Sintaxis y Semántica de los lenguajes

Gramáticas con atributos

2020

Facultad Regional Delta,
Universidad Tecnológica Nacional

Prof. Juan Miguel Santos

Gramática con atributos.

- Atributo.
- Regla de valuación.
- Atributo sintetizado.
- Atributo heredado.
- Definición Dirigida por la Sintaxis (DDS).
- Gramática de atributos.
- Gramática L-atribuida.
- Gramática S-atribuida.
- Grafo de dependencias.
- Ordenamiento topológico.
- Esquema de traducción.

Definición (atributo):

Un atributo es una variable tipada asociada a un símbolo de una gramática.

Ejemplo: Sea la gramática $G = < V_N, V_T, P, S >$ con $V_N = \{ S, A, B, C \}$, definimos S.a como el atributo a del tipo entero asociado a A. Sea la cadena w y sea τ el árbol de derivación de w, a cada instancia de A en el árbol τ le corresponderá una instancia del atributo a.

Definición (regla de valuación):

Una regla de valuación es una función que relaciona a los atributos de los símbolos que intervienen en una producción.

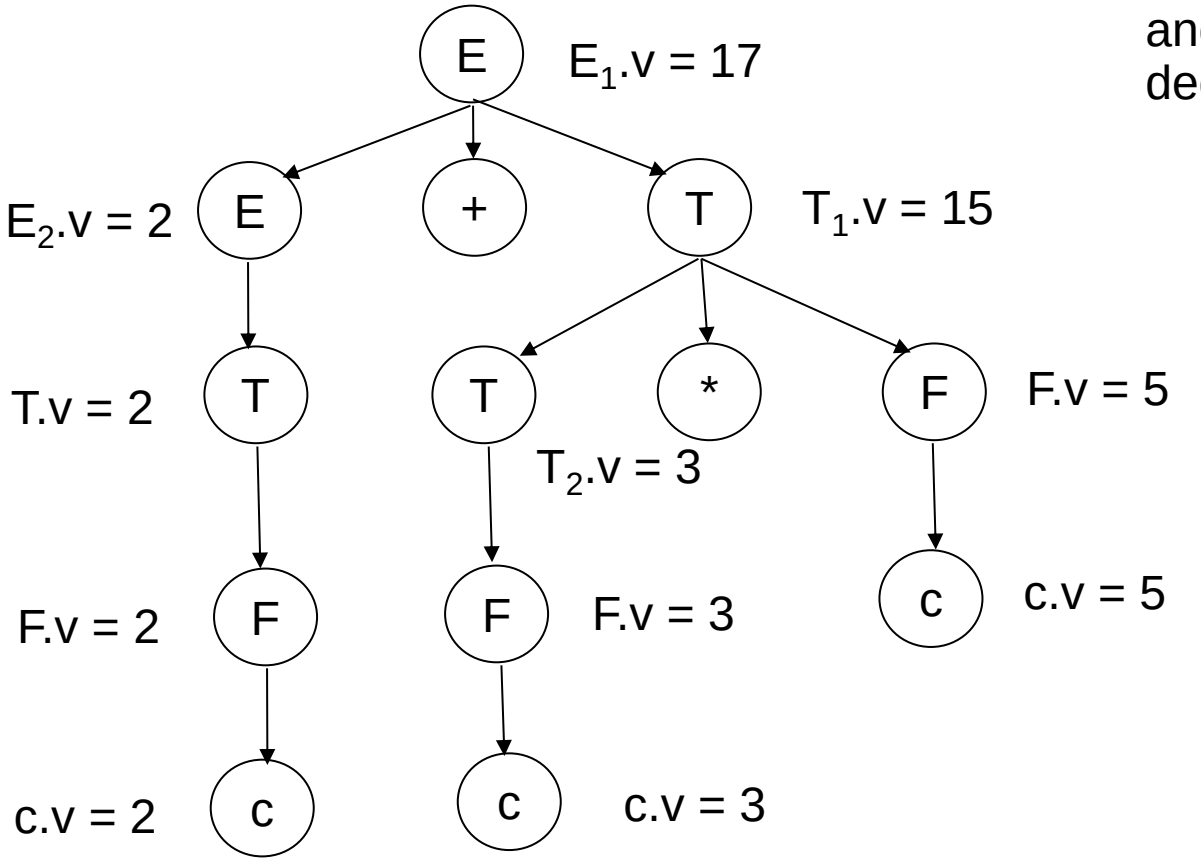
Ejemplo: Sea la gramática $G = < V_N, V_T, P, E >$ de las expresiones con
$P = \{ E \rightarrow E+T \ / \ T, T \rightarrow T * F \ / \ F, F \rightarrow (E) \ / \ c \}$,
definimos en cada símbolo de V_N un atributo para poder evaluar una expresión.

$E \rightarrow E+T$	$\{ E_1.v := E_2.v + T.v \}$
$E \rightarrow T$	$\{ E.v := T.v \}$
$T \rightarrow T*F$	$\{ T_1.v := T_2.v * F.v \}$
$T \rightarrow F$	$\{ T.v := F.v \}$
$F \rightarrow (E)$	$\{ F.v := E.v \}$
$F \rightarrow c$	$\{ F.v := c.v \}$

Notas:
1. El valor del atributo c.v es suministrado por el analizador lexicográfico.
2. Cuando en una producción aparece mas de una vez el mismo símbolo, se le agrega un subíndice para poder diferenciarlos.

El árbol de derivación de w = 2 + 3 * 5 es

Arbol de derivación con anotaciones o decorado.



Definición (atributo sintetizado):

Sea $G = < V_N, V_T, P, E >$ y $A \rightarrow \alpha$ una producción de P y sea
$A.a = f(\alpha_1.a, \alpha_2.a, .., \alpha_n.a)$ una regla de valuación
donde $\alpha = \alpha_1\alpha_2..\alpha_n$, $\alpha_i \in (V_N \cup V_T)$,
A.a es un atributo sintetizado ya que depende del valor de los atributos de los símbolos que están en la parte derecha de la producción (los hijos de A en el árbol).

Definición (atributo heredado):

Sea $G = < V_N, V_T, P, E >$ y $A \rightarrow \alpha$ una producción de P y sea
$\alpha_i.a = f(\alpha_1.a, \alpha_2.a, .., \alpha_n.a, A.a)$ una regla de valuación
donde $\alpha = \alpha_1\alpha_2..\alpha_n$, $\alpha_i \in (V_N \cup V_T)$,
$\alpha_i.a$ es un atributo heredado ya que depende del valor de cualquiera de los atributos de los símbolos en la producción (los hermanos o el padre en el árbol).

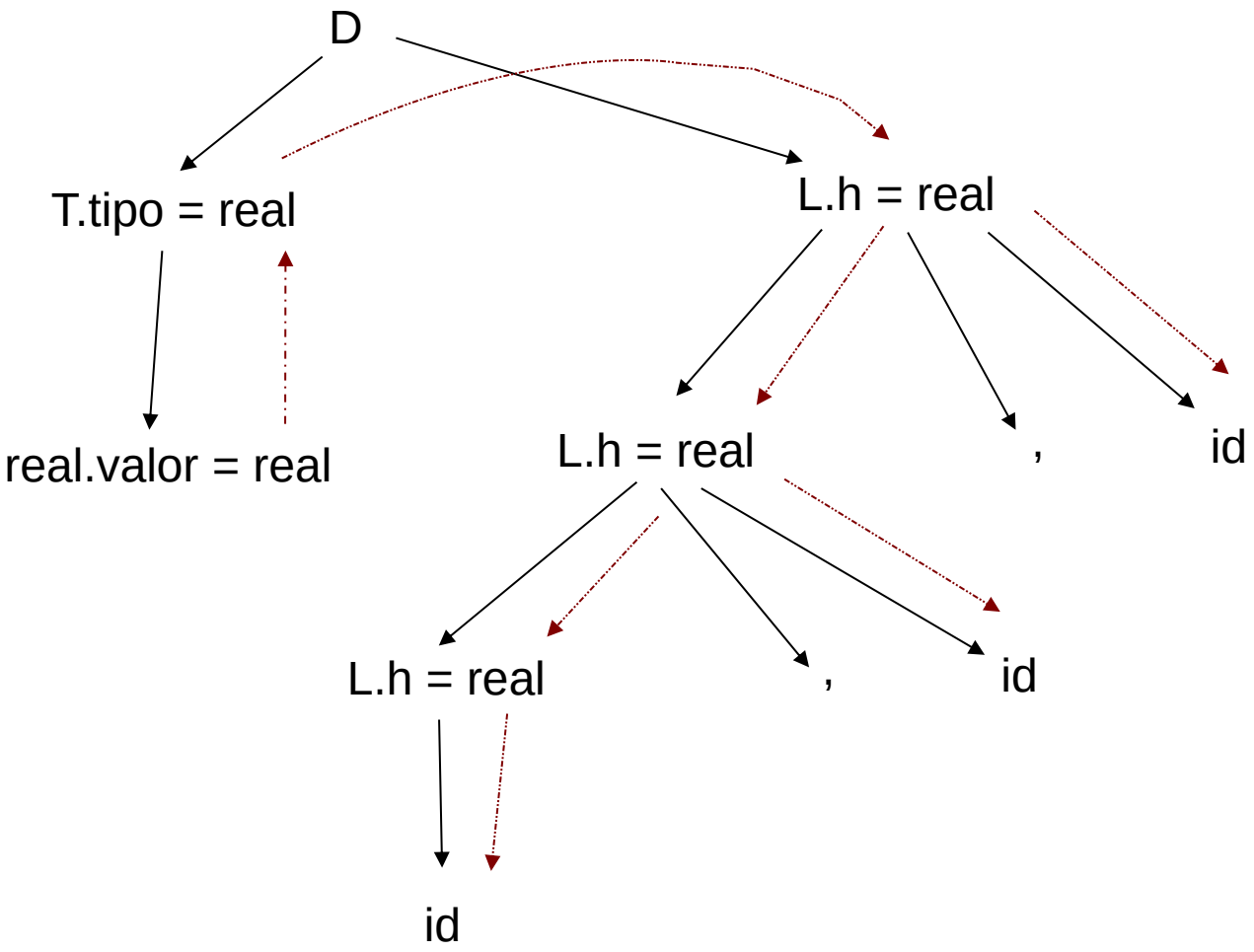
Ejemplo:

Sea G = < V_N, V_T, P, S > una gramática para la declaración de variables con

P = { $D \rightarrow T\ L$,
$T \rightarrow$ int,
$T \rightarrow$ real,
$L \rightarrow L$ “,” id,
$L \rightarrow$ id }

Se proponen los siguientes atributos y semántica para obtener el tipo de cada variable en la declaración:

$D \rightarrow T\ L$	{ L.h := T.tipo }
$T \rightarrow$ int	{ T.tipo := integer }
$T \rightarrow$ real	{ T.tipo := real }
$L \rightarrow L$ “,” id	{ L_2.h := L_1.h; Actualizar_tipo(id.nombre, L_1.h) }
$L \rightarrow$ id	{ Actualizar_tipo(id.nombre, L.h) }


D
T.tipo = real
L.h = real
real.valor = real
L.h = real
,
id
L.h = real
,
id
id

Definición (**Definición dirigida por la sintaxis - DDS**):

Es una generalización de una gramática en la que cada símbolo de la misma puede tener uno o mas atributos. Cada producción de la gramática puede tener una o más reglas de valuación que permiten valuar los atributos en la misma (sintetizados o heredados).

Definición (**Gramática de atributos**):

Es una Definición dirigida por la sintaxis donde las reglas usadas en la semántica NO tienen efectos globales (por ejemplo, actualización de una variable global, imprimir en alguna salida, manejar una tabla, etc..

Definición (**Gramática S-atribuída**):

Es una DDS en la cual sólo están definidos atributos sintetizados.

Grafo de dependencias (GDD)
Permite definir el orden de valuación de los atributos basado en las reglas de valuación.

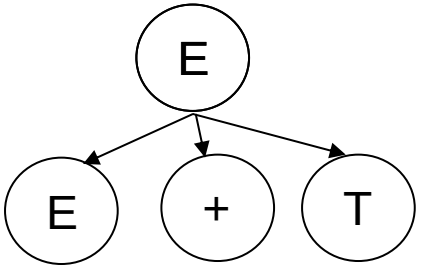
Construcción del grafo de dependencias para un árbol de derivación.

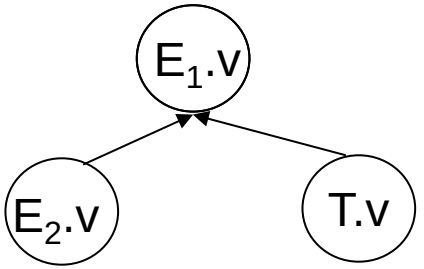
Para cada nodo **x** en el árbol de derivación
    Para cada atributo **a** del símbolo **x**
        Construir un nodo para **x.a** en el GDD

Para cada nodo **x** en el árbol de derivación
    Para cada regla de valuación $v := f(v_1, v_2,.., v_k)$ asociada con la producción
    que dio lugar a **x** y sus hijos en el árbol
        Para i desde 1 hasta k
            Construir un *arco dirigido* desde v_i hasta v.

Ejemplo:

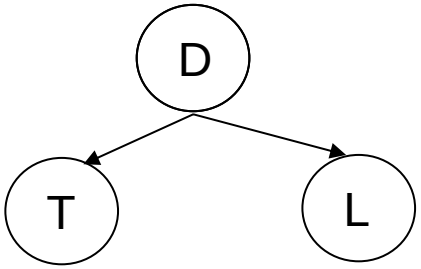
a) En $E \rightarrow E + T$ $\{E_1.v := E_2.v + T.v\}$

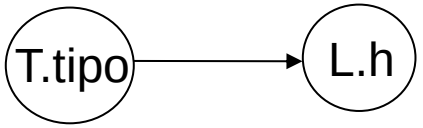




Ejemplo:

a) En $D \rightarrow T\ L$ $\{L.h := T.tipo\}$





Para construir una valuación:

1. Construir el árbol de derivación
2. Construir el grafo de dependencias
3. Seguir el o los ordenamientos topológicos y hacer la valuación

<u>Definición</u> (**ordenamiento topológico**):

Dado un grafo dirigido con nodos n_1, n_2, .., n_N,

$(n_{j1}, n_{j2}, .., n_{jk})$, $j_i \in [1, n]$, $i \in [1, k]$, $k \leq n$

es un ordenamiento topológico si ocurre que para todo $l < k$, el nodo n_{jl} tiene un arco dirigido a n_{jl+1} y el camino entre n_{j1} y n_{jk} es sin ciclos.

Definición (**gramática L-atribuída**):

Una DDS es una definición con atributos por la izquierda (o L-atribuída) si cada atributo heredado de X_j, $1 \leq j \leq n$, del lado derecho de $A \rightarrow X_1X_2..X_n$ depende de
1. los atributos de los símbolos X_1, X_2, .., X_{j-1} a la izquierda de X_j,
2. los atributos de A.

Nota: Fijarse que toda DDS con atributos sintetizados (S-atribuída) es una definición de atributos por la izquierda pues las restricciones 1 y 2 sólo se refieren a atributos sintetizados.

Ejemplo:

$A \rightarrow LM$ { $L.h := f_1(A.h)$; $M.h := f_2(L.s)$; $A.s := f_3(M.s)$ }
$A \rightarrow QR$ { $R.h := r(A.h)$; **$Q.h := q(R.s)$**; $A.s := f_4(Q.s)$ }

no es una gramática L-atribuída.

Definición (**esquema de traducción**):

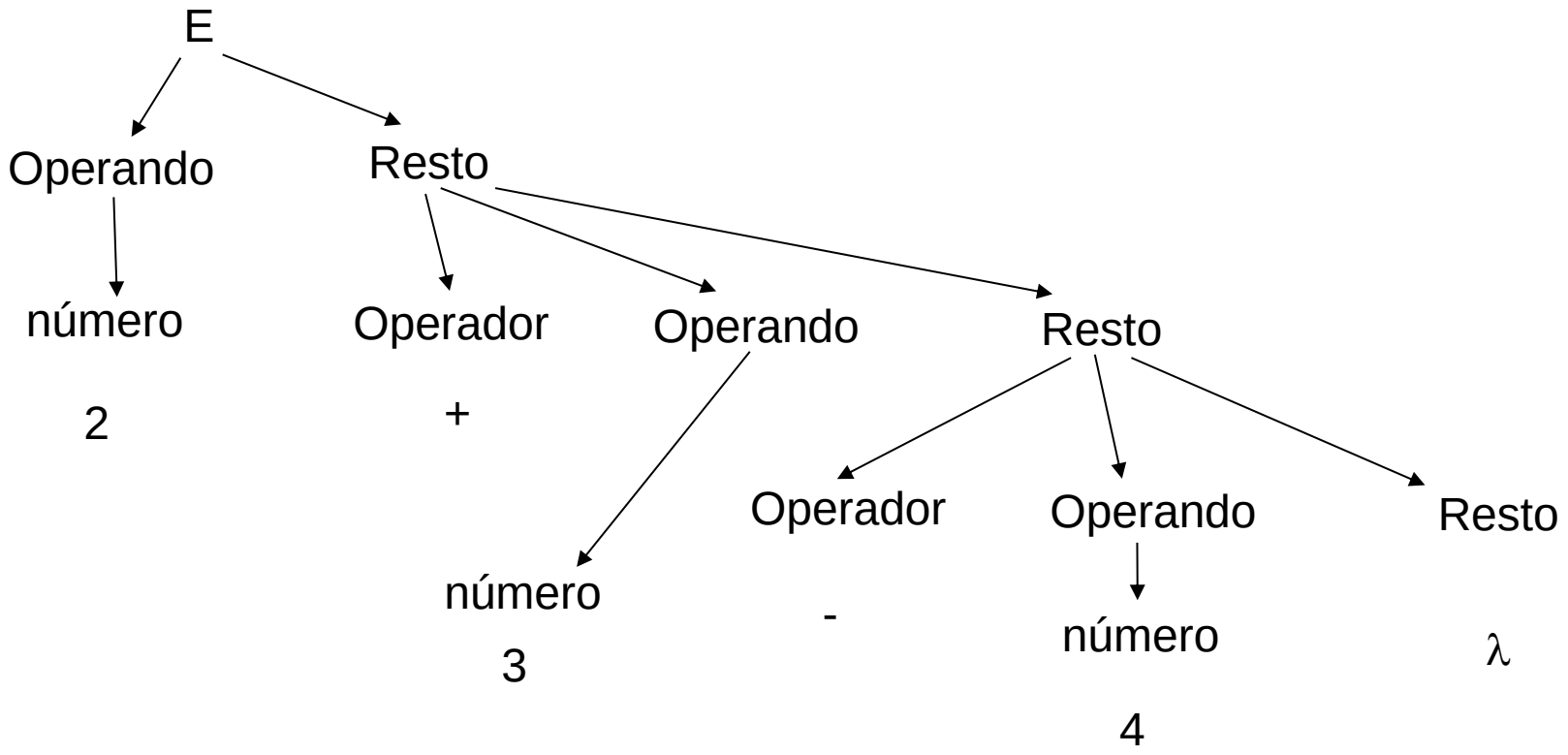
Un esquema de traducción es una gramática con atributos y las acciones semánticas pueden estar intercaladas entre los símbolos del lado derecho de una producción.

Nota: en un esquema de traducción las acciones semánticas tienen un orden de ejecución dado por su posición entre los símbolos de la parte derecha de una producción.

Ejemplo:
Sea el siguiente conjunto de producciones:

E → Operando Resto
Operando → número
Resto → Operador Operando Resto / λ
Operador → +/ - / *

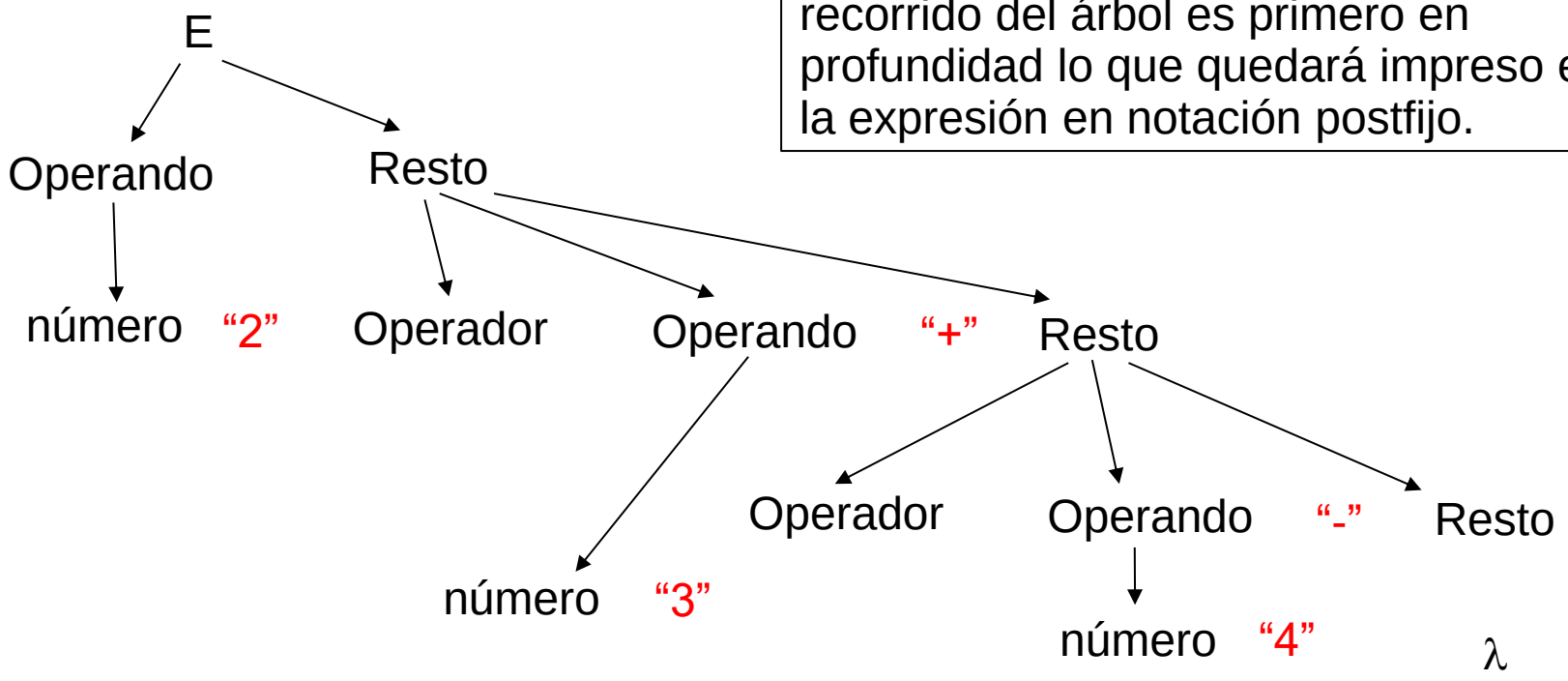
Sea w = 2 + 3 – 4



Esquema de traducción:

E → Operando Resto
Operando → número {print(numero.valor)}
Resto → Operador Operando {print(Operador.signo)} Resto / λ
Operador → +/ - / *

Con este esquema de traducción si el recorrido del árbol es primero en profundidad lo que quedará impreso es la expresión en notación postfijo.



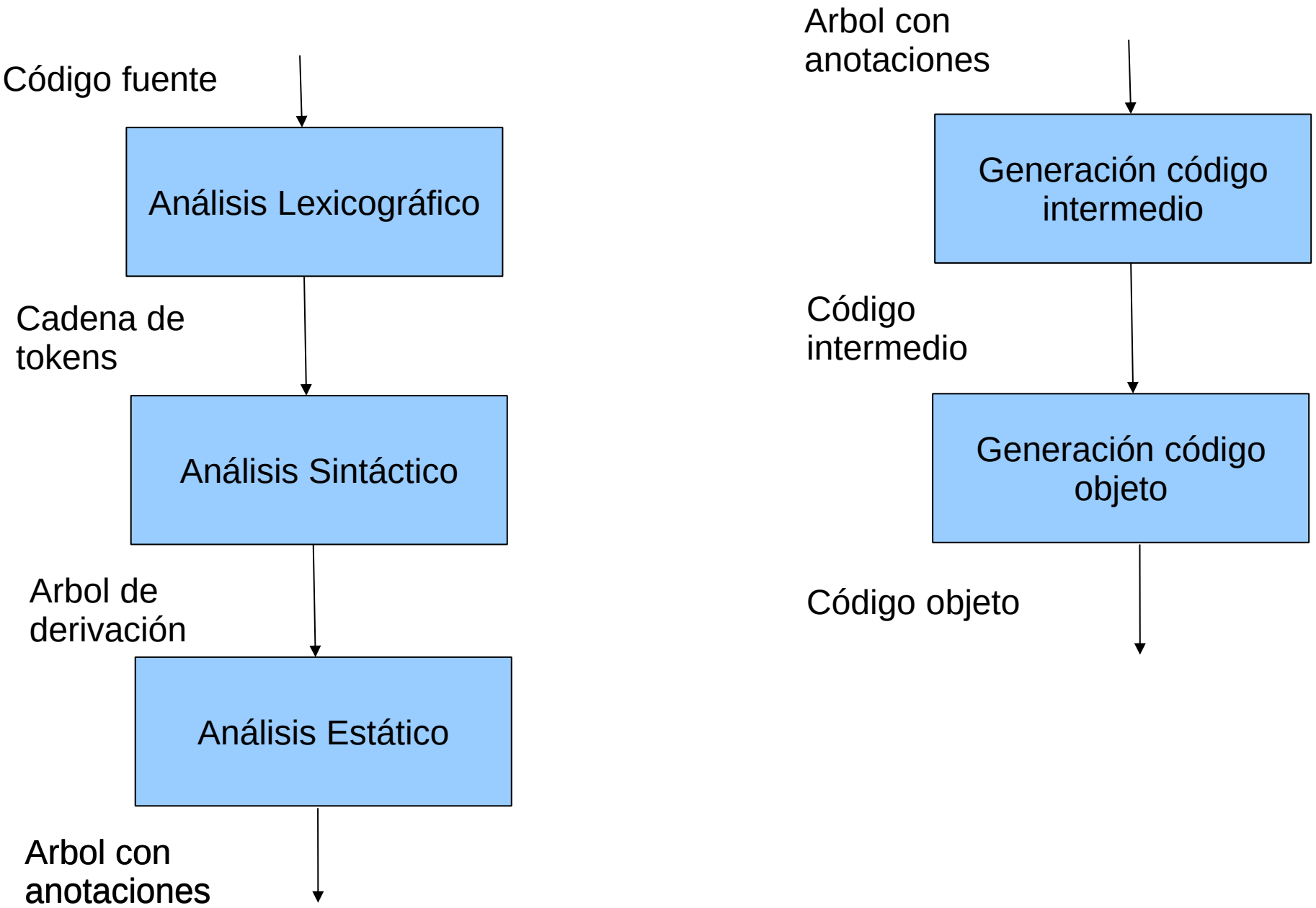
Valuación en una sola pasada.

Si una DDS es una gramática L-atribuida los atributos pueden ser completamente valuados siguiendo el orden dado por el recorrido del árbol de derivación primero en profundidad.

Esto es, podría evitarse la construcción del árbol de derivación.

Algoritmo Visitar (**b**)
Para cada hijo **a** de **b**
    Valuar los atributos heredados de **a** (de izquierda a derecha)
    Visitar(**a**)
Valuar los atributos sintetizados de **b**.

Esquema de un compilador



Las ventajas de separar el generador de código intermedio con el generador de código objeto son

1. Si debo implementar varios compiladores de esta manera podría implementar solamente un generador de código objeto a partir de código intermedio

2. Si cambio de plataforma (i.e. procesador) puedo tener un generador de código objeto para cada plataforma y así hago portable el compilador.

Análisis estático

A diferencia de la comprobación dinámica (que se realiza en tiempo de ejecución) en el análisis estático se llevan a cabo un variado conjunto de comprobaciones dependiendo de las características del lenguaje:

1. Comprobación de tipos.
    Ejemplo: int a[8]; int b,c; c= a + b;

2. Comprobación de flujo en una estructura.
    Ejemplo: break debe estar dentro de alguna estructura de control como loops, case, etc

3. Comprobación de unicidad.
    Ejemplo: en un case cada una de las opciones debe ser única.

4. Comprobación de apareamiento de etiquetas.
    Ejemplo: begin Chequear … end Chequear.

Nosotros vamos a ver el chequeo de tipos como un caso de aplicación de DDS para análisis estático.

Chequeo de tipos.

Definición (**expresión de tipo**):

Una expresión de tipo es, o bien un tipo básico o bien resulta de la aplicación de un constructor de tipo sobre una expresión de tipo.

Definición (**tipo básico**):

Un tipo básico es una expresión de tipo primitiva de un sistema de tipos (ver definición más adelante). Ejemplo de tipos básicos son: **boolean**, **character**, **integer**, **real**, **subrango**, etc.
Un tipo básico especial es **error_tipo**.

Constructores de tipo.
Hay varios tipos de constructores de tipo y pueden cambiar de acuerdo al sistema de tipos (ver definición más adelante):

1. **array:**
array(I, T)

donde I es un conjunto de índices y T es una expresión de tipo.
Ejemplo:

int a[10] se corresponde con array(1..10, integer)

2) **record**:

record($nombre_1$, $tipo_1$) x ($nombre_2$, $tipo_2$) x ..($nombre_k$, $tipo_k$)

Ejemplo:

struct tipo_a { *int* ctd; *char* desc[50];} a

se corresponde con

record(ctd, integer) x (desc, array(1..50, char))

3) **pointer**:

pointer(tipo)

Ejemplo:

char *ap;

se corresponde con

pointer(char)

4) **Funciones**: dominio $\rightarrow$ rango

D $\rightarrow$ R

Ejemplo:

int sumar(*int* a, *int* b)

se corresponde con

int x int $\rightarrow$ int

5) **etc.**

Definición (**sistema de tipos**):

Un sistema de tipos es un conjunto de reglas para asignar expresiones de tipo a las distintas partes de un programa.

Una forma de implementar un sistema de tipos es mediante una DDS.

Definición (**sistema de tipos seguro**):

Un sistema de tipos es seguro si al asignar a una porción del programa un tipo diferente a error_tipo se garantiza que no aparecerán errores de tipo durante la ejecución del mismo.

Definición (**lenguaje fuertemente tipado**):

Un lenguaje se dice fuertemente tipado si el compilador puede garantizar que los programas que acepte se ejecutarán sin errores de tipo.

Ejemplo:
Consideremos el lenguaje con el conjunto de producciones de la gramática:

$P \rightarrow D ; E$
$D \rightarrow D ; D$
$D \rightarrow id : T$
$T \rightarrow char$
$T \rightarrow int$
$T \rightarrow array [num] of T$
$T \rightarrow ^T$
$E \rightarrow literal$
$E \rightarrow num$
$E \rightarrow id$
$E \rightarrow E mod E$
$E \rightarrow E [E]$

El sistema de tipos tiene 2 tipos básicos: char e integer además de error_tipo

$P \rightarrow D ; E$

$D \rightarrow D ; D$

$D \rightarrow id : T$ {AggTipo(id.nombre, T.tipo)}

$T \rightarrow char$ {T.tipo := char}

$T \rightarrow int$ {T.tipo := integer}

$T \rightarrow array [num] of T$ {T_1.tipo := array(1..num, T_2.tipo)}

$T \rightarrow ^T$ {T_1.tipo := pointer(T_2.tipo)}

$E \rightarrow literal$ {E.tipo := char}

$E \rightarrow num$ {E.tipo := integer}

$E \rightarrow id$ {E.tipo := Tipo(id.nombre)}

$E \rightarrow E \text{ mod } E$ {E_1.tipo := if E_2.tipo == integer $\wedge$ E_3.tipo == integer
then integer
else error_tipo}

$E \rightarrow E [E]$ {E_1.tipo := if E_3.tipo == integer $\wedge$ E_2.tipo== array(s,t)
then t
else error_tipo}

Ejercicio: Considerar el agregado de las siguientes producciones

$T \rightarrow boolean$

Y en la gramática de las expresiones

$E \rightarrow E \text{ and } E$

$E \rightarrow E \text{ or } E$

$E \rightarrow \text{not } E$

Comprobación de proposiciones

Si, en el ejemplo anterior, cambiamos

$P \rightarrow D ; E$

por

$P \rightarrow D : S$

podríamos agregar instrucciones de asignación y control en nuestro lenguaje. Las expresiones serán parte de nuestros comandos

$S \rightarrow id = E$
$S \rightarrow if\ E\ then\ S$
$S \rightarrow do\ S\ while\ E$
$S \rightarrow S ; S$

Agregaremos el tipo básico tipo_vacío para expresar el tipo de las instrucciones. Es decir, si en una instrucción no hay un error de tipo entonces tendrá el tipo básico tipo_vacío, sino tendrá error_tipo.

$S \rightarrow id = E$ {S.tipo := if Tipo(id.nombre) == E.tipo
then vacío
else error_tipo}

$S \rightarrow$ if E then S {S_1.tipo := if E.tipo == boolean
then S_2.tipo
else error_tipo}

$S \rightarrow$ do S while E {S_1.tipo := if E.tipo == boolean
then S_2.tipo
else error_tipo}

$S \rightarrow S ; S$ {S_1.tipo := if S_2.tipo == vacio $\wedge$ S_3.tipo == vacio
then vacio
else error_tipo}

Si consideremos el agregado de

$E \rightarrow$ id_funcion(E)

la semántica a incluir debería ser

$E \rightarrow$id_funcion(E) {E_1.tipo := if E_2.tipo == s $\wedge$
Tipo(id_funcion.nombre) == s $\rightarrow$ t
then t
else error_tipo}

Conversiones de tipo

Si tenemos la siguiente porción de código

```
int a,c,b:
…
c = a + b;
```

y en otro programa esta otra

```
real a,b,c;
…
c = a + b;
```

cuando se genere código (intermedio u objeto), en un caso tendremos que traducirlo como suma de enteros y en otro como suma de flotantes.
Algo similar ocurre si uso el operador “+” para concatenar cadenas.

Esto se llama sobrecarga de operadores.

En estos casos, el significado del operador “+” se resuelve teniendo en cuenta el contexto.
Pero que pasa si el contexto es ambiguo:

```
int a;
real b,c;
…
c = a + b;
```

Definición (coersión):

Es el cambio de un tipo de una expresión para la resolución de un operador o una asignación con operandos de distinto tipo.
Puede ser implícita en cuyo caso la realiza el compilador, o explícita en cuyo caso está especificada en el código.

Ejemplo (de coersión implícita):

$E \rightarrow$ num {E.tipo := integer}

$E \rightarrow$ num.num {E.tipo := real}

$E \rightarrow$ id {E.tipo := Tipo(id.nombre)}

$E \rightarrow E$ op E {E1.tipo := if E_2.tipo == integer $\wedge$ E_3.tipo == integer
then integer
else if E_2.tipo == real $\wedge$ E_3.tipo == real
then real
else if E_2.tipo == integer $\wedge$ E_3.tipo == real
then real
else if E_2.tipo == real $\wedge$ E_3.tipo == integer
then real
else error_tipo}

Generación de código intermedio

Hay varios tipos de código intermedio, en realidad puede haber tantos como diseñadores de compiladores.
Nosotros veremos uno que se llama Código de tres direcciones.

Código de tres direcciones

Es una secuencia de proposiciones de la forma general

$$X := Y \text{ op } Z$$

donde X, Y, Z son nombres de etiquetas, constantes o variables (incluso temporales generadas por el compilador) y op representa algún tipo de operador.

Ejemplo: una expresión a + b * c exige dos proposiciones de 3 direcciones:

$$t_1 := b * c$$
$$t_2 := a + t_1$$

donde t1 son variables auxiliares suministradas durante la generación de código.

Tipos de proposiciones de tres direcciones

1. Proposiciones de asignación

X := Y op Z

donde op es un operador binario (i.e. lógico, aritmético, etc)

2. Proposiciones de asignación con operador unario

X := Y op Z

donde Z no es usado y op es un operador unario (i.e. Negación, desplazamiento de bits, operador de coersión, etc)

3. Proposiciones de copia

X := Y

4. Proposiciones de salto incondicional

X := Y op Z (o goto X)

donde Y y Z no son usados y op es el operador “goto”.

5. Proposiciones de salto condicional

X := Y op Z (o if Y op Z goto X)

donde Y y Z son expresiones, op es un operador lógico y X es una etiqueta (posición de otra proposición en la secuencia total de proposiciones).

6. Proposiciones para llamadas a funciones

X : = Y op Z (o Param X)

donde Y y Z no son usadas y X es una variable (dirección de la variable)

y X : = Y op Z (o Call X, Y)

donde X es una etiqueta e Y es una constante.

Ejemplo: para mi_función(x1, x2, x3, .., xn)

Param X1
Param X2
Param X3
..
Param Xn
Call mi_función, n

7. Proposiciones para asignaciones con índices de las formas

x = y[i] y x[i] = y

Donde las direcciones X, Y y Z son posiciones de memoria, o expresiones.

8. Proposiciones para asignaciones de la forma

x = &y, x = *y y *x = y

Donde se utilizan dos direcciones (X e Y) y el operador es uno de indirección, o valor apuntado o dirección apuntada.

Ejemplo:
Sean las siguientes producciones

$S \rightarrow id = E$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow – E$
$E \rightarrow (E)$
$E \rightarrow id$

Definimos un atributo que se llama “lugar” que es el nombre de la variable, y otro que se llama “código” que es la secuencia de proposiciones.

“gen” será una función la cual puede recibir distinta cantidad y tipos de argumentos, y la cual generará la cadena ascii de la proposición.
Por ejemplo, gen(id.lugar, ':=', E.lugar) generará una proposición

id := t

donde t es la posición dentro de la secuencia de proposiciones (posición de memoria) donde está el resultado de la evaluación de E e id es la posición dentro de la secuencia de proposiciones donde está el valor de la variable id.
La funciones EtiquetaNueva y VariableNueva permiten obtener nombres de etiquetas (fundamentalmente para saltos incondiconales) y reserva de lugares de memoria para variables.

$S \rightarrow id = E$ {S.código := E.código + gen(id.lugar, ':=', E.lugar)}

$E \rightarrow E + E$ {E_1.lugar := VariableNueva(); E_1.código := E_2.código + E_3.código + gen(E_1.lugar, ':=', E_2.lugar, '+', E_3.lugar)}

$E \rightarrow E * E$ {E_1.lugar := VariableNueva(); E_1.código := E_2.código + E_3.código + gen(E_1.lugar, ':=', E_2.lugar, '*', E_3.lugar)}

$E \rightarrow – E$ {E_1.lugar := VariableNueva(); E_1.código := E_2.código + gen(E_1.lugar, ':=', E_2.lugar, '-unario')}

$E \rightarrow (E)$ {E_1.lugar := E_1.lugar; E_1.código := E_2.código}

$E \rightarrow id$ {E.lugar := id.lugar; E.código := ' '}

Ejemplo:

Consideremos ahora la producción

$S \rightarrow$ while E do S

$S \rightarrow$ while E do S {S_1.comienzo := EtiquetaNueva();
S_1.después := EtiquetaNueva();
S_1.código := gen(S_1.comienzo, ':') +
E.codigo +
gen('if', E.lugar, '==', '0','goto',S_1.después) +
S_2.código +
gen('goto', S.comienzo) +
gen(S_1.después, ':') }