Repaso de Gramáticas, Lenguajes y sus transformaciones para PL

Gramática: $(\Sigma_T, \Sigma_N, S, P)$

- P — Conjunto de producciones sobre Σ_T y Σ_N
- S — Axioma (no terminal)
- Σ_N — Conjunto de símbolos no terminales
- Σ_T — Conjunto de símbolos terminales (alfabeto)

Tipos de Gramática que vamos a usar:

Tipo 2 (libres de contexto) y Tipo 3

Las producciones tienen la forma:

$W ::= x \qquad W \in \Sigma_N \qquad x \in (\Sigma_T \cup \Sigma_N)^*$

Se interpreta como:

Parte izquierda (W) ::= Parte derecha (x),

(parte izquierda genera parte derecha)

Donde la parte izquierda es un único No Terminal,
y la parte derecha es una secuencia de símbolos Terminales y No Terminales, pudiendo generar λ (palabra vacía).

Para las G3 añadimos unas restricciones adicionales:

Tienen que cumplir

O bien $x \in \Sigma_T \cup \{ \lambda \}$

O bien $x = yZ$ donde $y \in \Sigma_T$, $Z \in \Sigma_N$

Esto sería una Gramática de tipo 3 Lineal Derecha

También puede darse una G3 Lineal Izquierda

$W ::= x \qquad W \in \Sigma_N$

O bien $x \in \Sigma_T \cup \{ \lambda \}$

O bien $x = Zy$ donde $y \in \Sigma_T$, $Z \in \Sigma_N$

Pero de este tipo no lo usaremos habitualmente, o las intentaremos evitar. Motivos:

- Dan problemas a la hora de aplicar ciertos algoritmos
- Generan las palabras de derecha a izquierda, lo cuál es poco natural para cualquier proceso de lectura de un fichero (y para muchos humanos)

Correspondencias entre Lenguaje, Gramáticas y Autómatas.

Las Gramáticas generan un Lenguaje

Tipo 3, lenguajes que se pueden representar mediante Expresiones Regulares
Tipo 2, lenguajes libres de contexto (muchos lenguajes de programación)

Las Gramáticas tienen un Autómata equivalente capaz de procesar (reconocer) el Lenguaje equivalente.

Para las G3 es un AFD

Las G3LD se traducen inmediatamente a un AFD,
Las G3LI requieren de una transformación previa

Para las G2 es un Autómata a Pila (AP)

En general, en los lenguajes de programación subyacen gramáticas de tipo 2. El dispositivo capaz de procesar palabras de un lenguaje de tipo 2 es un Autómata a pila. La conversión de una Gramática de tipo 2 a AP es casi inmediata, pero hay un problema de calado. La mayoría de las Gramáticas generarán **APs no deterministas**, que en una situación concreta pueden aplicar dos transiciones distintas.

Recurriendo a un ejemplo sencillo:

S ::= aA | aB
A::= aA | a
B::= aB | b

Si queremos determinar si la palabra aaaaaab pertenece al lenguaje generado por la gramática, vemos que ya la primera aplicación de una producción plantea una indeterminación. Podemos aplicar tanto A::=aA como B::=aB. No queda más remedio que probar con una de las dos. Si optamos por la primera, descubriremos después de aplicar repetidas veces A::=aA que el último símbolo b no se puede generar, lo cuál invalida el camino. Teníamos que haber escogido el otro, pero en su momento no había forma de determinar la opción adecuada.

Este tipo de problemas sólo se pueden resolver si recurrimos a alguna técnica de búsqueda. En general, se realizará una búsqueda en un árbol en el que cada rama representa la derivación obtenida con una producción.
Para realizar la búsqueda en el árbol, una solución común es recurrir a técnicas de backtracking, que es la forma habitual en la que se basa la implementación de dispositivos no deterministas.

Su implementación puede ser más o menos sencilla, pero tiene un grave inconveniente, y es que, si la gramática induce muchas ramas de indeterminación, el proceso de búsqueda se ralentiza en exceso. Si hablamos de construir compiladores que deben

procesar ficheros con miles de líneas de código basándose en gramáticas de gran envergadura, el problema puede ser bastante grave [1].

En este contexto, las técnicas de búsqueda no ofrecen una eficiencia aceptable, ni siquiera al mejorarlas con heurísticas. Gran parte de la asignatura de Procesadores del Lenguaje se centra en desarrollar gramáticas que permitan generar *parsers* con un tiempo de respuesta casi lineal.

A la hora de diseñar las gramáticas tendremos que asumir ciertas restricciones:

1. Hay que evitar el NO DETERMINISMO

¿Cómo?
No es trivial, la solución recomendada es realizar un diseño cuidadoso de la gramática, evitando redundancias. En el ejemplo previo vemos redundancia en las producciones de A y B. Podemos resumirlas (con cuidado) eliminando una de ellas:

S ::= aA
A ::= aA | a | b

Una técnica habitual para estos casos es la **factorización**. Si observamos la primera producción del siguiente ejemplo: S ::= aA | aB

Vemos que ambas alternativas tienen una parte inicial en común (el símbolo "a"). La factorización es un proceso similar a "sacar factor común".
Añadimos un nuevo No Terminal que genera las opciones divergentes:

S ::= aZ
Z ::= A|B

¿Hemos resuelto el problema del No Determinismo?
Depende de lo que produzcan A y B. Supongamos los siguientes casos:

A) A ::= ba B ::= bb
Tendríamos Z →…→ ba | bb de nuevo No Determinista
Si factorizamos, nos queda:
Z ::= bY
Y ::= a | b ➔ problema resuelto

B) A ::= ba B ::= **b**
Tendríamos Z →…→ ba | b de nuevo No Determinista
Si factorizamos, nos queda:
Z ::= bY
Y ::= a | λ ➔ hemos cambiado una disyuntiva no determinista por una producción no generativa (anulable).

Si la gramática es de tipo 3 (y esta es equivalente a G3LD), obtendríamos un AFND con transiciones lambda, ¡PERO disponemos de un algoritmo para transformarlo en AFD!
Si la gramática es de tipo 2, la formulación Y ::= a | λ tiene mejor solución en muchos tipos de *parser* que las disyuntivas no deterministas.

[1] Hay lenguajes actuales cuyas gramáticas superan con facilidad las 500 producciones. Y de un buen compilador se espera que sea capaz de procesar incluso decenas de miles de líneas de código en un breve tiempo.

2. Otra cuestión que debemos evitar: La ambigüedad

Esta aparece cuando podemos obtener una misma derivación por caminos (de aplicación de producciones) diferentes
Un caso conocido lo plantea el encadenamiento de sentencias if-then[-else]

If (A) then if (B) then X else Y

SENT ::= if COND then SENT | if COND then SENT else SENT
SENT ::= X | Y

S | If C then S | If C then S else S	S | If C then S else S | If C then S
If COND then If COND then S Else S	If COND then If COND then S Else S

Ambas derivaciones son sintácticamente idénticas, pero presentan diferencias considerables a nivel semántico.

Para evitar la ambigüedad tenemos dos opciones:

A) Usar separadores específicos en el diseño gramatical (llaves, endif, …)
   If (A) then { if (B) then { X } else { Y } }
   If (A) then { if (B) then { X } } else { Y }
B) Usar precedencias y/o asociatividad (lo veremos más adelante)

Ejemplo:

Expresión ::= Expresión + Expresión | Expresión * Expresión | Numero
Numero ::= 0 | 1 | 2 | … | 9

3*4+5 se puede derivar según se apliquen las producciones como

3*(4+5) o (3*4)+5

De nuevo, ambos casos son sintácticamente indistinguibles, pero semánticamente diferentes, puesto que su evaluación genera 27 y 17 respectivamente.

La **ambigüedad** y el **no determinismo** son conceptos diferentes, pero con frecuencia aparecen vinculados porque implican diversas posibilidades de aplicar la gramática. Además, muchas veces son difíciles de detectar. La ambigüedad puede no ser un problema a nivel sintáctico, sólo cuando se intenta atribuir una semántica a una sentencia.

En el ejemplo del IF observamos también un No Determinismo.
A la hora de derivar SENTENCIA ¿cuál de las dos producciones posibles aplicamos?
Este No Determinismo lo podemos evitar rediseñando la gramática mediante una factorización.

```
SENT ::= if COND then SENT | if COND then SENT else SENT
```

Se transforma en:

```
SENT ::= if COND then SENT RESTO_IF

RESTO_IF ::= lambda | else SENT
```

Deberemos tener en cuenta una serie de aspectos a la hora de diseñar gramáticas:

Evitar la aparición de:
- No Determinismo
- Ambigüedad

Para ello buscaremos:
+ Que el primer símbolo de la parte derecha de las producciones sea Terminal.
- Evitar los no terminales como primer símbolo de la parte derecha.
- Evitar la Recursividad a Izquierdas (A ::= Aw).
- Evitar doble Recursividad en las producciones (A ::= xAwA).

Las técnicas que podemos aplicar son:
- Limpiar la gramática, pero no bien formar.
- Factorizar la gramática.
- Sustitución de No Terminales.
- Eliminación de Recursividad a Izquierdas.
- Paso a Forma Normal de Greibach.
- Paso a Forma Normal de Chomsky (aplicación del parser CYK)
- Buscar el diseño estructurado y sistemático de las gramáticas, por bloques/subgramáticas: $a^n b^m a^m b^n$
- Reescritura de la gramática.

La aplicación de todas estas cuestiones no está sujeta a un criterio claro. Depende mucho del tipo de gramática y analizador sintáctico que se vaya a emplear. Y también puede influir en gran medida cómo se integre el nivel sintáctico con el nivel semántico.

Al final, es la experiencia la que dicta qué transformaciones se deben aplicar o no. O seguir un proceso de "prueba y error". Por todo ello es conveniente tener una cierta soltura a la hora de manejar estos conceptos y técnicas.