaluación para cada producción en base al cómputo de los grafos $ADP(p)$, para cada producción p.

2. Computar π_q: las particiones viables de las ocurrencias de atributos de cada producción q y ordenarlas con respecto a los ordenes de evaluación correspondientes.

3. Generar secuencias de visita para cada producción a partir de la proyección de las particiones en base a los posibles contextos superiores.

La figura 6.4 muestra los planes generados para las producciones de la gramática de atributos de la figura 6.1.

```
P0: {[a,c,b,d,j,k]}
P1: {[a,e,c,f,g,h,b,d],[c,a,g,e,f,h,b,d]}
P2: {[a,c,e,g,f,h,b,d]}
P3: {[e,f,g,h]}
P4: {[e,g,f,h]}
```

Figura 6.4: Planes generados

6.2. Generación de planes de evaluación

El algoritmo 12 genera los planes de evaluación de una gramática de atributos.

El algoritmo utiliza una lista de trabajo (WL) que contiene pares de la forma (p, o), donde p es una producción y o es un orden de evaluación de los atributos del símbolo de la parte izquierda de p.

Cualquier orden arbitrario de los atributos del símbolo de comienzo (S) es un orden correcto[1].

El algoritmo finaliza cuando la lista de trabajo WL no contenga elementos. En cada iteración, se toma un elemento de (q, ω) de WL y se procede a calcular los posibles ordenes de evaluación para la producción p, la cual se define en la función Γ (representada como un arreglo).

$\Gamma[q, \omega, p_1, p_2, \ldots, p_k]$ denota el orden para una instancia de una producción q, siendo ω el orden de evaluación de los atributos del símbolo de su parte izquierda y con instancias de las producciones $p_1, p_2, \ldots, p_k$ como contexto inmediato inferior.

Habiendo calculado el orden para esta instancia particular de la producción q, luego se proyectan los ordenes de los atributos de los símbolos no terminales de la parte izquierda de las producciones $p_1, p_2, \ldots, p_k$ y se insertan los nuevos pares obtenidos (p_i, ω_i) en la lista de trabajo WL.

La función $project(\Psi, A(X_i))$ toma un plan (secuencia) Ψ y retorna un plan sólo conteniendo los atributos de X_i, manteniendo el orden entre los atributos de X_i según Ψ.

La función $TotalOrder(o)$ retorna un orden total compatible con el orden parcial o, es decir, adiciona arcos para que el orden resultante (visto como un grafo) tenga una única componente conexa.

6.3. Generación de las particiones

Las particiones viables de los atributos que ocurren en cada producción se computan mediante la implementación del algoritmo 10 en el capítulo enterior.

[1] Recordar que el símbolo de comienzo sólo contiene atributos sintetizados.

$WL = \emptyset$
$\Pi(q_0, \mu) := defined$, (μ is a arbitrary order of $A(S)$)
$WL := WL \cup \{(q_0, \mu)\}$
repeat
  $(q, \omega) :=$ an element of WL; $WL := WL - \{(q, \omega)\}$
  let $q : X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$
  for each $ADP(q|p_1, p_2, \ldots, p_k) \in SADP(q)$ **do**
    $\Psi := TotalOrder((ADP(q|p_1, p_2, \ldots, p_k) \cup \omega)$
    $\Gamma[q, \omega, p_1, p_2, \ldots, p_k] := \Psi$
    for each X_i, $(1 \leq i \leq k)$ **do**
      $\omega_i := project(\Psi, A(X_i))$
      $\Theta[q, \omega, i, p_1, p_2, \ldots, p_k] := \omega_i$
      if $\Pi(p_i, \omega_i) = undefined$ **then**
        $\Pi(p_i, \omega_i) := defined$
        $WL := WL \cup \{(p_i, \omega_i)\}$
      end if
    end for
  end for
until $WL = \emptyset$

Algoritmo 12: Algoritmo de generación de planes

Para el ejemplo de la figura 6.1 los planes posibles de cada producción se muestran en figura 6.5.

Se debe notar que las particiones obtenidas se deberán ordenar en base a los planes computados para cada producción.

```
P0:{{[k,d,c],[b,a,j]},{[k,c,d,b,j,a]}}
P1:{{[c,d,h,g],[b,e,f,b]},{[c,d,f,g],[d,h,e,a]},{[d,e,c,g,h,f,b,a]}}
P2:{{[e,a,c,d,g,h,b,f]},{[f,a,d,e,c,h,b,c]}}
P3:{{[g,f],[h,e]},{[e,h,f,h]}}
P4:{{[h,g],[e,f]},{[h,f,e,g]}}
```

Figura 6.5: Particiones posibles

En la figura 6.2 se muestran las particiones de atributos que inducen regiones disjuntas para su evaluación concurrente. El árbol de la figura 6.2 a) será evaluado por dos procesos concurrentes independientes, mientras que el árbol de la figura 6.2 b) debará ser evaluado por un solo proceso.

6.4. El evaluador generado

El programa generado por NCEval contiene la representación de los nodos del árbol, los cuales contienen los atributos del símbolo correspondiente, el número de producción del cual será la instancia raíz y referencias a los planes (en realidad a particiones de planes) que serán seleccionados en tiempo de ejecución.

El algoritmo de evaluación generado es una implementación del algoritmo mostrado en la figura 13.

```
μ is any order of atributes of root(Tree)
rr := select_partitions(Tree, μ)
max := 0
for each node (n, regions) ∈ rr do
  for i:=1 to regions − max do
    spawn eval_vs(n)
  end for
  max := regions
end for
end
```

Algoritmo 13: EvalAttributes(Tree)

La función $select\_partitions(Tree, \mu)$ es una implementación directa del algoritmo 11 visto en el capítulo anterior en la sección 5.3 con el efecto adicional que retorna una lista de pares de la forma $(node, regions)$ denominados *root regions*.

Cada par en *root region* contiene una referencia al nodo el cual corresponde a la raíz de una región en el árbol, inducida por las particiones seleccionadas. La componente *regions* del par denota el número de regiones en que se particiona el subárbol cuya raíz es *node*.

La detección de *root regions* se realiza llevando el número máximo de elementos de una partición (desde la raíz) durante el recorrido descendente del algoritmo 11 de la sección 5.3.

Cuando se visita a un nodo en el que se selecciona una partición con l elementos, $l - max > 0$ determina que el nodo corriente es un *root region* y $l - max$ es el número de regiones[2].

Cada proceso evaluador $eval\_vs_i(n)$ evalúa secuencialmente las instancias de los atributos que pertenecen al elemento i de la partición seleccionada en cada nodo del árbol.

Cada evaluador está basado en el algoritmo 5 de interpretación de secuencias de visita analizado en el capítulo 3.

6.5. Implementation

Se ha realizado una primera versión de NCEval en Java, con un diseño orientado a objetos, donde los procesos evaluadores son representados actualmente por threads.

El principal objetivo en el desarrollo de NCEval es que pueda integrarse en un sistema de procesamiento de lenguajes basado en gramáticas de atributos.

Esa es la principal razón en la que no se ha incluido en la implementación ninguna facilidad de parsing u otro componente, justamente para que sea un componente independiente de los mecanismos de parsing.

[2]La variable *max* inicialmente es cero.

La versión actual se ha utilizado como prototipo para tratar de determinar la factibilidad de implementación de las ideas presentadas en los capítulos anteriores y para poder realizar experimentos con gramáticas de atributos que se puedan encontrar en la práctica.

En [8] se presenta en mayor detalle las características de diseño de NCEval.

Capítulo 7

Extensiones de Gramáticas de Atributos

En este capítulo se estudian algunas extensiones que propuestas a las gramáticas de atributos. Las extensiones pretenden expandir la expresividad en algún sentido haciéndolas mas utilizables en la práctica dentro de algunos dominios de aplicación específicos.

Cualquier extensión generalmente afecta las formas de evaluación, por lo que además de caracterizarlas se analizan las adaptaciones de los métodos para su evaluación.

En las secciones siguientes se analizarán las gramáticas de atributos de alto orden, las gramáticas de atributos condicionales y las gramáticas orientadas a objetos.

7.1. Gramáticas de atributos de alto orden

Las Gramáticas de atributos de alto orden o *Higher Order Attribute Grammars* (HAG's), son una extensión de las gramáticas de atributos normales en el sentido en que desaparece la distinción entre los dominios de los árboles sintácticos y de los atributos. Se pueden computar árboles sintácticos en atributos y como consecuencia las funciones semánticas se pueden describir por medio de la evaluación de los atributos.

Una gramática de atributos tradicional describe la computación de atributos, es decir, valores relacionados a los nodos de un árbol sintáctico. El árbol se describe con una gramática libre de contexto y la computación de los atributos se define por medio de las funciones semánticas.

Las HOAG's promueven los árboles sintácticos (abstractos) a *"ciudadanos de primera clase"*. Es decir que un árbol sintáctico puede ser el resultado de una función semántica y pueden ser representados como atributos.

Los árboles utilizados como valores y definidos por reglas de atribución se denominan *atributos no-terminales (NTA's)*.

Una de las principales motivaciones que han dado lugar a las HAG's ha sido la debilidad de las GA tracionales dada por la separación entre sus niveles sintácticos y semánticos (donde la sintaxis es prioritaria). Los atributos sufren restricciones dadas

por las equaciones que los definen, mientras que el árbol sintáctico es irrestricto (excepto por las restricciones de la gramática libre de contexto subyacente). Los atributos transportan información entre los diferentes contextos, ellos no tienen forma de generar árboles de derivación.

En ciertos problemas, como ambientes de edición, compiladores de múltiples pasadas o en herramientas como asistentes de pruebas y sistemas de transformación de programas, es deseable que los atributos *guíen* a la sintaxis.

7.1.1. Definición de HAG's

Una gramática de atributos es una gramática de atributos con las siguientes extensiones(definiciones dadas por D. Swierstra y H. Vogt en [2]):

Definición 7.1.1 *Para cada producción $p : X_0 \to X_1 \dots X_k \in P$, el conjunto de atributos no terminales (NTA), está definido por*

$$NTA(p) = \{X_j \mid X_j := f(\dots) \in R(p)\}$$

Dado que un atributo no terminal también es un atributo, un árbol sintáctico puede contener NTA's[1] como hojas. Por este motivo, se debe cambiar la noción de árbol sintáctico.

Definición 7.1.2 *Una instancia del no terminal X en un árbol sintáctico se denomina:*

- *no terminal virtual si $X \in \bigcup_{p \in P} NTA(p)$ y la ecuación que define a X no sido evaluada todavía.*
- *no terminal instanciado si $X \notin \bigcup_{p \in P} NTA(p)$, o $X \in \bigcup_{p \in P} NTA(p)$ y la ecuación que define a X ha sido evaluada.*

Definición 7.1.3 *Un árbol sintáctico rotulado se define como:*

- *las hojas están rotuladas con símbolos terminales o no terminales virtuales*
- *los nodos (interiores) están rotulados con símbolos no terminales instanciados*

De aquí en adelante denotaremos a un atributo no terminal X como $\overline{\mathrm{X}}$.

Definición 7.1.4 *La función semántica f en una regla $\overline{X} := f(\dots)$ está bien tipada si f retorna un término representando un árbol sintáctico derivable de X.*

La definición anterior se usará para asegurar que un NTA $\overline{\mathrm{X}}$ será expandido a un árbol de derivación a partir de X.

[1] Atributos no terminales aún no computados

Definición 7.1.5 *Una gramática de alto orden es completa, si la gramática de atributos subyacente es completa y* $\forall p \in P$ *de la forma* $p : Y \rightarrow \mu$*, se cumple:*

- $NTA(p) \subseteq A(p)$

y $\forall \overline{X} \in NTA(p)$*:*

- $\overline{X}$ *ocurre en* μ
- *para toda ecuación de la forma* $\alpha := f(\gamma) \in R(p)$, $\overline{X} \notin \gamma$
- *para toda equación de la forma* $\overline{X} := f(\gamma) \in R(p)$, f *es* bien tipada

La definición anterior establece que los NTA's que son atributos locales a una producción, ocurren como un no terminal en la parte derecha de la producción y como un atributo en la parte izquierda de una ecuación.

La figura 14 muestra un algoritmo de evaluación para gramáticas de atributos de alto orden.

```
input: T (un árbol sintáctico rotulado no evaluado)
D := DT(T) { relación de dependecia entre atributos en T }
S := minimals(D) { instancias de atributos listos para evaluación }
while S ≠ ∅ do
  α ∈ S ; S = S − {α}
  t := eval(α)
  if α = X̄ ∈ NTA then
    expand(T, X̄, t)
    D := D ∪ DT(X)
    S := S ∪ minimals(DT(X))
  end if
  for all β ∈ succesor(α, D) do
    if ready_for_eval(β) then
      S := S ∪ β
    end if
  end for
end while
```

Algoritmo 14: Algoritmo de evaluación para HAG's

La evaluación comienza evaluando los atributos listos para la evaluación[2]. Cuando se computa una instancia de no terminal virtual $\overline{\mathrm{X}}$ se expande el árbol con el árbol obtenido t de $eval(\overline{\mathrm{X}})$ en la hoja rotulada $\overline{\mathrm{X}}$.

Ahora el no terminal virtual $\overline{\mathrm{X}}$ se torna un no terminal instanciado. Se expande el grafo de dependencias para contener las nuevas dependencias del árbol t. Luego se continua con la evaluación hasta que se hayan evaluado todas las instancias de atributos y se hayan realizado todas las posibles expansiones.

Debemos notar que hay dos problemas con el algoritmo de la figura 14:

[2]Aquellos atributos que no dependan de ningún otro.

1. no terminación.
2. una instancia de un atributo podría no recibir un valor.

El problema de la no terminación podría darse si el árbol sintáctico crece indefinidamente, en cuyo caso siempre habrá instancias de atributos no terminales virtuales que serán instanciados (expandidos).

Hay dos razones por las cuales una instancia de un atributo podría no recibir un valor:

1. un ciclo en D (nunca estrán listas para la evaluación)
2. si existe una instancia de un atributo no terminal $\overline{\mathrm{X}}$, el cual depende de un atributo sintetizado de $\overline{\mathrm{X}}$

El segundo problema surge porque los atributos sintetizados de $\overline{\mathrm{X}}$ se computan después que se realiza la expansión del árbol.

Para prevenir esta última situación, haremos que todos los atributos sintetizados de $\overline{\mathrm{X}}$ dependan de $\overline{\mathrm{X}}$. Por esto es necesario redefinir el conjunto de dependencias directas de una producción.

Definición 7.1.6 *Para cada $p : X_0 \to X_1 \dots X_k \in P$, el conjunto de dependencias directas extendidas está dado por*

$$EDP(p) = \{(\alpha \to \beta) \mid \beta := f(\dots \alpha \dots) \in P\}$$

$$\cup \{(\overline{X} \to \gamma) \mid \overline{X} \in NTA(p) \wedge \gamma \in S(X)\}$$

Esta última definición lleva al siguiente lema:

Lema 7.1.1 *Cada atributo no terminal virtual será computado si y sólo si no existen ciclos en D (usando EDP) durante el proceso de evaluación.*

Demostración: El uso de $EDP(p)$ prohibe que un atributo no terminal β sea definido en base a instancias de atributos que estarán alojados en el árbol que será conputado en β, ya que en caso contrario se introducirían ciclos en $EDP(p)$. □

Definición 7.1.7 *Una gramática de atributos es bien definida si para cualquier árbol sintáctico derivado todos los atributos pueden computarse utilizando el algoritmo 14.*

7.1.2. HAG Ordenadas (OHAG's)

Como se vio en los capítulos anteriores, la familia OAG es ampliamente utilizada en la práctica, ya que permite obtener órdenes de evaluacion con un algoritmo cuya complejidad es polinomial. De una HAG podrían derivarse órdenes si le aplicamos una *transformación* para obtener una GA tradicional y aplicar el algoritmo de Kastens para obtener los ordenes de evaluación y obtener las *secuencias de visita*.

A continuación se define una transformación de una HAG en una AG tradicional.

Definición 7.1.8 *Se H una HAG. La AG H' reducida es el resultado de la siguiente transformación de H:*

1. *reemplazar todas las ocurrencias de* $\overline{X}$ *en el rhs(p) por* X *, para todo* $p \in P$
2. $A(X) = A(X) \cup X.tree$
3. *reemplazar todas las ocurrencias de* $\overline{X}$ *en el lhs(p) por X.tree, para todo* $p \in P$

Teorema 7.1.1 *Una HAG es ordenada si la AG reducida correspondiente es OAG.*

Demostración: Si reemplazamos las ocurrencias de $X.tree$ en los órdenes de la AG reducida por NTA's $\overline{X}$, los ordenes obtenidos son ordenados. □

Existen varias herramientas que utilizan la familia OHAG e implementan la evaluación utilizando la técnica de secuencias de visita analizada anteriormente.

7.2. Gramáticas de atributos condicionales

En la práctica es deseable que las herramientas basadas en gramáticas de atributos contemplen ecuaciones condicionales de la forma

$$A.x := \text{ if } cond \text{ then } f(\ldots B.y \ldots) \text{ else } g(\ldots C.z \ldots)$$

donde *cond* es un predicado el cual puede tener referencias a ocurrencias de atributos de símbolos de la producción. Obviamente es posible abstraerse de estas construcciones, ya que podrían reemplazarse por funciones semánticas de la forma

$$c(cond, \ldots B.y \ldots C.z \ldots)$$

cuya definición sería tal cual la expresión condicional de arriba, pero su utilización no es muy práctica. Además, generalmente se asume que las funciones semánticas de una gramática de atributos son estrictas, por lo que deberían computarse previamente las instancias de los atributos que aparezcan como argumentos de la función c.

Es preciso hacer notar que en ciertos casos, algunas instancias de los atributos que aparezcan en una expresión condicional, podrían no evaluarse (si es que no son utilizados en otra ecuación).

Esto ha motivado que se hayan propuesto varias extensiones a las gramáticas de atributos tradicionales (o estándar) para soportar el uso de reglas de atribución condicionales.

Es fácil de ver que los atributos que aparezcan en la condición deberían ser evaluados *antes* de los atributos que aparecen en las funciones semánticas f y g.

En [12] Boyland propone un cambio sintáctico para agrupar en un mismo bloque las ecuaciones que dependen de la misma condición. La técnica que propone se basa en la división (split) de una producción con k ecuaciones condicionales en 2^k nuevas producciones. Cada una de esas reglas se corresponde con una combinación de los valores (true o false) de los k predicados en las ecuaciones. Esta técnica la denomina *node-spliting*.

De esta forma, se obtiene una nueva gramática de atributos sin ecuaciones condicionales y se extienden los grafos de dependencias directas de cada producción, para que las instancias de los atributos que aparecen como argumentos en cada función

semántica del cuerpo de cada una de las ramas de la ecuación condicional, dependan de las instancias de los atributos que aparecen en el predicado.

Boyland, además, realiza un análisis de varias propiedades de una gramática obtenida por *node-splitting* como circularidad, aplicación del método a cada familia estudiadas anteriormente y sus métodos de evaluación.

La técnica de *node-splitting* se enfoca en la detección de circularidades falsas que podrían originarse por distintos órdenes[3] causados por las ramas de una ecuación condicional.

Wuu Yang, en [43], propone un método de evaluación de gramáticas de atributos condicionales haciendo hincapié en evitar evaluaciones de instancias de atributos *inútiles* (aquellas instancias que no serán necesarias en ninguna ecuación luego de haberse evaluado la condición). En su enfoque, propone que cada instancia de atributo que aparezca en una de las ramas de una ecuación condicional, tenga una *guarda*, la cual habilitará (o deshabilitará) dinámicamente su evaluación.

7.3. Gramáticas de atributos y lenguajes de programación

En las siguientes subsecciones se realiza un análisis de las GA y su relación con lenguajes de programación de tres paradigmas diferentes: funcional, lógico y orientado a objetos.

7.3.1. Gramáticas de atributos y programación funcional

Una GA puede transformarse directamente en programas funcionales que soporten evaluación lazy. La transformación es muy simple e intuitiva: cada no terminal se corresponde con una función, las producciones a diferentes casos en el cuerpo de la función que se corresponde con el no terminal de la parte izquierda, los atributos heredados se corresponden con parámetros y los atributos sintetizados se encapsulan en el valor resultante (tupla o lista) de la función.

Con evaluación lazy es posible aún evaluar gramáticas de atributos circulares siempre que la relación de dependencias defina un reticulado finito y que las funciones semánticas sean monótonas tal como se vio en el capítulo 4

La evaluación lazy permite en forma absolutamente transparente la implementación bajo demanda, ya que un valor de un atributo no se evaluará hasta que sea utilizado.

La implementación de gramáticas de atributos con lenguajes funcionales modernos como Haskell permiten la obtención de programas altamente declarativos, ya que la traducción es casi directa.

Otra ventaja de los lenguajes declarativos como los funcionales y lógicos es que la gramática libre de contexto subyacente se traduce casi literalmente.

Se puede encontrar mas información sobre la relación de los formalismos en [36] y en [22].

[3] Originados por órdenes de evaluación mutuamente exclusivos.

7.3.2. Gramáticas de atributos y programación lógica

Es posible implementar una gramática de atributos de una manera muy simple realizando una traducción muy simple a programas lógicos (definite programs).

Cabe recordar que lenguajes lógicos como Prolog fueron creados inicialmente para atacar problemas de reconocimiento de lenguaje natural, por lo que son lenguajes altamente recomendables al menos para la implementación de prototipos de sistemas basados en gramáticas de atributos.

La mayoría de los sistemas Prolog permiten la definición de *Definite Clause Grammars (DGC)* las cuales son una notación para describir reglas de una gramática libre de contexto y soportan la noción de atributos (argumentos de los predicados), las cuales son traducidas directamente a *definite programs*.

p_0: $N \rightarrow L$
attribution
$N.v{:=}L.v$
end
p_1: $N \rightarrow L.L$
attribution
$N.v := L[1].v + L[2].v \,/\, 2^{L[2].l}$
end
p_2: $L \rightarrow L\ B$
attribution
$L[1].v := 2{*}L[2].v + B.v$
$L[1].l := L[2].l + 1$
end
p_3: $L \rightarrow B$
attribution
$L.v := B.v$
$L.l := 1$
end
p_4: $B \rightarrow 0$
attribution
$B.v := 0$
end
p_5: $B \rightarrow 1$
attribution
$B.v := 1$
end

Figura 7.1: GA que transforma valores binarios a reales en decimal

La semántica de una gramática de atributos puede definirse en términos de árboles sintácticos decorados (todas las instancias de los atributos se han computado) mientras que la semántica de un programa lógico puede definirse en términos de árboles de pruebas.

En la comparación de conceptos surgen algunas diferencias. Algunas de las características de las gramáticas de atributos que no están presentes en los programas lógicos se podrían mencionar: tipos de datos (many-sorted), la noción de relación de depen-

dencia y la noción de dominios semánticos de las funciones de atribución.

Un concepto en los que ambos mundos coinciden es el de que ambos formalismos representan (o se implementan) como un *programa de lógica con restricciones.*

En base a lo expuesto es posible dar una semántica declarativa de una gramática de atributos, esto es, dar una caracterización de un árbol sintáctico decorado por medio de árbol de prueba (proof tree).

Para conseguir tal caracterización es posible relacionar cada elemento de una gramática de atributos con un elemento de un árbol de prueba:

1. cada regla semántica se corresponde con una ecuación,

2. cada ocurrencia de un atributo se corresponde con una variable,

3. la parte derecha de una producción se corresponde con un término con los símbolos de función denotando alguna función total sobre su dominio semántico correspondiente.

Las ecuaciones semánticas de la gramática de atributos asociadas con una instancia de una producción pueden verse como ecuaciones caracterizando relaciones entre los valores de las instancias de los atributos.

De esta manera, cada árbol sintáctico denota un conjunto único de ecuaciones. Si la gramática de atributos no es circular, éste conjunto tiene una única solución caracterizando la decoración del árbol.

Se debe notar que esta decoración es puramente declarativa, es decir, no especifica cómo se deberían resolver las ecuaciones.

Las principales diferencias entre las gramáticas de atributos y la programación lógica son:

1. En las gramáticas de atributos el parsing es un concepto separado de la evaluación de atributos (*tree decoration*) mientras que en programas lógicos no hay tal separación.

2. La evaluación de atributos generalmente es un proceso multi-visita. SLD-resolución visita cada nodo del árbol de pruebas una sola vez y las computaciones se producen mediante unificación.

$$
\begin{array}{l}
b(0) \leftarrow zero \\
b(1) \leftarrow one \\
l(V,1) \leftarrow b(V) \\
l(2 * V1 + V2, L + 1) \leftarrow l(V1, L), b(V2) \\
n(V) \leftarrow l(V) \\
n(V1 + V2/2^{L2}) \leftarrow l(V1, L1), point, l(V2, L2) \\
zero \leftarrow \\
one \leftarrow \\
point \leftarrow
\end{array}
$$

Figura 7.2: Programa lógico obtenido

Ejemplo: la gramática de atributos de la figura 7.1 que describe la transformación numerales binarios en los números reales en correspondientes en decimal.

La figura 7.2 muestra el programa lógico obtenido por el método de transformación descripto.

Las DGCs (Definite Grammar Clauses) fueron introducidas para enriquecer a los programas lógicos con el concepto de *lenguaje*. Las *DGCs* pueden describir gramáticas (aún sensible del contexto) y pueden verse como un *enriquecimiento sintáctico* (syntax sugar), ya que los sistemas Prolog generalmente las "traducen" a cláusulas Prolog, o bien como un formalismo independiente.

Definición 7.3.1 *Una DGC es una tupla* $< N, T, P >$*, donde*

- N *es un conjunto (posiblemente infinito) de átomos.*
- T *es un conjunto (posiblemente infinito) de términos.*
- $P \subseteq N \times (N \cup T)^*$ *es un conjunto finito de reglas o producciones.*

Análogamente a las CFG, $N \cap T = \emptyset$ (conjuntos disjuntos) y N y T se denominan *no terminales* y *terminales*, respectivamente.

Las *DGC* son generalizaciones de las CFG y por lo tanto es posible generalizar el concepto de derivabilidad:

Definición 7.3.2 *Sean* $\alpha, \alpha', \beta \in (N \cup T)^*$*, y sea* $p(t_1, \ldots, t_n) \rightarrow \beta \in P$*,* α' *es directamente derivable desde* α *si y sólo si:*

- α *tiene la forma* $\alpha_1 p(s_1, \ldots, s_n)\alpha 2$*,*
- $p(t_1, \ldots, t_n)$ *y* $p(s_1, \ldots, s_n)$ *unifican*[4]*,*
- α' *tiene la forma* $\alpha_1 \beta \alpha_2$*.*

```
expr(X) --> term(Y), [+], expr(Z), { X is Y + Z }.
expr(X) --> term(X).
term(X) --> factor(Y), [*], term(Z), { X is Y * Z }.
term(X) --> factor(X).
factor(X) --> [X], { integer(X) }.
```

Figura 7.3: Una DGC que computa el valor de una expresión.

La figura 7.3, muestra una DGC de ejemplo. Como puede observarse, en una DCG es posible insertar código Prolog, lo que permite realizar computaciones extras durante el reconocimiento de la cadena de entrada. Es posible incorporar atributos a una DGC mediante argumentos (parámetros). La evaluación de atributos se puede combinar con $SLD - Resolution$ (mecanismo de inferencia utilizado en Prolog). Un argumento de entrada se corresponde con un atributo heredado y un argumento de salida con un atributo sintetizado. Los átomos deben encerrarse entre corchetes ([]), para que Prolog los pueda distinguir.

Los sistemas Prolog generalmente traducen la DGC de la figura 7.3 a cláusulas definidas como se muestra en la figura 7.4.

Para mayor información sobre programación lógica, Prolog y su relación con gramáticas libres de contexto, puede encontrarse en [51] y [52].

[4]Existe un unificador más general (mgu) θ. Para más información sobre *mgu*, ver [51].

```
expr(A, B, C) :- term(D, B, E), E = [+|F], expr(G, F, C), A is D + G.
expr(A, B, C) :- term(A, B, C).

term(A, B, C) :- factor(D, B, E), E = [*|F], term(G, F, C), A is D * G.
term(A, B, C) :- factor(A, B, C).

factor(A, [A|B], B) :- integer(A).
```

Figura 7.4: Traducción de la DGC de la figura 7.3.

7.3.3. Gramáticas de atributos y programación orientada a objetos

La programación orientada a objetos permite la implementación elegante de gramáticas de atributos.

En principio la gramática libre de contexto subyacente permite que su descripción en un lenguaje orientado a objetos permita capturar el árbol de jerarquía de categorías sintácticas definidas implícitamente entre los símbolos no terminales. Dicha jerarquía se representa mediante herencia. El polimorfismo basado en herencia (subtipos + sobrecarga) permite definir operaciones genéricas (como evaluación de los atributos) en forma natural.

Los conceptos subyacentes de la programación orientada a objetos, como clases, herencia, sobrecarga y redefinición de operaciones, han inspirado propuestas de extensiones de las gramáticas de atributos clásicas como las *Object Oriented context-free gramars* con sus extensiones naturales a *Object Oriented Attribute Grammars*(como las propuestas por Kai Koskimies en [2]).

A continuación se describe un posible esquema de traducción de una gramática de atributos GA a programas orientados a objetos. La evaluación procede bajo demanda.

- La clase base *ASTnode* sería una representación (abstracta) de un nodo del AST (abstract syntax tree). Su único atributo es una referencia al nodo padre.
- Cada no terminal X de la gramática tendría su clase correspondiente, la cual deriva de *ASTnode*. Los atributos sintetizados de X se representarían como métodos sin argumentos cuya implementación refleja la regla semántica que lo define. Los atributos heredados se representan también como métodos sin parámetros cuya implementación retornará el resultado de una invocación a un método del nodo padre que definirá su valor.

  Este enfoque implementa una GA puramente funcional.
- Cada producción de la forma $p : X_0 \rightarrow X_1 \ldots X_{p_n}$ se representa con una clase que contiene como atributos su parte izquierda (*left hand side (lhs)*) y un atributo para cada ocurrencia de símbolos de la parte derecha (*rhs*).

  Una clase que representa una producción contiene métodos que representan las ecuaciones que definen los atributos sintetizados de X_0 y los heredados de los símbolos X_i, $(1 \leq i \leq p_n)$.

La implementación de los métodos que representan las ecuaciones de los atributos heredados de los símbolos de la parte derecha de la producción, tienen como (único) argumento una referencia al nodo (instancia de un símbolo). Este argumento permite implementar en un único método las diferentes funciones semánticas que definen diferentes instancias del mismo atributo heredado para diferentes instancias del mismo símbolo que puede aparecer en el *rhs* de la producción p.

Dado un *AST*, la evaluación de cada atributo sintetizado v de la raíz se realiza mediante la invocación al método correspondiente *root.v()*. El orden de evaluación está implícito en la composición funcional (invocaciones a métodos). Si la *GA* no es circular la composición será finita y finalizará retornando el valor final.

Por supuesto, una implementación debería incluir un test de circularidad previo a la evaluación.

p_0: $N \rightarrow L$
attribution
$N.val = L.val$
$L.pos = 0$
end

p_1: $L \rightarrow L\ B$
attribution
$B.pos = L_0.pos$
$L_1.pos = L_0.pos + 1$
$L_0.val = B.val + L_1.val$
end

p_2: $L \rightarrow B$
attribution
$B.pos = L_0.pos$
$L_0.val = B.val$
end

p_3: $B \rightarrow$ *'0'*
attribution
$B.val = 0$
end

p_4: $B \rightarrow$ *'1'*
attribution
$B.val = 2^{B.pos}$
end

Figura 7.5: GA que obtiene el valor entero de un número binario.

La figura 7.5 muestra una GA que computa el valor de un número binario. La figura 7.6 su correspondiente implementación en un lenguaje orientado a objetos (C++).

7.4. Otros formalismos relacionados

En la bibliografía sobre gramáticas de atributos se encuentran algunos formalismos, que se describen brevemente a continuación y que en general son instancias, variaciones o subfamilias de las Gramáticas de Atributos de Alto Orden o de gramáticas de

```
class ASTnode {
   protected:
    virtual int L_pos(ASTnode & rhs_instance) {};
    virtual int B_pos(ASTnode & rhs_instance) {};
      ASTnode * parent;
};
class N : public ASTnode {};
class L : public ASTNode {};
class B : public ASTnode {};

class P0 : public N { // p0: N --> L
   public:
      P0(L p1) rhs1(p1) { rhs1.parent = this; }; // Constructor
      int val() { return rhs1.val(); }; // Syntetized attribute of N
      int L_pos(ASTnode & rhs_instance) { return 0; };// Inherit attribute of L
   private:
      L rhs1;
};

class P1 : public L { // p1: L --> L B
   public:
      P1(L p1,B p2) rhs1(p1) rhs2(p2)  { rhs1.parent = rhs1.parent = this; };
      int L_pos(ASTnode & rhs_instance)  { return parent->L_pos(rhs1) + 1; };
      int B_pos(ASTnode & rhs_instance)  { return parent->L_pos(rhs2); };
      int pos() { return parent->L_pos(*this); };
      int val() { return rhs1.val; };
   private:
      L rhs1;
      B rhs2;
};
...
class P4 : public B {      // p3: B --> '1'
   public:
      P4(One one) : rhs1(one) { rhs1.parent = this; };
      int pos() { return parent->B_pos(*this); };
      int val() { return 2 ** pos(); };
};
```

Figura 7.6: Implementación en un lenguaje orientado a objetos.

atributos con los dominios ampliados.

Las *gramáticas de atributos acopladas (Attribute Coupled Grammars)* introducidas por Ganzinger y Giegerich en 1984, en un intento de modelar el proceso de compilación de varias pasadas, pueden considerarse como una aplicación limitada de las HAG's. Esta familia de gramáticas permiten que un atributo sintetizado sea un árbol el cual será atribuido nuevamente, lo que hace, en definitva que sean HAG's con la restricción que NTA puede sólo ser instanciado al nivel mas externo.

Las *Affix Grammars*, desarrolladas pensando en aplicaciones para procesamiento de lenguaje natural, en 1962 y formalizadas en 1970, pertenecen a la familia de gramáticas de dos niveles. El nivel mas bajo consiste en un esquema de una gramática libre de contexto extendida con metavariables (*affixes*) y el segundo nivel define los dominios de esas metavariables. Los símbolos del primer nivel generalmente representan operaciones.

Las *Extended Affix Grammars*, introducidas por Koster en 1991, permiten que el dominio de las metavariables sean expresiones. Estas expresiones generalmente toman la forma de predicados, asemejándose a predicados en notación Prolog.

Lo interesante de este formalismo es que es autocontenido, es decir, su semántica está basada en sistemas de reescritura y no requiere otro sistema formal para las funciones u operaciones.

Para mayor información sobre estas extensiones, ver [2].

Capítulo 8

Herramientas basadas en gramáticas de atributos

En este capitulo se describen varias herramientas basadas en GA, las cuales se han utilizado mas ampliamente.

8.1. YACC: Yet Another Compiler-Compiler

Las herramientas *LEX* (LEXical analyzer geneator) y *YACC* (Yet Another Compiler-Compiler) han sido ampliamente utilizadas tanto en el ámbito académico como en la industria. Desarrollados en los laboratorios de At&T, estas herramientas se encontraban comúnmente en los sistemas tipo UNIX.

LEX es un generador de reconocedores de lenguajes regulares que se basa en una especificación del lenguaje regular como la unión de diferentes lenguajes expresados por medio de *expresiones regulares*.

A cada expresión regular especificada se asocia una acción (o bloque de código) a ejecutarse en el momento cuando se reconozca una cadena que sea descripta por esas expresión.

La herramienta genera un programa C que implementa un autómata finito determinístico. El diseño de Lex fue pensado en su uso integrado con YACC, aunque es posible su uso en forma aislada.

YACC es un generador de parsers que permite asociar atributos a los símbolos y acciones a cada prducción de la gramática.

La familia de gramáticas que *YACC* acepta son las LALR ya que genera un algoritmo de parsing *shift-reduce*.

YACC ejecuta las acciones asociadas a cada regla en el momento de la reducción del símbolo que precede la acción. Este mecanismo restringe la evaluación haciendo que la familia de gramáticas de atributos aceptada sea del tipo *l-atribuidas*.

En una especificación *YACC* es dificultoso utilizad atributos heredados aunque se puede simular por el hecho que permite la declaración de variables globales con lo cual es posible realizar cualquier tipo de acción semántica.

En realidad *YACC* no es una herramienta realmente basada en gramáticas de atributos, sino mas bien en una extensión del formalismo conocido como *esquemas de traducción*, lo que permite que las acciones asociadas a las producciones no sean estrictamente funcionales permitiendo acciones generales con características imperativas con efectos colaterales y las acciones no son realmente un conjunto de ecuaciones.

Es posible especificar una GA si se restringen las acciones para que se puedan observar como un conjunto de ecuaciones, aunque en realidad serán una secuencia.

Inicialmente, las primeras versiones de *YACC* generaban sólo código C. Actualmente, prácticamente existen versiones de *LEX* y *YACC* para cada lenguaje de programación que se ha definido.

Estas herramientas han sido ampliamente utilizadas para el desarrollo de frontends de intérpretes, compiladores y otras herramientas de procesamiento de lenguajes formales.

8.2. FNC-2 Attribute Grammar System

FNC-2 [56] es un sistema de procesamiento de gramáticas de atributos que soporta gramáticas de atributos absolutamente no circulares, que tiene como objetivos la facilidad de su uso y la eficiencia.

Ha sido desarrollado en el INRIA en un proyecto liderado por Martin Jourdan y Didier Parigot.

El lenguaje de especificación utilizado es modular, lo que permite describir en forma separada diferentes aspectos y pasos de un procesador de lenguajes.

Generalmente el trabajo de una herramienta de procesamiento de lenguajes se divide en pasos o etapas[1].

FNC-2 permite la descripción de cada paso, por lo que el procesador se conoce como *gramáticas de atributos acopladas* ya que el evaluador toma como entrada un árbol atribuido y tiene como salida otro árbol atribuido.

El sistema tiene diferentes lenguajes para las diferentes definiciones:

- Descripción de la sintaxis concreta (descripción del *léxico* y de la gramática libre de contexto mas la definición del árbol sintáctico abstracto), *atc.*

  Para esta tarea *FNC-2* utiliza dos herramientas, una basada en SYNTAX[2] y otra basada en *Lex* y *Yacc.*

- Los árboles intermedios son descriptos por gramáticas extendidas con declaraciones de los atributos utilizados. Estos árboles de denominan *Atributed Abstract Syntaxes (AASs)* utilizando el lenguaje *asx.*

- Las reglas de atribución se describen en *OLGA*, el cual es un lenguaje aplicativo puro sin ser un lenguaje funcional completo. OLGA es un lenguaje modular, lo

[1] Un compilador, por ejemplo, generalmente se divide en pasos: parsing, análisis semántico, generación de código, optimización, etc.

[2] Herramienta desarrollada por el INRIA.

que permite la definición de la GA en forma separada según los atributos de interés a computar.

Otro componente interesante de *FNC-2* es un generador de *unparsers* de árboles sintácticos atribuídos, denominado *ppat*. La salida está basada en la noción de cajas anidadas de texto, al estilo de TEX.

Finalmente, existe una herramienta que permite incluir los módulos que componen el sistema el cual funciona como una especie de *make*, denominado *mkfcn-2*.

8.2.1. Descripción interna

El evaluador de atributos del sistema *FNC-2* se basa en secuencias de visita. Las gramáticas absolutamente no circulares son transformadas a *l-orderded*. Esta transformación se realiza como se describe a continuación:

1. Se computan los grafos *In* y *Out* de cada producción.
2. En una pasada de arriba hacia abajo sobre las producciones de la gramática, se computa para cada no-terminal un conjunto de particiones totalmente ordenadas de sus atributos, es decir:
   a) Tomar una partición totalmente ordenada de los atributos del símbolo de la parte izquierda de la producción, sea π_{X_0}.
   b) Realizar la unión con el grafo de dependencias de la producción y los argumentos seleccionados de los no-terminales de la parte derecha de la producción, sean π_{X_i}, con $1 \leq i \leq k$ (k es el número de no-terminales de la parte derecha de la producción).
   c) Realizar un orden topológico del grafo resultante. Esto determina una partición totalmente ordenada de los atributos de cada símbolo no-terminal de la parte derecha de la producción. Se memoriza el mapping $(P_n, \pi_{X_0}, \ldots, \pi_{X_k})$
3. La GA resultante tiene como no-terminales los pares (X, π_{X_i}) y las producciones se construyen según los mappings memorizados.

Se debe hacer notar que la GA resultante no necesariamente deberá ser construída explícitamente, sino que se puede construir el evaluador basado en secuencias de visita, en el cual la operación *VISIT* tiene un argumento adicional que identifica la partición a utilizar en el nodo visitado.

El evaluador desarrollado en este trabajo, para las AG no circulares, es similar en este aspecto.

En [2], Martin Jourdan y Didier Parigot, describen detalles internos de el sistema *FNC-2*.

8.3. TOOLS

El sistema TOOLS (Translator for Object-Oriented Language Specifications) tiene características orientadas a objetos e incluye los conceptos descriptos en el capítulo 7.

Soporta gramáticas de atributos *S-atribuidas*. La información contextual (atributos heredados) se representa mediante objetos asociados al árbol sintáctico.

El método para desarrollar especificaciones en TOOLS se debe realizar en tres pasos:

1. Definición de la sintaxis (gramática libre de contexto).

2. Definición de las clases (objetos del lenguaje). Esto define una representación abstracta del lenguaje.

3. Especificación de la sintaxis con las clases. Esto implica extender los símbolos de la gramática con atributos sintetizados.

El uso de atributos sintetizados no limita las especificaciones porque generalmente la gramática de atributos sólo define la creación de los objetos. Para hacer procesamiento adicional se utiliza la activación de los objetos creados (por medio de la invocación de sus métodos).

La figura 8.3 muestra una parte de un pequeño intérprete de expresiones.

```
...
type Expr = class
  value:Integer;
  sort Add:(left:Expr); op:Oper; right:Expr);
  begin
    execute left; execute right;
    if op=Plus then
      value := left.value+right.value
    else
      value := left.value-right.value
  end
end;
...
```

Figura 8.1: Un ejemplo en TOOLS.

8.3.1. Manejo de contextos

La información contextual, para lo cual son necesarios los atributos heredados, se maneja utilizando colecciones de objetos, las cuales son accedidos por un conjunto (*familia*) de objetos, los cuales estarán asociados a los correspondientes no terminales de la gramática libre de contexto. De esta forma, esos objetos constituyen un ambiente no local accedido por medio de referencias y pueden ser accedidos globalmente de la especificación, generalmente especificando su identificador de objeto (los objetos en TOOLS pueden tener *nombres*, generalmente strings).

Este mecanismo permite definir en forma muy simple, ambientes, máquinas abstractas, etc.

TOOLS se describe en detalle en [2] por Kai Koskimies.

8.4. ELI

La construcción de compiladores ha sido un área muy desarrollada en las ciencias de la computación. Es un área de ingeniería de software el cual ha tenido un gran éxito desde el punto de vista que es posible generar un producto de software completamente en base a especificaciones formales.

Para lidiar con la complejidad de un compilador, es conveniente separar las diferentes tareas a realizar en pasos bien definidos. En general las etapas en que se puede dividir un compilador son las siguientes:

1. **Estructuración**: el análisis léxico, el cual es responsable de generar una secuencia de tokens y el análisis sintáctico, el cual es el encargado de analizar la estructura de las frases y su salida es un árbol de sintaxis.

2. **Traducción**: este paso generalmente se encarga de hacer los análisis semánticos, como por ejemplo chequeo de tipos, análisis de alcance de identificadores, identificación de operadores, representación de datos y de acciones. En esta etapa, generalmente se utilizan gramáticas de atributos para su especificación.

   La salida producida en esta etapa generalmente es una representación intermedia del programa.

3. **Codificación**: En esta etapa, la cual puede comprender varias subetapas, se realizan optimizaciones y la generación de código.

ELI tiene como objetivo principal que cada una de estas etapas pueda ser generadas a partir de especificaciones. Para cada etapa se definen lenguajes de especificación y brinda funciones de biblioteca especializadas.
Las principales especificaciones que se pueden realizar en **ELI**, son:

1. **GLA**: especificación del análisis léxico.

2. **PGS** y **Cola**: especificación de análisis sintáctico. *PGS* genera parsers LALR y *Cola* genera parsers LL(1).

3. **GAG** y **LIGA**: son dos generadores de evaluadores de gramáticas de atributos.

4. **PTG**: es un traductor de código fuente a otro código fuente (*Program Text Generator*).

El sistema tiene algunas características muy interesantes para que el usuario no tenga que realizar sobreespecificaciones. Un ejemplo es que es posible que en la especificación de una gramática de atributos no sea necesario realizar computaciones de encadenamientos (*chains*), ya que el sistema genera automáticamente tales reglas.

El lenguaje *LIDO*[3] tiene algunos operadores sobre los atributos que hacen mas cómodas y compactas las especificaciones. A modo de ejemplo, la figura 8.2 muestra una especificación de la sintaxis concreta y la figura 8.3 muestra una especificación *LIGA* de inferencia de tipos en expresiones.

[3]LIDO es el lenguaje de especificación del sistema LIGA.

```
p1:  Expr   ::= Expr AddOpr Fact
p2:  Expr   ::= Fact
p3:  Fact   ::= Fact MulOpr Term
p4:  Fact   ::= Term
p5:  Term   ::= '(' Expr ')'
p6:  Term   ::= Ident
p7:  AddOpr ::= '+'
p8:  AddOpr ::= '-'
p9:  MulOpr ::= '*'
p10: MulOpr ::= '/'
```

Figura 8.2: Especificación ELI de la sintaxis concreta de una expresión.

```
RULE ap1: Expr ::= Expr Opr Expr
STATIC
  Opr.type := result_type(Opr.op,Expr[1].type,Expr[2].type);
  message_if(EQ(Opr.type,InvalidType),"Incompatible types");
  ...
  Expr[1].type := OILOprType(Opr.target
END
```

Figura 8.3: Especificación ELI de una GA de inferencia de tipos en expresiones.

En las figuras 8.2 y 8.3 se puede notar que en la descripción de la gramática de atributos se basa en la sintaxis abstracta más que en la concreta. *ELI* realiza esta transformación en forma automática computando clases de equivalencia entre los símbolos de la gramática concreta. En el ejemplo dado, los símbolos $\{Expr, Factor, Term\}$ pertenecen a la misma clase de equivalencia, como también los símbolos $\{Opr, AddOpr, MulOpr\}$.

Entre los operadores de alcance de LIGA podemos mencionar *INCLUDING* y *CONSTITUENTS* los cuales permiten alcanzar instancias de atributos más allá del alcance de la regla en que aparecen.

El operador *INCLUDING attrs* accede a la instancia más próxima en los contextos (superiores) de los símbolos que aparecen en su lista asociada[4].

El operador *CONSTITUENTS symbol.attr* permite el acceso a instancias en subárboles de la regla en la cual aparece.

Mas información sobre detalles del sistema ELI en [14].

8.5. AGCC (Attribute Grammars Compiler-Compiler)

En el marco de desarrollo de esta tesis y con la finalidad de experimentar con métodos de evaluación y aportar con una herramienta utilizable en la práctica se desarrolló *agcc*.

Uno de los principales objetivos en su desarrollo fue el soporte a la familia de gramáticas de atributos bien definidas no circulares y que el lenguaje de especificación

[4] *attrs* es una lista de elementos de la forma *símbolo.attributo*.

```
ATTRIBUTE GRAMMAR
  declaration-example

TYPE D_TYPE={INT,CHAR};

SEMANTIC DOMAINS:
  infix +: (int,int) -> int;
  size: D_TYPE -> int;
END

SYNTHETIZED ATTRIBUTES:
  m: int of id, lid, decl
  t: D_TYPE of type, decl
END

INHERIT ATTRIBUTES:
  t: D_TYPE of id, lid, decl
END

BEGIN GRAMMAR:

RULE r1: decl -> type lid
         COMPUTE
            decl.t=type.t
            decl.m=lid.m
            lid.t=type.t
END

RULE r2: type -> integer
         COMPUTE
           type.t=INTEGER
END

RULE r3: type -> character
         COMPUTE
           type.t=CHARACTER
END

RULE r4: lid -> id
         COMPUTE
           id.t=lid.t
           lid.m=size(lid.t)
END

RULE r5: lid -> lid id
         COMPUTE
           id.t=lid[0].t
           lid[1]=lid[0].t
           lid[0].m=lid[1].m + size(lid[0].t)
END

END ATTRIBUTE GRAMMAR
```

Figura 8.4: Ejemplo de una especificación **agcc**

sea independiente del lenguaje de programación de destino del generador de código.

La primera versión de *agcc* generaba tanto el analizador sintáctico como el analizador semántico. En la versión actual se ha separado la etapa de generación de analizador sintáctico del analizador semántico. De esta forma el usuario tiene la libertad de usar su herramienta preferida de análisis sintáctico, la cual debe generar un AST (abstrac syntax tree). La especificación de la GA se debe basar en la sintaxis abstracta (AST) y *agcc* genera el evaluador semántico correspondiente.

La figura 8.4 muestra un ejemplo de una especificación que describe la computación de la memoria requerida por los identificadores de una declaración.

Una especificación contiene las siguientes secciones:

1. Definiciones de tipos de datos usados. El ejemplo muestra el uso de una enumeración.

2. Declaración de los dominios sintáticos.

3. Declaración de los atributos sintetizados y heredados de cada símbolo.

4. Definiciones de las reglas de la gramática libre de contexto y sus reglas de atribución.

La sección de declaración de los dominios semánticos describen funciones no interpretadas. Su definición es requerida a sólo efectos de la realización de verificación de

tipos de las reglas semánticas.

AGCC realiza el análisis de la especificación y realiza las siguientes funciones:

- Realiza el test de circularidad de la especificación.
- Genera una representación abstracta de la GA.
- Genera el grafo de dependencias directas de cada producción ($DP(p)$).
- Genera planes de evaluación.
- Genera un evaluador de la GA utilizando herramientas externas del sistema.

Las salidas de la herramienta pueden ser la entrada de herramientas externas, como por ejemplo intérpretes de evaluación, generadores de código o visualizadores de grafos.

La sintaxis de las diferentes salidas producidas por *agcc* tienen un formato basado en XML. Se utilizan formatos ad-hoc (definidos por sus correspondientes DTDs) para cada clase de salida. Por ejemplo, la salida de los grafos de dependencias directas de cada producción se codifican en *GXL*[48] (*http://www.gupro.de/GXL/index.html*), mientras que las reglas semánticas se representan utilizando las técnicas descriptas en [31] (srcML) para representar código fuente.

El formato XML puede ser fácilmente reconocido por parsers determinísticos y permite que diferentes módulos puedan intercambiar información. Las salidas de *agcc* también puede ser codificadas en *Aterms* (ver [10]). Los *Aterms* pueden representarse en forma muy compacta y existen bibliotecas para su manipulación en diferentes lenguajes de programación [11].

Este diseño permite la construcción modular de herramientas adicionales para la construcción de procesadores basados en gramáticas de atributos.

Para mayor información sobre el diseño e implementación de la versión original de *agcc*, ver [9]. En *http://dc.exa.unrc.edu.ar/docentes/marroyo* se puede encontrar mas información sobre las últimas versiones de *agcc* y su documentación correspondiente.

8.6. Otras herramientas

Existen una gran variedad de herramientas basadas en gramáticas de atributos.

Por mencionar algunos que han sido relevantes en algún momento, es posible hacer una breve descripción de los siguientes sistemas:

- **OPTRAN**: es un sistema en lotes (batch) basado en un lenguaje de transformaciones de árboles. Los árboles son atribuidos y una GA es utilizada para la definición de su valuación. La familia de GAs que acepta son las *absolutamente no circulares*.

  El sistema fue escrito en Pascal y genera código Pascal.

- **SSCC**: es una herramienta basada en las *gramáticas de atributos ordenadas extedidas* descriptas anteriormente en éste trabajo por Wuu Yang. El evaluador es una implementación del algoritmo descripto. Para mayor información ver el reporte técnico en [46].

- **UUAG** (Utrecht University Attribute Grammar system): es un sistema que toma como entrada una AG y genera un programa Haskell. El lenguaje de especificación permite definir tipos abstractos de datos y no tiene limitaciones sobre la AG. UUAG aprovecha la evaluación lazy de Haskell para evaluar GA circulares.

  Para mayor información sobre UUAG, ver la referencia bibliográfica [39].

- **Silver**: es una herramienta basada en gramáticas de atributos extensibles[13]. Una GA puede ser extendida definiendo nuevas producciones y nuevas reglas semánticas (bajo ciertas condiciones) o redefiniendo reglas de atribución. La idea es similar al concepto de herencia en los lenguajes orientados a objetos.

  De esta manera, una gramática de atributos extensible permite modularizar especificaciones lo cual es útil para la generación de herramientas basadas en lenguajes formales.

  Para mayoy información sobre *Silver*, ver [38].

Capítulo 9

Conclusiones y trabajo futuro

En este trabajo se ha presentado un análisis de las gramáticas de atributos, sus características, clasificación y sus extensiones. Se han presentado los últimos avances en el tema y se han analizado diferentes métodos de evaluación y los problemas relacionados en términos de complejidad computacional.

Se ha presentado un análisis de las principales herramientas existentes basadas en gramáticas de atributos y sus extensiones.

Se han desarrollado algoritmos de evaluación estáticos y dinámicos y se ha propuesto un evaluador de atributos (NCEval) concurrente basado en secuencias de visita cuya partición de atributos a evaluar por cada proceso se genera estáticamente. El algoritmo es una extensión de la propuesta por Wuu Yang adaptado a la familia de GA NC(1).

Hasta el momento no se conocen herramientas basadas en gramáticas de atributos que usen estas técnicas.

NCEval y *agcc* son las principales constribuciones de esta tesis.

Si bien las gramáticas de atributos han perdido la atención que tuvieron en los primeros años de su desarrollo, en la actualidad han resurgido, principalmente en el desarrollo de extensiones y en el estudio de sus aplicaciones.

Las limitaciones en cuanto a la eficiencia de su evaluación, van siendo superados por los avances en el poder computacional de las computadoras actuales, tanto en velocidad de procesamiento, como en capacidad de memoria.

Como se vio en el marco de este trabajo, la complejidad de los principales algoritmos, como por ejemplo el test de circularidad, ha dejado de ser un problema no tratable en la práctica en la actualidad.

Se pretende seguir trabajando en el tema, principalmente en el aspecto del desarrollo de herramientas de especificación de procesadores de lenguajes y en las otras aplicaciones de las gramáticas de atributos para la resolución de problemas, como pueden ser el desarrollo de prototipos de software y como herramienta básica para la definición de sistemas *verificables* por medio de la comparación de las funcionalidades de un sistema con la semántica definida por una gramática de atributos.

Este último tema es uno de los principales aspectos de su estudio en varios proyectos de investigación actuales.

Actualmente hay un creciente trabajo en investigaciones sobre lenguajes (o meta-lenguajes) de propósitos específicos (Domain Specific Languages)[37] en los cuales las gramáticas de atributos aparecen como la herramienta fundamental (y natural) para la especificación sintáctica y semántica.

Además se están realizando esfuerzos en el uso de las gramáticas de atributos para el desarrollo de herramientas o ambientes basados en lenguajes. A modo de ejemplo, se ha desarrollado una herramienta de verificación para el lenguaje Oberon, la cual se basa en la transformación de predicados en acciones de programas al estilo de la lógica de Hoare.

La tendencia actual para el desarrollo de intérpretes y compiladores se centra en el uso de ambientes de construcción específicos. En este campo hay varios esfuerzos en las *Gramáticas de Atributos Extensibles (EAG)*, las cuales permiten integrar en un único formalismo, componentes descriptos en forma separada[13].

Además de las aplicacciones tradicionales, las gramáticas de atributos están siendo aplicadas en diversos dominios de aplicación, algunos de los cuales se enumeran a continuación:

1. Generación de programas eficientes[27].
2. Eliminación de ambigüedades en gramáticas libres de contexto ambiguas[40].
3. Reconocimiento de lenguaje natural[50].
4. Herramientas de análisis estático de programas[30].
5. Desarrollo de lenguajes de propósitos especiales[49],[37].

y para la especificación y construcción de procesadores de lenguajes en general[34].

Como trabajo futuro de aplicación de GA, el autor de esta tesis tiene proyectado el desarrollo de un entorno de especificación y generación de ambientes de procesadores de lenguajes similar al proyecto *The Meta-Environment*[57] utilizando *agcc*.

El proyecto *The Meta-Environment* se basa en sistemas de reescritura mas que en gramáticas de atributos y permite el desarrollo de IDE's basados en lenguajes definidos y la generación de procesadores de lenguajes. En [32] se describe el proyecto y el conjunto de formalismos y herramientas utilizadas.

Finalmente, Wouter Swierstra, en [58], describe claramente la importancia de las gramáticas de atributos.

Bibliografía

[1] H. Alblas. *Iteration of Transformation passes over attributed program trees.* Acta Informatica 27, 1989. pp: 1-40.

[2] H. Alblas, B. Melichar (Ed.).*Attribute Grammars. Applications and Systems.* International Summer School SAGA, Prague. Junio 1991. Proceedings Springer-Verlag. I.S.B.N.: 3-540-54572-7.

[3] Aguirre, Grispan. 1998. *Un parser TOP-DOWN no recursivo modificado para la evaluaci'on de atributos.*. WAIT 98. 1998.

[4] Aguirre, Arroyo, Grinspan. 1999. *Un ambiente de ejecución de procesadores de lenguajes orientado a objetos basado en gramáticas de atributos.*

[5] Aho, Sethi, Ullman. *The theory of Parsing, Translation and Compiling.* Vol 1. and Vol. 2. Prntice Hall, Englewood Cliffs, N.J.. 1972.

[6] Aho, Sethi, Ullman. *Compilers, Concepts, Principles and Tools.* Addison-Wesley. 1985.

[7] Arroyo M., Aguirre J., Florio N. 2003. *Generación de evaluadores estáticos de gramáticas de atributos bien definidas.* CACIC 2003.

[8] Arroyo M., Aguirre J. Florio N. 2002. *Generación estática de evaluadores concurrentes para gramáticas de atributos NC(1).* CACIC 2002.

[9] Arroyo M., Aguirre J., Alarcón R. *agcc: un generador de procesadores de lenguajes basado en gramáticas de atributos.* XI Congreso Argentino de Ciencias de la Computación, CACIC 2005. Concordia, Entre Ríos, Argentina.

[10] Brand, M.G.J. van den, H.A. de Jong, P. Klint, and P.A. Olivier. *Efficient Annotated Terms.* Software – Practice & Experience 30:259–291. 2000.

[11] Brand, M.G.J. van den and P. Klint. *ATerms for manipulation and exchange of structured data: It's all about sharing.* Information and Software Technology. 49:55–64. 2007.

[12] Boyland J.T. 1996. *Conditional Attribute Grammars.* ACM Transactions on Programming Languages and Systems 18. pag. 73-108.

[13] R. Marti, T. Murer. *Extensible Attribute Grammars.* TIK Report 92-6. 1992.

[14] Department of Electrical and Computer Engineering, University of Colorado. 1991. *Eli Documentation.* Technical Report, Boulder, California.

[15] Engelfriet J., Filé. 1982. *Simple multi-visit attribute grammars*. J. Comput. Syst. Sci. 24, 3, 283-314.

[16] Farrow R. *Automatic Generation of fixed-point-finding evaluators for circular attribute grammars*. Proc. SIGPLAN '86 Symposium on Compiler Construction. 1986. pp: 85-98.

[17] Grosch J. 1992. *Efficient Evaluation of Well Formed Attribute Grammars and Beyond*. GMD Forschungsstelle an der Universität Karlsruhe. Germany.

[18] J. Hopcroft, R. Motwani, J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Thirth Edition. Addison-Wesley. 2007. ISBN: 0321462254.

[19] *The Design and Construction of Compilers*. R. Hunter. John Wiley and Sons. 1981. ISBN: 978-0471280545.

[20] Jazayeri, Ogden and Rounds. 1975. *The intrinsically exponential complexity of the circularity problem for attribute grammars*. Comm. ACM 18. December 2, Pag: 697-706.

[21] . S.C. Johnson. *YACC: Yet Another Compiler-Compiler*. Manual. Murray Hill. N.J. 1975.

[22] T. Johnson. *Attribute Grammars as a Functional Programming Paradigm*. Chapter of book *Functional Programming Languages and Computer Architecture*. Pages:154-173. 1987.

[23] U. Kastens. 1980. *Ordered Attribute Grammars*. Acta Informatica. Vol. 13, pp. 229-256.

[24] Uwe Kastens. *Construction of Application Generators using Eli*. Proceedings ALEL'96 Workshop on Compiler Techniques for Application Domain Languages and Extensible Language Models, Technical Report LU-CS-TR:96-173. Lund University, Sweden, April 1996.

[25] E. Klein. 1992. *Parallel Ordered Attribute Grammars*. Proceedings of the 1992 International Conference on Computer Languages. pp. 106-16.

[26] D. Knuth. 1968. *Semantics of context free languages*. Math Systems Theory 2. June 2.Pag: 127-145.

[27] M.F. Kuiper and S.D. Swierstra. *Using attribute grammars to derive efficient functional programs*. Computing Science in the Netherlands CSN '87, November 1987.

[28] M. F. Kuiper. *Paralell Attribute Evaluation: Structure of Evaluators and Detection of Parallelism*. Attribute Grammars and their Applications (WAGA). Paris. P. Deransart & M. Jourdan, eds. Lecture Notes in Comp. Science 461. Springer-Verlag. 1990. Pag: 61-75.

[29] J. Levine, T. Mason, D. Brown. *Lex & YACC*. Second Edition. O' Reilly. 1992.

[30] Mads Rosendahl. *Abstract Interpretation Using Attribute Grammars*. Proceedings of the International Conference WAGA on Attribute Grammars and their Applications. Lencture Notes in Computer Science. pp: 143-156. 1990. Spriger Verlag, London, UK.

[31] Maletic, J.I., Collard, M., Marcus, A. *"Source Code Files as Structured Documents*. Proceedings of the the 10th IEEE International Workshop on Program Comprehension (IWPC 2002), Paris, France, June 26-29, 2002, pp.289-292.

[32] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. *The ASF+SDF Meta-Environment: a Component-Based Language Development Environment*. 365–370. Compiler Construction. Lecture Notes in Computer Science. 2001. Springer-Verlag.

[33] Thomas Noll, Heiko Vogler. 1992. *Top-Down Parsing with Simultaneous Evaluation of Noncircular Attribute Grammars*. Aachener Informatik-Berichte Nr. 92-14.

[34] Jukka Paakki. *Attribute grammar paradigms - a high-level methodology in language implementation*. ACM Computing Surveys. Vol 27, Issue 2. Pages: 196-255. 1995.

[35] Riis Nielson. 1983. *Computation sequences:a way of characterize classes of attribute grammars*. Acta Informática 19, pp. 225-268.

[36] D. Rushall. *An Attribute Grammar Evaluator in Haskell*. Technical report. University of Manchester. 1992.

[37] J. Saraiva, S. Schneider. *Embedding Domain Specific Languages in the Attribute Grammar Formalism*. Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9 - Volume 9. 2003.

[38] Eric Van Wyk, Derek Bodin, Jimin Gao, Jimin Gao. *Silver: An Extensible Attribute Grammar System*. Workshop on Language Descriptions, Tools, and Applications (LDTA). 2007.

[39] S. Doaitse Swierstra, Pablo R. Azero Alcocer, Joao Saraiva. *Designing and Implementing Combinator Languages*. Department of Computer Science, Utretcht University. The Netherlands.

[40] . Valentin David. *Attribute Grammars for C++ Disambiguation*. LRDE, 2004.

[41] Wuu Yang, W. C. Cheng. *A Polynomial Time Extension to Ordered Attribute Grammars*. Department of Computer and Information Science. National Chiao-Tung University, Hsin-Chu, Taiwan, R.O.C.

[42] Wuu-Yang. 1998. *Multi-Plan Attribute Grammars*. Department of Computer Information Science. National Chiao-Tung University, Hsin-Chu, Taiwan, R.O.C.

[43] Wuu-Yang. 1998. *Conditional Evaluation in Simple Multi-Visit Attribute Grammar Evaluators*. Department of Computer and Information Science. National Chiao-Tung University, Hsin-Chu, Taiwan, R.O.C.

[44] Wuu-Yang. 1999. *A Classification of Non Circular Attribute Grammars based on Lookahead behavior*. Department of Computer and Information Science. National Chiao-Tung University, Hsin-Chu, Taiwan, R.O.C.

[45] Wuu Yang. 1999. *A finest partitioning algorithm for attribute grammars*. 1999. Second Workshop on Attribute Grammars and their Applications - WAGA99. pp. 77-92.

[46] Wuu-Yang. 2001. *SSCC: A software Tool Based on Extended Ordered Attribute Grammars*. Technical report. Departament of Computer and Information Science. National Chiao-Tung University. HsinChu, Taiwan, R.O.C.

[47] Waite, W. and G. Goos. *Compiler Construction*. Springer Verlag. 1984.

[48] A. Winter, B. Kullbach, V. Riediger. *An Overview of the GXL Graph Exchange Language*. Springer Verlag: S. Diehl (ed.) Software Visualization · International Seminar Dagstuhl Castle, Germany, May 20-25, 2001 Revised Lectures.

[49] Eric Van Wyk. *Domain Specific Meta Languages*. Proceedings of the 2000 ACM symposium on Applied computing - Volume 2. Pages: 799 - 803. 2000.

[50] A. Zoltan, T. Gyimóthy, T. Horváth, K. Fábricz. *Attribute grammar specification for a natural language understanding interface*. Proceedings of the international conference on Attribute grammars and their applications. Paris, France. Pages: 313-326. Springer Verlag, New York, USA. 1990.

[51] Ulf Nilson, Jan Maluszynsky. *Logic, Programming and Prolog*. Second Edition. J. Wiley and Sons. 1995.

[52] P. Blackburn, J. Bos, K. Striegnitz. *Learn Prolog Now*. College publications. 2006.

[53] AntLR. URL: *http://www.antrl.org/*

[54] JavaCC. URL: *http://javacc.dev.java.net/*

[55] JavaCUP. URL: *http://www2.cs.tum.edu/projects/cup/*

[56] FNC-2 Attribute Grammar System.
URL: *http://www-sop.inria.fr/members/Didier.Parigot/www/fnc2/littlefnc2.html*

[57] Meta-Environment.
URL: *http://www.cwi.nl/htbin/sen1/twiki/bin/view/Meta-Environment/WebHome*.

[58] Wouter Swierstra. *Why Attribute Grammars Matter*.
URL: *http://www.haskell.org/tmrwiki/WhyAttributeGrammarsMatter*

Índice de figuras

Índice de cuadros

Índice de algoritmos