UNIVERSITAT JAUME·I

4º Ingeniería Informática

II26 Procesadores de lenguaje

Analizador sintáctico

Esquema del tema

1. Introducción

El analizador sintáctico tiene como objetivo encontrar las estructuras presentes en su entrada. Estas estructuras se pueden representar mediante el árbol de análisis sintáctico, que explica cómo se puede derivar la cadena de entrada en la gramática que especifica el lenguaje. Aunque en la práctica es habitual que el árbol de análisis no llegue a construirse, se trata de una abstracción que nos permite entender mejor todo el proceso.

Para construir la especificación sintáctica de los lenguajes de programación, se suelen emplear gramáticas incontextuales, generalmente restringidas para que el análisis se pueda realizar de manera eficiente.

Para que sea posible construir el árbol de análisis, es necesario que la entrada no presente errores sintácticos. En caso de que los haya, el analizador debe informar de su presencia adecuadamente y, si es posible, intentar continuar el análisis.

2. Gramáticas incontextuales

Ya hemos comentado que las gramáticas incontextuales se emplean para especificar la sintaxis de los lenguajes de programación. Sabemos que son un paso intermedio entre las gramáticas regulares y las sensibles al contexto. Es natural preguntarse por qué nos detenemos en ellas. Hay diversas razones:

- Tienen suficiente capacidad expresiva para representar la mayor parte de las construcciones presentes en el compilador. Además, es un formalismo que puede aumentarse con sencillez para incorporar restricciones que no son puramente incontextuales.
- Son razonablemente sencillas de diseñar. Pese a que el poder expresivo de las gramáticas sensibles al contexto es mucho mayor, su diseño es excesivamente complejo para ser empleado en la práctica.
- Permiten un análisis eficiente en el caso general y muy eficiente si se introducen una serie de restricciones que no suponen una limitación excesiva de su poder expresivo.

Ahora recordaremos los conceptos más importantes de las gramáticas incontextuales.

2.1. Definición

Como seguramente recordarás, una gramática incontextual es una cuádrupla $G = (N, \Sigma, P, \langle \mathsf{S} \rangle)$ donde

- N es un alfabeto de *no terminales*.
- Σ es un alfabeto de *terminales*, disjunto con N.
- $P \subseteq N \times (N \cup \Sigma)^*$ es un conjunto de *producciones* o *reglas*.
- $\langle \mathsf{S} \rangle \in N$ es el *símbolo inicial*.

Como ves, las producciones de una gramática incontextual tienen únicamente un no terminal en la parte izquierda.

2.2. Derivaciones

El funcionamiento de las gramáticas se basa en el concepto de *derivación*. Comenzando por una cadena formada únicamente por el símbolo inicial, se aplican repetidamente las reglas de P hasta llegar a una cadena formada únicamente por terminales. Aplicar una regla consiste simplemente en encontrar, dentro de la cadena actual, la parte izquierda de la regla y sustituirla por la parte derecha correspondiente.

Por ejemplo, sea la gramática: $G = (\{\langle \mathsf{S} \rangle, \langle \mathsf{E} \rangle\}, \{\mathsf{id}, +, *\}, P, \langle \mathsf{S} \rangle)$, con

$$\begin{aligned} P \;=\; \{&\langle \mathsf{S} \rangle \to \langle \mathsf{E} \rangle, \\ &\langle \mathsf{E} \rangle \to \langle \mathsf{E} \rangle + \langle \mathsf{E} \rangle, \\ &\langle \mathsf{E} \rangle \to \langle \mathsf{E} \rangle * \langle \mathsf{E} \rangle, \\ &\langle \mathsf{E} \rangle \to \mathsf{id}\}; \end{aligned}$$

para derivar **id+id*id** podemos hacer:

$$\begin{aligned} \langle \mathsf{S} \rangle &\Rightarrow \langle \mathsf{E} \rangle \\ &\Rightarrow \langle \mathsf{E} \rangle + \langle \mathsf{E} \rangle \\ &\Rightarrow \langle \mathsf{E} \rangle + \langle \mathsf{E} \rangle * \langle \mathsf{E} \rangle \\ &\Rightarrow \mathsf{id} + \langle \mathsf{E} \rangle * \langle \mathsf{E} \rangle \\ &\Rightarrow \mathsf{id} + \mathsf{id} * \langle \mathsf{E} \rangle \\ &\Rightarrow \mathsf{id} + \mathsf{id} * \mathsf{id} \end{aligned}$$

Ejercicio 1

Encuentra otra derivación de **id+id*id**.

Ejercicio* 2

¿Cuántas derivaciones distintas tiene la cadena **id+id*id**?

De una manera más formal, tenemos las siguientes definiciones:

Derivación directa: η deriva directamente γ (en G) si $\eta = \omega_1 \langle \mathsf{A} \rangle \omega_2$, $\gamma = \omega_1 \beta \omega_2$ y $\langle \mathsf{A} \rangle \to \beta \in P$. Lo denotamos con $\eta \Rightarrow \gamma$.

Derivación: η deriva γ (en G) si hay una secuencia de cadenas $\{\xi_1, \xi_2, \ldots, \xi_n\}$ de $(N \cup \Sigma)^*$ tal que $\eta = \xi_1$, $\gamma = \xi_n$ y $\xi_i \Rightarrow \xi_{i+1}$ para $1 \leq i < n$. Lo denotamos con $\eta \overset{*}{\Rightarrow} \gamma$.

Forma sentencial: Cadena α de $(N \cup \Sigma)^*$ tal que $\langle \mathsf{S} \rangle \overset{*}{\Rightarrow} \alpha$.

Sentencia: forma sentencial perteneciente a Σ^*.

Lenguaje definido (o generado) por la gramática G: $L(G) = \{x \in \Sigma^* \mid \langle \mathsf{S} \rangle \overset{*}{\Rightarrow} x\}$.

2.3. Convenios de notación

En lo sucesivo y con objeto de no tener que indicar explícitamente a qué alfabeto (N o Σ) pertenece cada símbolo o cadena, asumiremos unos convenios que permitirán deducirlo inmediatamente:

- Representamos los **terminales** con un tipo de letra mecanográfico (`a`, `c`, `+`, `:=`), en negrita (**identificador**, **entero**) o entrecomillado ("`(`", "`)`").
- Los **no terminales** se representan encerrados entre ángulos ($\langle$A$\rangle$, $\langle$Programa$\rangle$).
- Un **símbolo que puede ser terminal o no terminal**, indistintamente, se representará con alguna de las últimas letras mayúsculas del alfabeto latino (X, Y,...).
- La **cadenas de símbolos terminales** (elementos de Σ^*) se representan con la últimas letras minúsculas del alfabeto latino (u, v, w,...).
- Las **cadenas de símbolos terminales o no terminales** (elementos de $(N \cup \Sigma)^*$) se representan con letras minúsculas del alfabeto griego (α, β, γ,...).

Además, cuando escribamos una gramática, sólo mostraremos sus reglas y asumiremos que el símbolo inicial es el que está en la parte izquierda de la primera regla.

También asumiremos que la gramática no tiene ni *símbolos inútiles* ni *reglas inútiles*. Esto es, para cada símbolo o regla, habrá al menos una derivación de una sentencia del lenguaje que contenga ese símbolo o regla.

2.4. Algunas definiciones adicionales

Si una producción tiene el símbolo $\langle$A$\rangle$ en su parte izquierda, la llamamos $\langle$A$\rangle$-producción. Si su parte derecha es únicamente la cadena vacía, decimos que es una λ-producción.

Si α deriva β en uno o más pasos, escribiremos $\alpha \overset{+}{\Rightarrow} \beta$.

Las producciones del tipo $\langle$A$\rangle \rightarrow \langle$B$\rangle$ se llaman *producciones simples*. Si tenemos que para algún no terminal $\langle$A$\rangle \overset{+}{\Rightarrow} \langle$A$\rangle$, decimos que la gramática tiene *ciclos*. Si para alguna β tenemos que $\langle$A$\rangle \overset{+}{\Rightarrow} \langle$A$\rangle\beta$, diremos que la gramática tiene *recursividad por la izquierda*. Análogamente, la gramática tiene *recursividad por la derecha* si para alguna β tenemos que $\langle$A$\rangle \overset{+}{\Rightarrow} \beta\langle$A$\rangle$.

2.5. Árboles de análisis y ambigüedad

Sigamos con la gramática G:

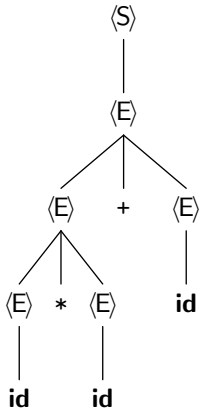
$$
\begin{array}{lcl}
\langle \mathsf{S} \rangle & \rightarrow & \langle \mathsf{E} \rangle \\
\langle \mathsf{E} \rangle & \rightarrow & \langle \mathsf{E} \rangle \mathtt{+} \langle \mathsf{E} \rangle \\
\langle \mathsf{E} \rangle & \rightarrow & \langle \mathsf{E} \rangle \mathtt{*} \langle \mathsf{E} \rangle \\
\langle \mathsf{E} \rangle & \rightarrow & \mathbf{id}
\end{array}
$$

La sentencia **id*id+id**, tiene distintas derivaciones:

$$
\begin{array}{l}
\langle \mathsf{S} \rangle \Rightarrow \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{*} \langle \mathsf{E} \rangle \mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{*}\mathbf{id}\mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \mathbf{id}\mathtt{*}\mathbf{id}\mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \mathbf{id}\mathtt{*}\mathbf{id}\mathtt{+}\mathbf{id}, \\
\langle \mathsf{S} \rangle \Rightarrow \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{+}\mathbf{id} \Rightarrow \langle \mathsf{E} \rangle \mathtt{*} \langle \mathsf{E} \rangle \mathtt{+}\mathbf{id} \Rightarrow \langle \mathsf{E} \rangle \mathtt{*}\mathbf{id}\mathtt{+}\mathbf{id} \Rightarrow \mathbf{id}\mathtt{*}\mathbf{id}\mathtt{+}\mathbf{id}, \\
\langle \mathsf{S} \rangle \Rightarrow \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{*} \langle \mathsf{E} \rangle \mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \mathbf{id}\mathtt{*} \langle \mathsf{E} \rangle \mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \mathbf{id}\mathtt{*}\mathbf{id}\mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \mathbf{id}\mathtt{*}\mathbf{id}\mathtt{+}\mathbf{id}, \\
\langle \mathsf{S} \rangle \Rightarrow \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{*} \langle \mathsf{E} \rangle \mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{*}\mathbf{id}\mathtt{+} \langle \mathsf{E} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathtt{*}\mathbf{id}\mathtt{+}\mathbf{id} \Rightarrow \mathbf{id}\mathtt{*}\mathbf{id}\mathtt{+}\mathbf{id}, \\
\ldots
\end{array}
$$

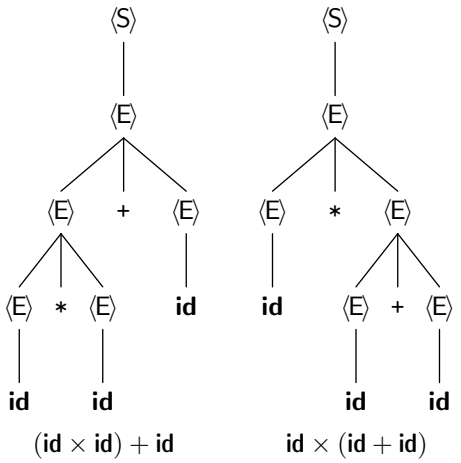
© Universitat Jaume I 2010-2011

Sin embargo, en cierto sentido, todas estas derivaciones representan una misma "estructura". De alguna manera nos transmiten la idea de que se está sumando el producto de los dos primeros **id** con el tercero. Podemos representar esto de forma compacta mediante un árbol como el siguiente:



Esto es lo que se conoce como *árbol de análisis* o *de derivación*. Intuitivamente, lo que hacemos es representar "en paralelo" las reglas necesarias para derivar la cadena de entrada. La raíz es el símbolo inicial de la gramática y cada nodo interior representa una producción: está etiquetado con un no terminal ⟨A⟩ y sus hijos de izquierda a derecha están etiquetados con $X_1, X_2, \ldots, X_n$ de modo que $\langle A \rangle \to X_1 X_2 \ldots X_n$ es una regla de la gramática.

Puede suceder que una misma cadena tenga asociado más de un árbol de derivación. Por ejemplo, la cadena anterior tiene asociados dos árboles, cada uno con un significado distinto:



Decimos que una sentencia es *ambigua* (con respecto a una gramática) si tiene más de un árbol de análisis. Si al menos una de las sentencias que se derivan de una gramática es ambigua, decimos que la gramática es ambigua. En algunos casos, es posible "resolver" la ambigüedad modificando la gramática adecuadamente, como veremos para el caso de las expresiones aritméticas. Sin embargo, existen lenguajes que tienen la propiedad de ser *inherentemente* ambiguos; no existe ninguna gramática no ambigua que los genere.

Está claro que la ambigüedad es perjudicial para un lenguaje de programación y debe ser evitada. Desgraciadamente, averiguar si una gramática es ambigua es un problema indecidible. Lo que sí que podremos garantizar es que las gramáticas pertenecientes a las familias que estudiaremos son no ambiguas.

2.6. Derivaciones canónicas

Hemos visto que para una sentencia de la gramática existen en general bastantes derivaciones posibles. De entre todas ellas, se suelen destacar dos, a las que se les llama derivaciones canónicas.

Estas se caracterizan por aplicar las reglas de manera sistemática sobre el no terminal más a la izquierda (*derivación canónica por la izquierda*) o más a la derecha (*derivación canónica por la derecha*).

Así una derivación canónica por la izquierda de **id∗id+id** es:

$$\langle \underset{\uparrow}{\mathsf{S}} \rangle \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{+} \langle \mathsf{E} \rangle \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{*} \langle \mathsf{E} \rangle \mathsf{+} \langle \mathsf{E} \rangle \Rightarrow \mathsf{id*} \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{+} \langle \mathsf{E} \rangle \Rightarrow \mathsf{id*id+} \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \mathsf{id*id+id}.$$

Cada árbol de derivación tiene asociada una derivación canónica por la izquierda y otra por la derecha. Por eso, si la sentencia es ambigua, habrá, al menos, dos derivaciones canónicas por la izquierda y otras tantas por la derecha. Dado que la gramática del ejemplo es ambigua, tenemos otra posible derivación por la izquierda para la expresión:

$$\langle \underset{\uparrow}{\mathsf{S}} \rangle \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{*} \langle \mathsf{E} \rangle \Rightarrow \mathsf{id*} \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \mathsf{id*} \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{+} \langle \mathsf{E} \rangle \Rightarrow \mathsf{id*id+} \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \mathsf{id*id+id}.$$

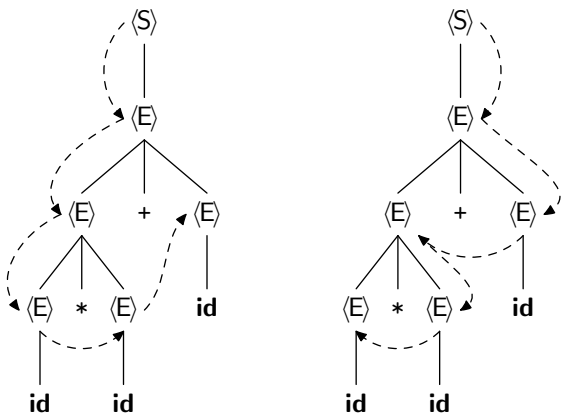
Las derivaciones canónicas por la derecha de **id∗id+id** son:

$$\langle \underset{\uparrow}{\mathsf{S}} \rangle \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathsf{+} \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{+id} \Rightarrow \langle \mathsf{E} \rangle \mathsf{*} \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{+id} \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{*id+id} \Rightarrow \mathsf{id*id+id},$$
$$\langle \underset{\uparrow}{\mathsf{S}} \rangle \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathsf{*} \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathsf{*} \langle \mathsf{E} \rangle \mathsf{+} \langle \underset{\uparrow}{\mathsf{E}} \rangle \Rightarrow \langle \mathsf{E} \rangle \mathsf{*} \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{+id} \Rightarrow \langle \underset{\uparrow}{\mathsf{E}} \rangle \mathsf{*id+id} \Rightarrow \mathsf{id*id+id}.$$

Es sencillo obtener, a partir del árbol de análisis, las correspondientes derivaciones canónicas por la derecha y por la izquierda:

- La derivación canónica por la izquierda se obtiene recorriendo el árbol de modo que se visita cada nodo, se escribe la regla correspondiente, y después, recursivamente, se visitan sus hijos de izquierda a derecha.
- La derivación canónica por la derecha se obtiene recorriendo el árbol de modo que se visita cada nodo, se escribe la regla correspondiente, y después, recursivamente, se visitan sus hijos de derecha a izquierda.

En nuestro ejemplo, los órdenes de visita para obtener las derivaciones canónicas por la izquierda y por la derecha, respectivamente, son:



© Universitat Jaume I 2010-2011

Ejercicio 3

Dada la gramática con las reglas:

$$\begin{aligned} \langle \mathsf{E} \rangle &\rightarrow \langle \mathsf{E} \rangle + \langle \mathsf{T} \rangle \mid \langle \mathsf{T} \rangle \\ \langle \mathsf{T} \rangle &\rightarrow \langle \mathsf{T} \rangle * \langle \mathsf{F} \rangle \mid \langle \mathsf{F} \rangle \\ \langle \mathsf{F} \rangle &\rightarrow \mathbf{id} \end{aligned}$$

Encuentra el árbol de análisis de **id*id+id**. A partir de este árbol, escribe la derivación canónica por la derecha y por la izquierda.

¿Existe más de un árbol de análisis para esa sentencia?

Ejercicio* 4

¿Puede una sentencia tener infinitos árboles de análisis con una gramática?

Pista: piensa en una gramática con ciclos.

3. Algunas construcciones de los lenguajes de programación

Ahora veremos cómo se expresan algunas de las construcciones más habituales de los lenguajes de programación mediante gramáticas. Lógicamente, la lista de construcciones presentadas no será completa. Debes tomarla más como una lista de ejemplos que como una referencia. Es más, las menciones que se hacen a C y Pascal no implican que ésta sea la manera de escribir las estructuras en estos lenguajes; simplemente sirven para que tengas una idea de a qué nos referimos.

3.1. Estructura del programa

Los lenguajes tipo Pascal tienen una estructura similar a:

$$\langle \mathsf{Programa} \rangle \rightarrow \langle \mathsf{Cabecera} \rangle \langle \mathsf{Bloque} \rangle .$$

con:

$$\begin{aligned} \langle \mathsf{Cabecera} \rangle &\rightarrow \mathbf{program\ id\,;} \\ \langle \mathsf{Bloque} \rangle &\rightarrow \langle \mathsf{Declaraciones} \rangle\ \mathbf{begin}\ \langle \mathsf{ListaSentencias} \rangle\ \mathbf{end} \\ \langle \mathsf{Declaraciones} \rangle &\rightarrow \langle \mathsf{DConstantes} \rangle \langle \mathsf{DTipos} \rangle \langle \mathsf{DVariables} \rangle \langle \mathsf{DSubrutinas} \rangle \end{aligned}$$

Los lenguajes tipo C tienen una estructura similar a:

$$\langle \mathsf{Programa} \rangle \rightarrow \langle \mathsf{ListaDeclaraciones} \rangle$$

con:

$$\begin{aligned} \langle \mathsf{ListaDeclaraciones} \rangle &\rightarrow \langle \mathsf{Declaración} \rangle \langle \mathsf{ListaDeclaraciones} \rangle \mid \lambda \\ \langle \mathsf{Declaración} \rangle &\rightarrow \langle \mathsf{DeclVariable} \rangle \mid \langle \mathsf{DeclTipo} \rangle \mid \langle \mathsf{DeclFunción} \rangle \end{aligned}$$

Ejercicio 5

Construye una gramática para un lenguaje tipo Pascal en que las declaraciones de constantes, tipos, variables y subprogramas puedan aparecer en cualquier orden y cualquier número de veces.

3.2. Secuencias de instrucciones

Habitualmente se distinguen dos tipos de secuencias de instrucciones:

- Sentencias *separadas* por punto y coma (Pascal):

$$\begin{aligned} \langle \text{ListaSentencias} \rangle &\rightarrow \langle \text{Sentencia} \rangle \ ; \ \langle \text{ListaSentencias} \rangle \\ \langle \text{ListaSentencias} \rangle &\rightarrow \langle \text{Sentencia} \rangle \end{aligned}$$

- Sentencias *terminadas* por punto y coma (C):

$$\begin{aligned} \langle \text{ListaSentencias} \rangle &\rightarrow \langle \text{Sentencia} \rangle \langle \text{ListaSentencias} \rangle \\ \langle \text{ListaSentencias} \rangle &\rightarrow \langle \text{Sentencia} \rangle \end{aligned}$$

En el segundo caso, las producciones para ⟨Sentencia⟩ deben incorporar el punto y coma, ya que, aunque informalmente se dice que C es un lenguaje en que las sentencias terminan en punto y coma, no todas lo hacen. Por ejemplo, la sentencia compuesta no termina en punto y coma, lo que hace que las sentencias de control de flujo tampoco terminen en punto y coma si afectan a una sentencia compuesta.

Ejercicio 6

¿Qué se hace en Pascal para que una lista de sentencias separadas por punto y coma pueda, en determinadas circunstancias, terminar en un punto y coma?

3.3. Sentencia compuesta

La sentencia compuesta es simplemente una lista de sentencias encerrada entre dos marcadores adecuados. Por ejemplo, en Pascal:

$$\langle \text{Sentencia} \rangle \rightarrow \textbf{begin} \ \langle \text{ListaSentencias} \rangle \ \textbf{end}$$

En C es posible incluir declaraciones dentro de las sentencias compuestas:

$$\langle \text{Sentencia} \rangle \rightarrow \{ \langle \text{Declaraciones} \rangle \langle \text{ListaSentencias} \rangle \}$$

En C++ se pueden intercalar las declaraciones con las sentencias:

$$\langle \text{Sentencia} \rangle \rightarrow \{ \langle \text{ListaSentenciasYDeclaraciones} \rangle \}$$

3.4. Declaraciones de variables

Podemos distinguir dos tipos de declaraciones de variables:

- Un tipo precediendo a una lista de identificadores (C):

$$\begin{aligned} \langle \text{DeclVariables} \rangle &\rightarrow \langle \text{Tipo} \rangle \langle \text{ListaIds} \rangle \, ; \\ \langle \text{ListaIds} \rangle &\rightarrow \langle \text{Id} \rangle \, , \langle \text{ListaIds} \rangle \mid \langle \text{Id} \rangle \end{aligned}$$

  donde ⟨Id⟩ es un identificador, posiblemente afectado por modificadores como * o corchetes.

- Un tipo siguiendo a una lista de identificadores (Pascal):

$$\begin{aligned} \langle \text{DeclVariables} \rangle &\rightarrow \langle \text{ListaIds} \rangle : \langle \text{Tipo} \rangle \\ \langle \text{ListaIds} \rangle &\rightarrow \textbf{id} \, , \langle \text{ListaIds} \rangle \mid \textbf{id} \end{aligned}$$

© Universitat Jaume I 2010-2011

3.5. Declaración de subrutinas

Las subrutinas suelen declararse como una cabecera seguida de un bloque (Pascal):

⟨DeclSubrutinas⟩ → ⟨Cabecera⟩⟨Bloque⟩

Si no hay anidamiento, en lugar de un bloque se puede emplear una lista de declaraciones y sentencias delimitadas (C):

⟨DeclSubrutinas⟩ → ⟨Cabecera⟩{⟨Declaraciones⟩⟨ListaSentencias⟩}

3.6. Estructuras de control condicional

La estructura de control condicional puede llevar un fin explícito como en:

⟨Sentencia⟩ → **if** ⟨Condición⟩ **then** ⟨ListaSentencias⟩ **endif**
⟨Sentencia⟩ → **if** ⟨Condición⟩ **then** ⟨ListaSentencias⟩ **else** ⟨ListaSentencias⟩ **endif**

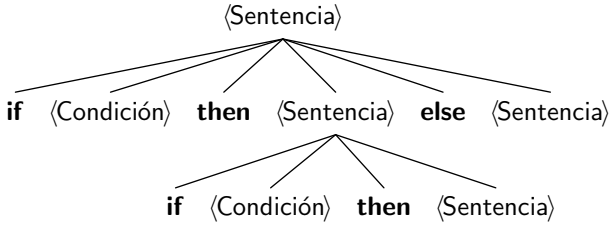
Si no lo lleva, podemos utilizar las reglas siguientes:

⟨Sentencia⟩ → **if** ⟨Condición⟩ **then** ⟨Sentencia⟩
⟨Sentencia⟩ → **if** ⟨Condición⟩ **then** ⟨Sentencia⟩ **else** ⟨Sentencia⟩

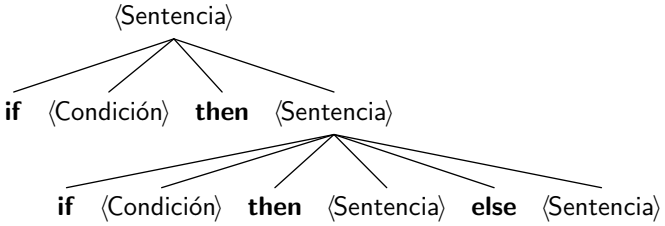
El problema de esta construcción es que es ambigua. Sea la siguiente forma sentencial:

if ⟨Condición⟩ **then if** ⟨Condición⟩ **then** ⟨Sentencia⟩ **else** ⟨Sentencia⟩

Un árbol de derivación es:



En este caso, el **else** pertenece al primer **if**. Pero también tenemos el árbol:



Este caso se corresponde a la regla habitual de asociar el **else** al **if** más cercano. Para incorporar esta regla a la gramática, tendremos dos no terminales que reflejarán si los **if-then** tienen que llevar obligatoriamente un **else** o no. El no terminal ⟨Sentencia⟩ podrá derivar condicionales con y sin parte **else** y el resto de sentencias. Por otro lado, las sentencias que se deriven del no terminal ⟨ConElse⟩ serán condicionales completos o sentencias de otro tipo. La gramática queda:

⟨Sentencia⟩ → **if** ⟨Condición⟩ **then** ⟨Sentencia⟩
| **if** ⟨Condición⟩ **then** ⟨ConElse⟩ **else** ⟨Sentencia⟩
| ⟨OtrasSentencias⟩
⟨ConElse⟩ → **if** ⟨Condición⟩ **then** ⟨ConElse⟩ **else** ⟨ConElse⟩
| ⟨OtrasSentencias⟩

En la práctica, esta modificación complica la construcción del analizador por la presencia de prefijos comunes (lo veremos más adelante en este mismo tema) y lo que se hace es manipular el analizador sintáctico para que cumpla la regla.

Ejercicio 7

Encuentra el árbol de derivación de **if id then if id then print else return** con la gramática:

$$
\begin{array}{rcl}
\langle \text{Sentencia} \rangle & \rightarrow & \textbf{if } \langle \text{Condición} \rangle \textbf{ then } \langle \text{Sentencia} \rangle \\
& | & \textbf{if } \langle \text{Condición} \rangle \textbf{ then } \langle \text{ConElse} \rangle \textbf{ else } \langle \text{Sentencia} \rangle \\
& | & \langle \text{OtrasSentencias} \rangle \\
\langle \text{ConElse} \rangle & \rightarrow & \textbf{if } \langle \text{Condición} \rangle \textbf{ then } \langle \text{ConElse} \rangle \textbf{ else } \langle \text{ConElse} \rangle \\
& | & \langle \text{OtrasSentencias} \rangle \\
\langle \text{OtrasSentencias} \rangle & \rightarrow & \textbf{print} \mid \textbf{return} \\
\langle \text{Condición} \rangle & \rightarrow & \textbf{id}
\end{array}
$$

Ejercicio* 8

Nos han propuesto la siguiente gramática para resolver el problema de la ambigüedad del **if-then**:

$$
\begin{array}{rcl}
\langle \text{Sentencia} \rangle & \rightarrow & \textbf{if } \langle \text{Condición} \rangle \textbf{ then } \langle \text{Sentencia} \rangle \\
\langle \text{Sentencia} \rangle & \rightarrow & \langle \text{ConElse} \rangle \\
\langle \text{ConElse} \rangle & \rightarrow & \textbf{if } \langle \text{Condición} \rangle \textbf{ then } \langle \text{ConElse} \rangle \textbf{ else } \langle \text{Sentencia} \rangle \\
\langle \text{ConElse} \rangle & \rightarrow & \langle \text{OtrasSentencias} \rangle
\end{array}
$$

Demuestra que esta gramática es ambigua.

3.7. Estructuras de control repetitivas

Las estructuras de control repetitivas suelen tener una condición al principio:

$$\langle \text{Sentencia} \rangle \rightarrow \textbf{while } \langle \text{Condición} \rangle \textbf{ do } \langle \text{Sentencia} \rangle$$

o al final:

$$\langle \text{Sentencia} \rangle \rightarrow \textbf{repeat } \langle \text{ListaSentencias} \rangle \textbf{ until } \langle \text{Condición} \rangle$$

Si queremos que en un bucle del primer tipo se admitan varias sentencias, sin utilizar la sentencia compuesta, necesitamos algún tipo de terminador:

$$\langle \text{Sentencia} \rangle \rightarrow \textbf{while } \langle \text{Condición} \rangle \textbf{ do } \langle \text{ListaSentencias} \rangle \textbf{ endwhile}$$

Si dentro de la estructura de control se permite la salida prematura del bucle, por ejemplo con **break**, tenemos dos alternativas para especificar que sólo pueden aparecer dentro del bucle:

- Reflejarlo directamente en la gramática. El problema es que la solución inmediata es más complicada de lo que parece a primera vista. Si probamos con:

$$
\begin{array}{rcl}
\langle \text{Sentencia} \rangle & \rightarrow & \textbf{repeat } \langle \text{SentenciasConBreak} \rangle \textbf{ until } \langle \text{Condición} \rangle \\
\langle \text{SentenciasConBreak} \rangle & \rightarrow & \langle \text{Sentencia} \rangle \langle \text{SentenciasConBreak} \rangle \\
\langle \text{SentenciasConBreak} \rangle & \rightarrow & \textbf{break ;} \langle \text{SentenciasConBreak} \rangle \\
\langle \text{SentenciasConBreak} \rangle & \rightarrow & \lambda
\end{array}
$$

© Universitat Jaume I 2010-2011

Tenemos el problema de que **break** puede estar, por ejemplo, dentro de un condicional o una sentencia compuesta. Esto obliga a que esencialmente haya que replicar las reglas de ⟨Sentencia⟩ en un nuevo no terminal, por ejemplo ⟨SentenciaOBreak⟩ para incluir esta posibilidad.

- Tratar el **break** como una sentencia más y añadir una comprobación durante el análisis semántico.

Esto es un ejemplo de lo que suele suceder al diseñar el compilador: determinadas fases se pueden simplificar según cómo se hayan diseñado las otras. Aquí simplificamos el análisis sintáctico a costa de complicar ligeramente el semántico.

Ejercicio* 9

Reescribe la gramática siguiente para excluir la posibilidad de que **break** aparezca fuera del bucle.

⟨Programa⟩ → ⟨Declaraciones⟩⟨Compuesta⟩
⟨Declaraciones⟩ → **variable** ⟨Declaraciones⟩
⟨Declaraciones⟩ → **constant** ⟨Declaraciones⟩
⟨Declaraciones⟩ → λ
⟨Compuesta⟩ → **begin** ⟨ListaSentencias⟩ **end**
⟨ListaSentencias⟩ → ⟨Sentencia⟩⟨ListaSentencias⟩|λ
⟨Sentencia⟩ → ⟨Compuesta⟩
⟨Sentencia⟩ → **repeat** ⟨ListaSentencias⟩ **until** ⟨Expresión⟩
⟨Sentencia⟩ → **if** ⟨Expresión⟩ **then** ⟨Sentencia⟩
⟨Sentencia⟩ → **if** ⟨Expresión⟩ **then** ⟨Sentencia⟩ **else** ⟨Sentencia⟩
⟨Sentencia⟩ → **break**
⟨Sentencia⟩ → **other**
⟨Expresión⟩ → **id** | **cte**

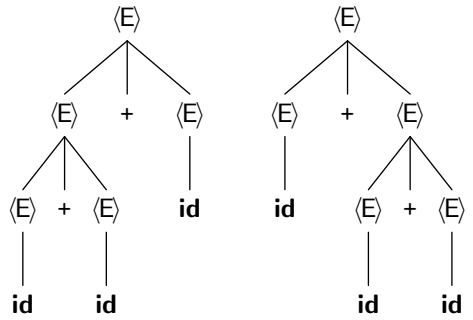
3.8. Expresiones aritméticas

Si queremos permitir operaciones en notación *infija* con los operadores de suma, resta, producto, división y exponenciación, podemos utilizar una gramática como la siguiente:

⟨E⟩ → ⟨E⟩+⟨E⟩
| ⟨E⟩−⟨E⟩
| ⟨E⟩∗⟨E⟩
| ⟨E⟩/⟨E⟩
| ⟨E⟩↑⟨E⟩
| “(”⟨E⟩“)”
| **id**

Como ya te habrás dado cuenta, por su similitud con el ejemplo de la sección 2.5, esta gramática

es ambigua. Por ejemplo, la expresión **id+id+id**, tiene dos árboles de análisis:



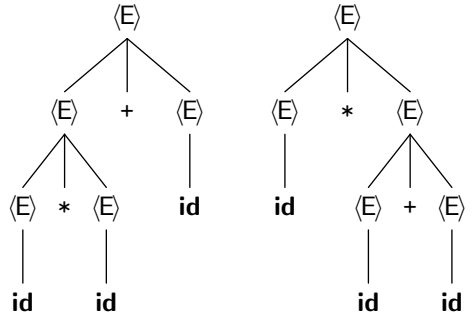
Lo habitual es elegir la primera interpretación, siguiendo las *reglas de asociatividad*. Normalmente, se sigue el siguiente convenio:

- Los operadores de suma, resta, producto y división son asociativos por la izquierda.
- El operador de exponenciación es asociativo por la derecha.

Ejercicio 10

¿Qué árbol de análisis asociamos a **id↑id↑id**?

Sin embargo, la asociatividad no nos permite resolver la ambigüedad que aparece cuando hay operadores distintos en la expresión. Por ejemplo, **id∗id+id** tiene dos árboles asociados:



Para estos casos utilizamos *reglas de prioridad*. Habitualmente:

- El operador de exponenciación es el más prioritario.
- Los operadores de multiplicación y división tienen la misma prioridad, que es menor que la de la exponenciación y mayor que la de la suma y resta.
- Los operadores de suma y resta tienen la misma prioridad, y ésta es la más baja.

Con estas reglas, el árbol que utilizaríamos sería el de la izquierda.

Ejercicio 11

¿Qué árbol de análisis corresponde a la cadena **id+id∗id↑id∗id+id**?

Para diseñar una gramática que recoja tanto la asociatividad como las prioridades, utilizaremos las siguientes ideas:

- Cada nivel de prioridad se asociará con un no terminal.
- La asociatividad por la izquierda se reflejará en recursividad por la izquierda; la asociatividad por la derecha en recursividad por la derecha.

En nuestro caso, tendremos que:

- Una expresión será bien la suma de una expresión con un término, bien la diferencia entre una expresión y un término, bien un único término.

© Universitat Jaume I 2010-2011

- Un término será bien el producto de un término por un factor, bien el cociente entre un término y un factor, bien un único factor.
- Un factor será bien una base elevada a un factor, bien una única base.
- Una base será un identificador o una expresión encerrada entre paréntesis.

De esta manera, nuestra gramática es:

$$\begin{array}{lcl} \langle \mathsf{E} \rangle & \to & \langle \mathsf{E} \rangle \mathsf{+} \langle \mathsf{T} \rangle \mid \langle \mathsf{E} \rangle \mathsf{-} \langle \mathsf{T} \rangle \mid \langle \mathsf{T} \rangle \\ \langle \mathsf{T} \rangle & \to & \langle \mathsf{T} \rangle \mathsf{*} \langle \mathsf{F} \rangle \mid \langle \mathsf{T} \rangle \mathsf{/} \langle \mathsf{F} \rangle \mid \langle \mathsf{F} \rangle \\ \langle \mathsf{F} \rangle & \to & \langle \mathsf{B} \rangle \uparrow \langle \mathsf{F} \rangle \mid \langle \mathsf{B} \rangle \\ \langle \mathsf{B} \rangle & \to & \mathbf{id} \mid \text{“(”} \langle \mathsf{E} \rangle \text{“)”} \end{array}$$

Ejercicio 12

Halla el árbol de análisis de **id*id↑(id+id)**.

El añadir operadores unarios no supone mayor complicación. Si queremos completar la gramática anterior para que acepte el operador de cambio de signo con una prioridad mayor que la exponenciación, haríamos:

$$\begin{array}{lcl} \langle \mathsf{E} \rangle & \to & \langle \mathsf{E} \rangle \mathsf{+} \langle \mathsf{T} \rangle \mid \langle \mathsf{E} \rangle \mathsf{-} \langle \mathsf{T} \rangle \mid \langle \mathsf{T} \rangle \\ \langle \mathsf{T} \rangle & \to & \langle \mathsf{T} \rangle \mathsf{*} \langle \mathsf{F} \rangle \mid \langle \mathsf{T} \rangle \mathsf{/} \langle \mathsf{F} \rangle \mid \langle \mathsf{F} \rangle \\ \langle \mathsf{F} \rangle & \to & \langle \mathsf{B} \rangle \uparrow \langle \mathsf{F} \rangle \mid \langle \mathsf{B} \rangle \\ \langle \mathsf{B} \rangle & \to & \mathbf{id} \mid \text{“(”} \langle \mathsf{E} \rangle \text{“)”} \mid \mathsf{-} \langle \mathsf{B} \rangle \end{array}$$

Ejercicio* 13

Analiza **-id↑-id** con la gramática anterior y con esta:

$$\begin{array}{lcl} \langle \mathsf{E} \rangle & \to & \langle \mathsf{E} \rangle \mathsf{+} \langle \mathsf{T} \rangle \mid \langle \mathsf{E} \rangle \mathsf{-} \langle \mathsf{T} \rangle \mid \langle \mathsf{T} \rangle \\ \langle \mathsf{T} \rangle & \to & \langle \mathsf{T} \rangle \mathsf{*} \langle \mathsf{F} \rangle \mid \langle \mathsf{T} \rangle \mathsf{/} \langle \mathsf{F} \rangle \mid \langle \mathsf{F} \rangle \\ \langle \mathsf{F} \rangle & \to & \langle \mathsf{B} \rangle \uparrow \langle \mathsf{F} \rangle \mid \mathsf{-} \langle \mathsf{F} \rangle \mid \langle \mathsf{B} \rangle \\ \langle \mathsf{B} \rangle & \to & \mathbf{id} \mid \text{“(”} \langle \mathsf{E} \rangle \text{“)”} \end{array}$$

¿Por qué son distintas?

Es habitual que los operadores no aparezcan totalmente separados sino en categorías que asocien operadores con características similares. Por ejemplo, en casi todos los contextos donde puede aparecer un **+** también puede aparecer un **-**. Esto nos lleva a agruparlos en la categoría **opad** (por operador aditivo). Si también creamos la categoría **opmul** para los operadores multiplicativos, nuestra gramática queda:

$$\begin{array}{lcl} \langle \mathsf{E} \rangle & \to & \langle \mathsf{E} \rangle \ \mathbf{opad} \ \langle \mathsf{T} \rangle \mid \langle \mathsf{T} \rangle \\ \langle \mathsf{T} \rangle & \to & \langle \mathsf{T} \rangle \ \mathbf{opmul} \ \langle \mathsf{F} \rangle \mid \langle \mathsf{F} \rangle \\ \langle \mathsf{F} \rangle & \to & \langle \mathsf{B} \rangle \uparrow \langle \mathsf{F} \rangle \mid \langle \mathsf{B} \rangle \\ \langle \mathsf{B} \rangle & \to & \mathbf{id} \mid \text{“(”} \langle \mathsf{E} \rangle \text{“)”} \mid \mathbf{opad} \ \langle \mathsf{B} \rangle \end{array}$$

Fíjate en que ahora se acepta como expresión válida **+id**. Para evitarlo, lo que se hace es que el analizador semántico la rechace. Eso tiene además la ventaja de que los mensajes de error del semántico suelen ser más informativos. Este es otro ejemplo de simplificación del análisis sintáctico a costa de una ligera complicación del semántico.

Ejercicio 14

Define conjuntos de producciones adecuados para modelar las siguientes construcciones de un lenguaje de programación:

- Una declaración de variables consiste en un identificador de tipo (**id**, **int** o **float**) seguido de una lista, separada por comas, de identificadores, cada uno de los cuales puede estar seguido del signo igual y una expresión de inicialización.
- Una declaración de función consiste en un identificador de tipo o en la palabra reservada **void**, un identificador que da nombre a la función, una lista de parámetros formales entre paréntesis y un cuerpo de función. Una lista de parámetros formales es una sucesión (posiblemente vacía) de parámetros formales separados por comas. Un argumento formal empieza opcionalmente por la palabra **var** seguida de un identificador, dos puntos y un identificador de tipo.
- Un elemento de un vector se referencia mediante un identificador seguido de uno o más índices. Cada índice es una expresión aritmética encerrada entre corchetes.

4. Extensiones de las gramáticas incontextuales

Ahora veremos cómo podemos extender la definición de gramática incontextual de modo que resulte más fácil escribirlas. La extensión que utilizaremos tiene el mismo poder expresivo que la definición habitual, por lo que conservará sus propiedades.

4.1. Definiciones

Una *gramática con partes derechas regulares* (GPDR) es una cuádrupla $G = (N, \Sigma, P, \langle \mathsf{S} \rangle)$ donde

- N es un alfabeto de *no terminales*.
- Σ es un alfabeto de *terminales*, disjunto con N.
- $P \subseteq N \times \mathcal{R}(N \cup \Sigma)$ es un conjunto de *producciones* o *reglas*.
- $\langle \mathsf{S} \rangle \in N$ es el *símbolo inicial*.

Usamos $\mathcal{R}(N \cup \Sigma)$ para representar el conjunto de las expresiones regulares básicas sobre $N \cup \Sigma$. Recuerda que las expresiones regulares básicas se construían a partir del conjunto vacío, la cadena vacía, los símbolos del alfabeto y las operaciones de disyunción, concatenación y clausura. Para simplificar la formulación, en esta parte, excluiremos el conjunto vacío del conjunto de expresiones regulares.

Así pues, una gramática con partes derechas regulares es similar a una gramática incontextual con la diferencia de que la parte derecha de las reglas contiene expresiones regulares formadas a partir de terminales y no terminales.

Dado que hemos modificado la definición de gramática, tendremos que cambiar la definición de derivación. Ahora las formas sentenciales son expresiones regulares sobre $N \cup \Sigma$. Las reglas que se aplican son:

1. $\alpha\langle \mathsf{A} \rangle\beta \Rightarrow \alpha\gamma\beta$, si $\langle \mathsf{A} \rangle \to \gamma$ pertenece a P. Esta es la definición habitual.
2. $\alpha(\rho_1|\rho_2|\ldots|\rho_n)\beta \Rightarrow \alpha\rho_i\beta$, si $1 \leq i \leq n$. Es decir, si tenemos una disyunción, podemos escoger una cualquiera de las alternativas.
3. $\alpha(\rho)^*\beta \Rightarrow \alpha\beta$. Si tenemos una clausura, podemos sustituirla por la cadena vacía.
4. $\alpha(\rho)^*\beta \Rightarrow \alpha\rho(\rho)^*\beta$. Si tenemos una clausura, podemos sustituirla por una copia de su base y una copia de ella misma.

© Universitat Jaume I 2010-2011

Como puedes imaginar, las reglas referentes a la clausura son las que dan su utilidad a las gramáticas con partes derechas regulares.

El lenguaje definido por una gramática con partes derechas regulares es:

$$L(G) = \{x \in \Sigma^* \mid \langle \mathsf{S} \rangle \stackrel{*}{\Rightarrow} x\}.$$

Esta es la misma definición que para las gramáticas incontextuales, como era de esperar.

Una GPDR para las declaraciones de variables consistentes en listas de identificadores seguidas por un tipo entero o real podría ser:

$$\begin{array}{lcl} \langle \mathsf{DeclVariables} \rangle & \rightarrow & \langle \mathsf{ListaVariables} \rangle \,\mathbf{:(\ integer \mid float\ );} \\ \langle \mathsf{ListaVariables} \rangle & \rightarrow & \mathbf{id\ (,\ id)}^* \end{array}$$

Con esta gramática, la derivación por la izquierda de **id , id : integer ;** es:

$$\begin{array}{rcl} \langle \mathsf{DeclVariables} \rangle & \stackrel{1}{\Rightarrow} & \langle \mathsf{ListaVariables} \rangle \,\mathbf{:(\ integer \mid float\ );} \\ & \stackrel{1}{\Rightarrow} & \mathbf{id\ (,\ id)}^* \mathbf{:(\ integer \mid float\ );} \\ & \stackrel{4}{\Rightarrow} & \mathbf{id\ ,\ id\ (,\ id)}^* \mathbf{:(\ integer \mid float\ );} \\ & \stackrel{3}{\Rightarrow} & \mathbf{id\ ,\ id :(\ integer \mid float\ );} \\ & \stackrel{2}{\Rightarrow} & \mathbf{id\ ,\ id :\ integer ;} \end{array}$$

(El número encima de las flechas indica cuál de las reglas para la derivación se ha aplicado.)

4.2. Equivalencia con las gramáticas incontextuales

Hemos afirmado que las GPDR tienen el mismo poder expresivo que las incontextuales. Esto quiere decir que si un lenguaje tiene una gramática incontextual que lo represente, también tendrá una GPDR. Análogamente, si un lenguaje tiene una GPDR, también tendrá una gramática incontextual. Ahora justificaremos estas afirmaciones.

En primer lugar, está claro que toda gramática incontextual es un caso particular de GPDR. Así la primera parte está trivialmente demostrada.

En cuanto a la segunda parte, utilizaremos un procedimiento constructivo. Empezamos con las reglas de nuestra gramática. Iremos sustituyendo aquellas reglas que utilicen extensiones por otras que tengan menos extensiones. Finalmente, todas las reglas tendrán en la parte derecha únicamente secuencias de terminales y no terminales, con lo que habremos obtenido la gramática incontextual equivalente.

Si la regla tiene la forma $\langle \mathsf{A} \rangle \rightarrow \alpha(\rho_1|\rho_2|\ldots|\rho_n)\beta$, creamos un nuevo no terminal $\langle \mathsf{N} \rangle$ y sustituimos la regla por el conjunto de reglas:

$$\begin{array}{lcl} \langle \mathsf{A} \rangle & \rightarrow & \alpha\langle \mathsf{N} \rangle\beta \\ \langle \mathsf{N} \rangle & \rightarrow & \rho_1 \\ \langle \mathsf{N} \rangle & \rightarrow & \rho_2 \\ \ldots & & \\ \langle \mathsf{N} \rangle & \rightarrow & \rho_n \end{array}$$

Si la regla tiene la forma $\langle \mathsf{A} \rangle \rightarrow \alpha(\rho)^*\beta$, creamos un nuevo no terminal $\langle \mathsf{N} \rangle$ y sustituimos la regla por el conjunto de reglas:

$$\begin{array}{lcl} \langle \mathsf{A} \rangle & \rightarrow & \alpha\langle \mathsf{N} \rangle\beta \\ \langle \mathsf{N} \rangle & \rightarrow & \rho\langle \mathsf{N} \rangle \\ \langle \mathsf{N} \rangle & \rightarrow & \lambda \end{array}$$

Es fácil demostrar que estas transformaciones no cambian el lenguaje generado por la gramática. Además, cada vez que aplicamos una de las transformaciones se reduce en uno el número de extensiones de la gramática. Por lo tanto, si repetimos el proceso suficientes veces obtendremos una gramática incontextual sin partes derechas regulares.

Así, la gramática que hemos empleado para las declaraciones de variables es equivalente a:

$$\begin{aligned}
\langle \text{DeclVariables} \rangle &\rightarrow \langle \text{ListaVariables} \rangle \,\mathbf{:}\, \langle \text{DeclVariables2} \rangle \,\mathbf{;} \\
\langle \text{DeclVariables2} \rangle &\rightarrow \mathbf{integer} \mid \mathbf{float} \\
\langle \text{ListaVariables} \rangle &\rightarrow \mathbf{id}\ \langle \text{ListaVariables2} \rangle \\
\langle \text{ListaVariables2} \rangle &\rightarrow \mathbf{,}\ \mathbf{id}\ \langle \text{ListaVariables2} \rangle \\
\langle \text{ListaVariables2} \rangle &\rightarrow \lambda
\end{aligned}$$

Ejercicio 15

Transforma la siguiente GPDR en una gramática incontextual:

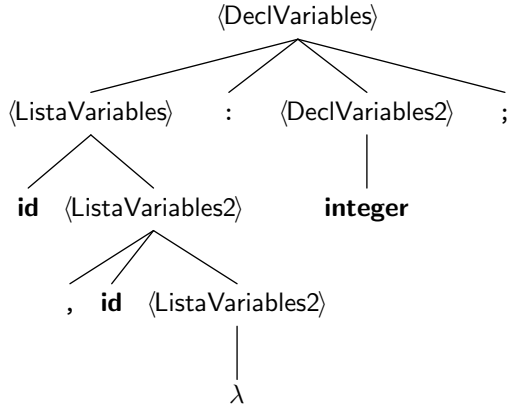
$$\begin{aligned}
\langle \text{S} \rangle &\rightarrow \big((\mathbf{print}\ \langle \text{Expr} \rangle | \mathbf{int}\ \langle \text{ListaId} \rangle)\,\mathbf{;} \big)^* \\
\langle \text{Expr} \rangle &\rightarrow \mathbf{id}(\mathbf{+}\ \mathbf{id})^* \\
\langle \text{ListaId} \rangle &\rightarrow \mathbf{id}(\mathbf{,}\ \mathbf{id})^*
\end{aligned}$$

Encuentra una derivación de **int id , id ; print id ;** con la gramática original y con la nueva.

4.3. Árboles de análisis

A la hora de dibujar el árbol de derivación de una sentencia del lenguaje, existen tres posibilidades, que reflejan dos formas de interpretar las GPDR. La primera opción es dibujar el árbol como si se hubiera transformado la GPDR en una gramática incontextual equivalente. También se puede escribir un árbol equivalente en el que los no terminales que se introducen al construir la gramática equivalente son sustituidos por las correspondientes expresiones regulares. En ambos casos, estamos considerando que la GPDR es una gramática incontextual escrita de forma compacta. Otra opción es considerar que las expresiones regulares en las partes derechas suponen de hecho condensar un número potencialmente infinito de reglas. En este caso, el número de hijos de cada no terminal no está limitado.

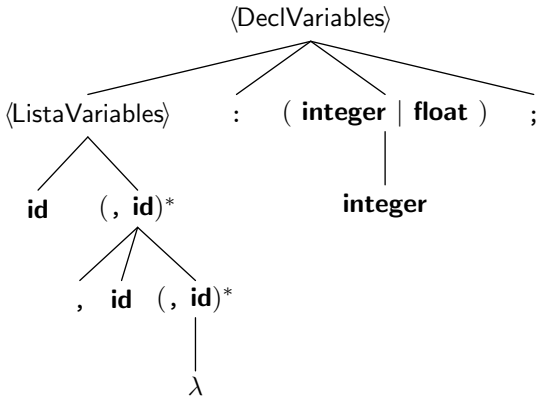
Según la primera interpretación, el árbol de análisis de **id , id : integer ;** con la GPDR para las declaraciones sería:



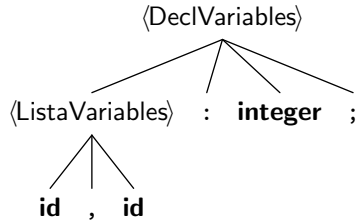
También es posible no introducir nuevos no terminales si escribimos explícitamente las expre-

© Universitat Jaume I 2010-2011

siones regulares:



Sin embargo, si utilizamos la segunda interpretación, obtenemos:



Esta última interpretación es la que seguiremos las anteriores introducen una estructura que realmente no estaba presente en la gramática.

Ejercicio 16

Construye el árbol de análisis de **int id , id ; print id ;** con la gramática del ejercicio 15. Utiliza las tres posibilidades.

4.4. Reescritura de algunas construcciones

Ahora repasaremos algunas de las construcciones que hemos visto antes, adaptándolas a las GPDR.

4.4.1. Secuencias de instrucciones

Las secuencias de instrucciones separadas por punto y coma, las podemos expresar mediante la regla:

$$\langle \text{ListaSentencias} \rangle \quad \rightarrow \quad \langle \text{Sentencia} \rangle (\, ; \langle \text{Sentencia} \rangle)^*$$

Si queremos sentencias terminadas en punto y coma, con el punto y coma explícito:

$$\langle \text{ListaSentencias} \rangle \quad \rightarrow \quad (\langle \text{Sentencia} \rangle \, ;)^*$$

Ejercicio 17

La regla anterior permite que la secuencia de sentencias sea vacía, ¿cómo puedes hacer que haya al menos una sentencia?

4.4.2. Declaraciones de variables

Si tenemos un tipo precediendo a una lista de identificadores (C):

$$\langle \text{DeclVariables} \rangle \quad \rightarrow \quad \langle \text{Tipo} \rangle \ \langle \text{Id} \rangle \ (\, , \langle \text{Id} \rangle)^* \, ;$$

El caso del tipo siguiendo a la lista de identificadores (Pascal):

$$\langle \text{DeclVariables} \rangle \quad \rightarrow \quad \textbf{id} (\, , \textbf{id})^* : \langle \text{Tipo} \rangle$$

4.4.3. Expresiones aritméticas

Podemos simplificar las producciones correspondientes a las expresiones aritméticas. Sin embargo, esta simplificación tiene un precio; no podremos expresar directamente en la gramática la asociatividad de los operadores. Tendremos que relegar en fases posteriores la responsabilidad de definir la asociatividad. En la práctica esto no supondrá ningún problema. Sin embargo, seguiremos teniendo un no terminal por cada nivel de prioridad. La gramática que teníamos para las expresiones la podemos reescribir así:

$$\begin{aligned}
\langle \mathsf{E} \rangle &\rightarrow \langle \mathsf{T} \rangle ((\mathtt{+}|\mathtt{-})\langle \mathsf{T} \rangle)^* \\
\langle \mathsf{T} \rangle &\rightarrow \langle \mathsf{F} \rangle ((\mathtt{*}|\mathtt{/})\langle \mathsf{F} \rangle)^* \\
\langle \mathsf{F} \rangle &\rightarrow \langle \mathsf{B} \rangle (\uparrow \langle \mathsf{B} \rangle)^* \\
\langle \mathsf{B} \rangle &\rightarrow \mathbf{id} | \text{“(”} \langle \mathsf{E} \rangle \text{“)”}
\end{aligned}$$

Si agrupamos + junto con - en la categoría léxica **opad** (operador aditivo) y * junto con / en **opmult** (operador multiplicativo), nos queda una gramática ligeramente más clara:

$$\begin{aligned}
\langle \mathsf{E} \rangle &\rightarrow \langle \mathsf{T} \rangle (\mathbf{opad}\ \langle \mathsf{T} \rangle)^* \\
\langle \mathsf{T} \rangle &\rightarrow \langle \mathsf{F} \rangle (\mathbf{opmult}\ \langle \mathsf{F} \rangle)^* \\
\langle \mathsf{F} \rangle &\rightarrow \langle \mathsf{B} \rangle (\uparrow \langle \mathsf{B} \rangle)^* \\
\langle \mathsf{B} \rangle &\rightarrow \mathbf{id} | \text{“(”} \langle \mathsf{E} \rangle \text{“)”}
\end{aligned}$$

Ejercicio 18

Halla el árbol de análisis de **id**∗**id**↑(**id**+**id**).

Ejercicio 19

Aumenta la gramática para permitir operadores unarios. Dales la máxima prioridad.

Ejercicio 20

Reescribe las construcciones del ejercicio 14 utilizando GPDR.

5. Análisis descendente recursivo

Una vez definido nuestro lenguaje mediante una gramática G, estamos interesados en resolver el siguiente problema:

> Dada una cadena x y una gramática G, ¿pertenece x a $L(G)$?

Esto es lo que se conoce como el *problema del análisis*.

Existen diversos algoritmos que resuelven este problema de manera general. Dos de los más conocidos son:

Algoritmo de Cocke-Younger-Kasami o CYK. Este algoritmo utiliza el esquema algorítmico de Programación Dinámica. Su coste es $O(|x|^3)$ y necesita una gramática en forma normal de Chomsky.

Algoritmo de Early este algoritmo tiene un coste igual al CYK si la gramática es ambigua. Si no lo es, el coste se reduce a $O(|x|^2)$ aunque, para muchas gramáticas, el coste es $O(|x|)$.

Nuestro objetivo será buscar algoritmos que garanticen un coste lineal con la entrada. Para ello, tendremos que poner algunas restricciones a las gramáticas que se podrán utilizar.

Dependiendo del modo en que se construye el árbol de análisis, distinguimos dos familias de algoritmos de análisis:

Análisis descendente se construye el árbol partiendo de la raíz hasta llegar a las hojas. En este proceso se va “prediciendo” lo que vendrá después, por lo que también se le denomina *análisis predictivo*.

© Universitat Jaume I 2010-2011

Análisis ascendente se construye el árbol desde las hojas. La detección de las construcciones se realiza una vez que se han visto completamente.

En este tema estudiaremos un tipo de análisis descendente: el análisis LL(1), que es fácil de implementar manualmente y es razonablemente potente; a final de curso, estudiaremos métodos de análisis ascendente. Comenzaremos por ver cómo analizar las gramáticas incontextuales y después pasaremos a las GPDR.

5.1. Un ejemplo

Supongamos que tenemos la siguiente gramática:

$$\begin{aligned}
\langle S\rangle &\rightarrow \textbf{if } \langle E\rangle \textbf{ then } \langle S\rangle \\
\langle S\rangle &\rightarrow \textbf{while } \langle E\rangle \textbf{ do } \langle S\rangle \\
\langle S\rangle &\rightarrow \textbf{id}:=\langle E\rangle \\
\langle E\rangle &\rightarrow \textbf{id}|\textbf{cte}
\end{aligned}$$

Vamos a crear un objeto Python que comprobará si un programa es o no correcto. Para ello, supondremos que ya hemos implementado un analizador léxico que tiene un método `siguiente` que devuelve un objeto con el atributo `cat` que indica la categoría del componente devuelto. En nuestro caso, las categorías posibles son: `asig`, `cte`, `do`, `id`, `if`, `then` y `while`. Además, tendremos la categoría `eof` para marcar el final de la entrada.

El constructor de nuestro objeto recibirá como único parámetro el analizador léxico, intentará analizar la cadena de entrada y después se asegurará de que se ha terminado la entrada. El código correspondiente es:

```
2     def __init__(self, lexico):
3       self.lexico= lexico
4       self.avanza()
5       self.analizaS()
6       self.comprueba("eof")
7
```

Podemos ver tres llamadas a métodos:

- `avanza()` se utiliza para llamar `siguiente` sobre el analizador léxico y guardar el resultado en `self.componente`:

```
8     def avanza(self):
9       self.componente= self.lexico.siguiente()
10
```

- `analizaS()` se emplea para ver si es posible considerar la entrada como una cadena generada por $\langle S\rangle$.
- `comprueba(cat)` se emplea para comprobar que el componente leído tiene la categoría que interesa y avanzar en la entrada si es así. En caso contrario llama a `self.error`:

```
11    def comprueba(self, cat):
12      if self.componente.cat== cat:
13        self.avanza()
14      else:
15        self.error()
16
```

Lógicamente, el método más interesante es `analizaS`. Pensemos en cómo puede decidir cuál de las alternativas para $\langle S\rangle$ podemos encontrar en la entrada. Si observamos el componente actual y vemos que es un **if**, tendremos que utilizar la primera regla; si fuera un **while**, la segunda; si

fuera un **id**, la tercera; en cualquier otro caso habríamos detectado un error. Así, el esqueleto de `self.analizaS` será:

```
def analizaS(self):
  if self.componente.cat== "if":
    # <S> -> if <E> then <S>
    ...
  elif self.componente.cat== "while":
    # <S> -> while <E> do <S>
    ...
  elif self.componente.cat=="id":
    # <S> -> id := <E>
    ...
  else:
    self.error()
```

Y ¿cómo analizamos cada una de las reglas? Mediante una secuencia en la que reflejamos los componentes de la regla en orden: cada terminal está representado mediante una llamada a `comprueba` y cada no terminal mediante una llamada al método `analiza` correspondiente. Antes debemos llamar a `avanza` puesto que ya hemos visto que el terminal era adecuado. Por ejemplo, para el caso del **if**, obtenemos:

```
# <S> -> if <E> then <S>
self.avanza()
self.analizaE()
self.comprueba("then")
self.analizaS()
```

El método `analizaS` completo es:

```
20    def analizaS(self):
21      if self.componente.cat== "if":
22        # <S> -> if <E> then <S>
23        self.avanza()
24        self.analizaE()
25        self.comprueba("then")
26        self.analizaS()
27      elif self.componente.cat== "while":
28        # <S> -> while <E> do <S>
29        self.avanza()
30        self.analizaE()
31        self.comprueba("do")
32        self.analizaS()
33      elif self.componente.cat=="id":
34        # <S> -> id := <E>
35        self.avanza()
36        self.comprueba("asig")
37        self.analizaE()
38      else:
39        self.error()
40
```

Podemos hacer lo mismo con `analizaE`. El código completo del analizador está en la figura 1.

5.2. Introducción al concepto de gramática LL(1)

La gramática del ejemplo anterior nos ha permitido construir fácilmente el analizador porque tiene una característica muy importante: cuando queremos analizar si una cadena se puede derivar

© Universitat Jaume I 2010-2011

```
1   class Analizador:
2     def __init__(self, lexico):
3       self.lexico= lexico
4       self.avanza()
5       self.analizaS()
6       self.comprueba("eof")
7
8     def avanza(self):
9       self.componente= self.lexico.siguiente()
10
11    def comprueba(self, cat):
12      if self.componente.cat== cat:
13        self.avanza()
14      else:
15        self.error()
16
17    def error(self):
18      raise ErrorSintactico
19
20    def analizaS(self):
21      if self.componente.cat== "if":
22        # <S> -> if <E> then <S>
23        self.avanza()
24        self.analizaE()
25        self.comprueba("then")
26        self.analizaS()
27      elif self.componente.cat== "while":
28        # <S> -> while <E> do <S>
29        self.avanza()
30        self.analizaE()
31        self.comprueba("do")
32        self.analizaS()
33      elif self.componente.cat=="id":
34        # <S> -> id := <E>
35        self.avanza()
36        self.comprueba("asig")
37        self.analizaE()
38      else:
39        self.error()
40
41    def analizaE(self):
42      if self.componente.cat== "id":
43        # <E> -> id
44        self.avanza()
45      elif self.componente.cat== "cte":
46        # <E> -> cte
47        self.avanza()
48      else:
49        self.error()
```

Figura 1: Código del analizador sintáctico para la gramática de la página 18.

de un no terminal, podemos elegir entre las posibles reglas de ese no terminal mirando sólo el primer símbolo de la cadena.

Más adelante, daremos una definición más formal (y completa) de esta propiedad y a las gramáticas que la cumplan, las llamaremos gramáticas LL(1). El nombre viene de que se pueden interpretar que el algoritmo de análisis lee la gramática de izquierda a derecha (Left to Right, primera L), busca una derivación canónica por la izquierda (Left, segunda L) y utiliza un símbolo de anticipación (el uno).

Fíjate, además en que podremos estar seguros de que una gramática así es determinista.

5.3. Cálculo de primeros

En el ejemplo anterior hemos podido decidir en cada caso qué producción aplicar porque los primeros símbolos de cada alternativa eran terminales diferentes. Sin embargo, exigir esta condición para que podamos analizar cadenas con una gramática es excesivo. Vamos a ver cómo podemos relajarla sin perder la capacidad de escribir un analizador.

Supongamos que modificamos la gramática anterior para permitir que se emplee el operador unario `*` de modo que sea posible utilizar punteros. La nueva gramática tiene este aspecto:

$$
\begin{aligned}
\langle \mathsf{S} \rangle &\rightarrow \textbf{if } \langle \mathsf{E} \rangle \textbf{ then } \langle \mathsf{S} \rangle \,|\, \textbf{while } \langle \mathsf{E} \rangle \textbf{ do } \langle \mathsf{S} \rangle \,|\, \langle \mathsf{I} \rangle := \langle \mathsf{E} \rangle \\
\langle \mathsf{E} \rangle &\rightarrow \langle \mathsf{I} \rangle \,|\, \textbf{cte} \\
\langle \mathsf{I} \rangle &\rightarrow *\langle \mathsf{I} \rangle \,|\, \textbf{id}
\end{aligned}
$$

Ahora tenemos un problema para implementar `analizaS` porque la regla $\langle \mathsf{S} \rangle \rightarrow \langle \mathsf{I} \rangle := \langle \mathsf{E} \rangle$ no tiene un terminal al principio. Sin embargo, un análisis rápido nos permite ver que las cadenas producidas por $\langle \mathsf{I} \rangle$ sólo pueden comenzar por `*` o por **id**. Así pues, podemos modificar `analizaS` y dejarlo de la siguiente forma:

```
20      def analizaS(self):
21        if self.componente.cat== "if":
22          # <S> -> if <E> then <S>
23          self.avanza()
24          self.analizaE()
25          self.comprueba("then")
26          self.analizaS()
27        elif self.componente.cat== "while":
28          # <S> -> while <E> do <S>
29          self.avanza()
30          self.analizaE()
31          self.comprueba("do")
32          self.analizaS()
33        elif self.componente.cat in ["id", "asterisco"]:
34          # <S> -> <I> := <E>
35          self.analizaI()
36          self.comprueba("asig")
37          self.analizaE()
38        else:
39          self.error()
40
```

Vamos a formalizar la intuición anterior. En primer lugar, para decidir entre las alternativas de una producción, debemos saber cómo empiezan las cadenas que puede generar. Si tenemos una forma sentencial α, podemos definir primeros(α) como:

$$\text{primeros}(\alpha) = \{\mathtt{a} \in \Sigma \mid \alpha \stackrel{*}{\Rightarrow} \mathtt{a}\beta\}.$$

Es decir, primeros(α) es el conjunto de terminales que pueden ser el primer símbolo de las cadenas generadas por α[1].

Para escribir el algoritmo de cálculo de los primeros de una forma sentencial, necesitamos el concepto de *anulable*. Decimos que una forma sentencial es anulable (lo representamos como anulable(α)) si de ella se puede derivar la cadena vacía.

Si la gramática no tiene recursividad por la izquierda, podemos calcular el conjunto de primeros de α mediante el siguiente algoritmo recursivo:

Algoritmo primeros $(\alpha : (N \cup \Sigma)^*)$
  $C = \emptyset$
  si $\alpha = \mathtt{a}\beta$ **entonces**
    $C := \{\mathtt{a}\}$;
  si no si $\alpha = \langle \mathsf{A} \rangle \beta$ **entonces**
    para todo $\langle \mathsf{A} \rangle \to \gamma \in P$ **hacer**
      $C := C \cup \text{primeros}(\gamma)$;
    fin para
    si anulable($\langle \mathsf{A} \rangle$) **entonces**
      $C := C \cup \text{primeros}(\beta)$;
    fin si
  fin si
  devolver C;
fin primeros

El funcionamiento del algoritmo es sencillo:

- Si α comienza por un terminal, éste es el único primero.

[1] En algunos textos, cuando la cadena vacía puede ser derivada de α, se la considera parte de los primeros de α. Esto supone una leve modificación de la formalización.

© Universitat Jaume I 2010-2011

- Si α comienza por el no terminal $\langle \mathsf{A} \rangle$, tendremos que obtener los primeros de $\langle \mathsf{A} \rangle$. Además, si $\langle \mathsf{A} \rangle$ es anulable también habrá que mirar los primeros de lo que le sigue.

Podemos averiguar si α es anulable mediante el siguiente algoritmo:

Algoritmo anulable ($\alpha : (N \cup \Sigma)^*$)
  si $\alpha = \lambda$ **entonces**
    devolver cierto;
  si no si $\alpha = \mathsf{a}\beta$ **entonces**
    devolver falso;
  si no si $\alpha = \langle \mathsf{A} \rangle \beta$ **entonces**
    si existe γ tal que $\langle \mathsf{A} \rangle \to \gamma \in P$ y anulable(γ) **entonces**
      devolver anulable(β);
    si no
      devolver falso;
    fin si
  fin si
fin anulable

Ejercicio 21

Calcula los primeros de los no terminales de la siguiente gramática:

$$\begin{aligned} \langle \mathsf{S} \rangle &\to \langle \mathsf{A} \rangle \langle \mathsf{B} \rangle | \langle \mathsf{B} \rangle \ \mathbf{c} \\ \langle \mathsf{A} \rangle &\to \mathbf{a\ b} \mid \mathbf{c\ d} \\ \langle \mathsf{B} \rangle &\to \mathbf{b\ a} \mid \lambda \end{aligned}$$

Estos algoritmos sirven si la gramática no tiene recursividad a izquierdas. En caso contrario, los cálculos de primeros y anulables se complican ligeramente (mira el ejercicio 23). Sin embargo, dado que las gramáticas con recursividad a izquierdas no son LL(1), no estudiaremos ese caso aquí y veremos más adelante cómo eliminar la recursividad a izquierdas.

Ahora podemos escribir el método de análisis de un no terminal de la siguiente manera:

- Calculamos los primeros de las partes derechas de las reglas de la gramática.
- Después, escribimos un condicional compuesto que compara `self.componente` con cada uno de los posibles primeros. Cuando la condición es cierta, se transcribe la regla como hasta ahora. El `else` corresponde a un error sintáctico.

Como ya teníamos el método `analizaS`, nos faltarán los correspondientes a $\langle \mathsf{E} \rangle$:

```
41    def analizaE(self):
42      if self.componente.cat in ["id", "asterisco"]:
43        # <E> -> <I>
44        self.analizaI()
45      elif self.componente.cat== "cte":
46        # <E> -> cte
47        self.avanza()
48      else:
49        self.error()
50
```

y a $\langle \mathsf{I} \rangle$:

```
51    def analizaI(self):
52      if self.componente.cat== "asterisco":
53        # <I> -> *<I>
```

```
54          self.avanza()
55          self.analizaI()
56        elif self.componente.cat== "id":
57          # <I> -> id
58          self.avanza()
59        else:
60          self.error()
```

5.4. Cálculo de siguientes

¿Qué sucede cuando algún no terminal es anulable? Vamos a estudiarlo aumentando la gramática de nuestro ejemplo para permitir accesos a vectores:

$$\begin{aligned}
\langle \mathsf{S}\rangle &\rightarrow \textbf{if } \langle \mathsf{E}\rangle \textbf{ then } \langle \mathsf{S}\rangle \,|\, \textbf{while } \langle \mathsf{E}\rangle \textbf{ do } \langle \mathsf{S}\rangle \,|\, \langle \mathsf{I}\rangle \,\mathtt{:=}\, \langle \mathsf{E}\rangle \\
\langle \mathsf{E}\rangle &\rightarrow \langle \mathsf{I}\rangle \,|\, \textbf{cte} \\
\langle \mathsf{I}\rangle &\rightarrow \mathtt{*}\langle \mathsf{I}\rangle \,|\, \textbf{id } \langle \mathsf{A}\rangle \\
\langle \mathsf{A}\rangle &\rightarrow \mathtt{[}\langle \mathsf{E}\rangle \mathtt{]} \,|\, \lambda
\end{aligned}$$

El problema se nos plantea al implementar `analizaA`. Está claro que si `self.componente` es un corchete abierto, debemos avanzar, analizar $\langle \mathsf{E}\rangle$ y comprobar el corchete cerrado. El problema es decidir cuándo optamos por la cadena vacía. Lo que tendremos que hacer es optar por esta regla cuando `self.componente` corresponda a un componente que "pueda seguir" a $\langle \mathsf{A}\rangle$. De una manera más formal, Llamaremos *siguientes* de un no terminal $\langle \mathsf{A}\rangle$ a los miembros del conjunto:

$$\text{siguientes}(\langle \mathsf{A}\rangle) = \begin{cases} \{\mathtt{a} \in \Sigma \,|\, \langle \mathsf{S}\rangle \overset{*}{\Rightarrow} \alpha\langle \mathsf{A}\rangle\beta,\ \mathtt{a} \in \text{primeros}(\beta)\} \cup \{\$\} & \text{si } \langle \mathsf{S}\rangle \overset{*}{\Rightarrow} \alpha\langle \mathsf{A}\rangle, \\ \{\mathtt{a} \in \Sigma \,|\, \langle \mathsf{S}\rangle \overset{*}{\Rightarrow} \alpha\langle \mathsf{A}\rangle\beta,\ \mathtt{a} \in \text{primeros}(\beta)\} & \text{si no.} \end{cases}$$

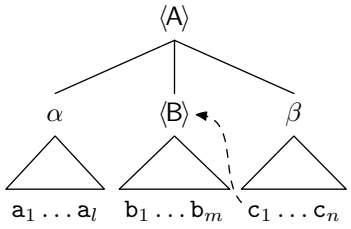
La idea es que los siguientes de un no terminal $\langle \mathsf{A}\rangle$ son aquellos terminales que pueden llegar a aparecer detrás de $\langle \mathsf{A}\rangle$ en alguna forma sentencial. En el caso del símbolo inicial y de aquellos que pueden aparecer al final de una forma sentencial, además incluimos el fin de la entrada.

Para encontrar los siguientes de cada no terminal, podemos emplear las siguientes ecuaciones:

- En primer lugar:
$$\{\$\} \subseteq \text{siguientes}(\langle \mathsf{S}\rangle). \tag{1}$$
- Si $\langle \mathsf{A}\rangle \rightarrow \alpha\langle \mathsf{B}\rangle\beta \in P$, entonces:
$$\text{primeros}(\beta) \subseteq \text{siguientes}(\langle \mathsf{B}\rangle); \tag{2}$$
- si, además, anulable(β) entonces:
$$\text{siguientes}(\langle \mathsf{A}\rangle) \subseteq \text{siguientes}(\langle \mathsf{B}\rangle). \tag{3}$$

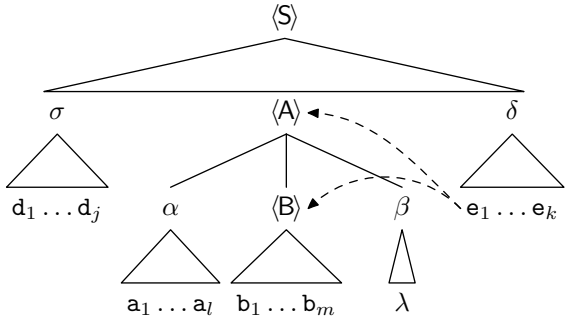
La ecuación (1) es fácil de entender: cualquier cadena generada a partir del símbolo inicial estará seguida del fin de la entrada.

Para entender (2), nos planteamos cómo se genera la cadena $\mathtt{a}_1 \ldots \mathtt{a}_l \mathtt{b}_1 \ldots \mathtt{b}_m \mathtt{c}_1 \ldots \mathtt{c}_n$ a partir de $\langle \mathsf{A}\rangle$:



© Universitat Jaume I 2010-2011

Es fácil ver qué "va detrás" (los siguientes) de ⟨B⟩: los primeros de β. Por otro lado, si β puede anularse, tenemos una situación como la esta:



Con lo que ⟨B⟩ y ⟨A⟩ comparten siguientes, lo que explica la ecuación (3).
Para resolver las ecuaciones podemos ir aplicándolas en tres pasos:

1. Como inicialización, asociamos a todos los no terminales distintos del inicial el conjunto vacío y al símbolo inicial le asociamos el conjunto {**$**} para que se cumpla la ecuación (1).
2. Para cumplir con la ecuación (2), añadimos al conjunto de siguientes de cada no terminal ⟨A⟩ los primeros de los β tales que $\alpha\langle A\rangle\beta$ constituye la parte derecha de alguna regla.
3. Recorremos ordenadamente las reglas aplicando la ecuación (3). Para ello, por cada regla $\langle A\rangle \to \alpha\langle B\rangle\beta$ con β anulable, añadimos los siguientes de ⟨A⟩ a los de ⟨B⟩. Este paso tendremos que repetirlo hasta que no cambie ninguno de los conjuntos de siguientes en un recorrido completo de las reglas.

Vamos a aplicar el algoritmo a nuestro ejemplo. En primer lugar, inicializamos los siguientes al conjunto vacío salvo para el símbolo inicial:

No terminal	Siguientes
⟨S⟩	{**$**}
⟨E⟩	∅
⟨I⟩	∅
⟨A⟩	∅

Después, añadimos los símbolos que aparecen inmediatamente después de cada no terminal:

No terminal	Siguientes
⟨S⟩	{**$**}
⟨E⟩	{**]**, **do**, **then**}
⟨I⟩	{**:=**}
⟨A⟩	∅

Ahora aplicamos ordenadamente la ecuación (3) hasta que no haya modificaciones. Así, a partir de $\langle S\rangle \to \langle I\rangle$ **:=** $\langle E\rangle$, aumentamos los siguientes de ⟨E⟩:

No terminal	Siguientes
⟨S⟩	{**$**}
⟨E⟩	{**]**, **$**, **do**, **then**}
⟨I⟩	{**:=**}
⟨A⟩	∅

Con la regla ⟨E⟩ → ⟨I⟩, aumentamos los siguientes de ⟨I⟩:

No terminal	Siguientes
⟨S⟩	{**$**}
⟨E⟩	{**]**, **$**, **do**, **then**}
⟨I⟩	{**:=**, **]**, **$**, **do**, **then**}
⟨A⟩	∅

A partir de ⟨I⟩ → **id**⟨A⟩, aumentamos los siguientes de ⟨A⟩:

No terminal	Siguientes
⟨S⟩	{**$**}
⟨E⟩	{**]**, **$**, **do**, **then**}
⟨I⟩	{**:=**, **]**, **$**, **do**, **then**}
⟨A⟩	{**:=**, **]**, **$**, **do**, **then**}

Si volvemos a repasar las reglas, vemos que los siguientes no varían, así que podemos dar por terminado el proceso.

Ahora podemos implementar `analizaA`:

```
63    def analizaA(self):
64      if self.componente.cat== "acor":
65        # <A> -> [ <E> ]
66        self.avanza()
67        self.analizaE()
68        self.comprueba("ccor")
69      elif self.componente.cat in [ "asig", "ccor", "eof", "do", "then" ]:
70        # <A> -> lambda
71        pass
72      else:
73        self.error()
```

En algunos casos, hay que repetir el tercer paso más veces. Por ejemplo, vamos a calcular los siguientes para la gramática:

$$\begin{aligned} \langle A \rangle &\rightarrow \langle B \rangle\ \mathbf{c} \mid \langle C \rangle\ \mathbf{d} \\ \langle B \rangle &\rightarrow \mathbf{b} \mid \mathbf{a}\ \langle A \rangle \\ \langle C \rangle &\rightarrow \mathbf{a}\ \langle B \rangle \end{aligned}$$

Inicialmente, tenemos vacíos los conjuntos de siguientes, salvo el de ⟨A⟩:

No terminal	Siguientes
⟨A⟩	{**$**}
⟨B⟩	∅
⟨C⟩	∅

Ahora, por la ecuación (2), añadimos `c` a los siguientes de ⟨B⟩ y `d` a los de ⟨C⟩:

No terminal	Siguientes
⟨A⟩	{**$**}
⟨B⟩	{**c**}
⟨C⟩	{**d**}

© Universitat Jaume I 2010-2011

La ecuación (3) sólo se puede aplicar a $\langle B \rangle \rightarrow \mathbf{a}\ \langle A \rangle$ y $\langle C \rangle \rightarrow \mathbf{a}\ \langle B \rangle$. Así que, copiamos los siguientes de $\langle B \rangle$ en los de $\langle A \rangle$:

No terminal	Siguientes
$\langle A \rangle$	$\{\mathbf{c}, \mathbf{\$}\}$
$\langle B \rangle$	$\{\mathbf{c}\}$
$\langle C \rangle$	$\{\mathbf{d}\}$

Y los de $\langle C \rangle$ en los de $\langle B \rangle$:

No terminal	Siguientes
$\langle A \rangle$	$\{\mathbf{c}, \mathbf{\$}\}$
$\langle B \rangle$	$\{\mathbf{c}, \mathbf{d}\}$
$\langle C \rangle$	$\{\mathbf{d}\}$

Y todavía no hemos terminado. Como ha habido modificaciones en el último recorrido, tenemos que dar una nueva pasada. Copiamos los siguientes de $\langle B \rangle$ en los de $\langle A \rangle$:

No terminal	Siguientes
$\langle A \rangle$	$\{\mathbf{c}, \mathbf{d}, \mathbf{\$}\}$
$\langle B \rangle$	$\{\mathbf{c}, \mathbf{d}\}$
$\langle C \rangle$	$\{\mathbf{d}\}$

Al copiar los siguientes de $\langle C \rangle$ en los de $\langle B \rangle$ no hay cambios. Haciendo ahora un último recorrido, vemos que no hay más cambios y terminamos.

Ejercicio 22

Calcula los primeros y siguientes de los no terminales de la siguiente gramática:

$$
\begin{aligned}
\langle S \rangle &\rightarrow \langle Id \rangle\ \mathbf{:=}\ \langle E \rangle\ \mathbf{;} \\
\langle S \rangle &\rightarrow \mathbf{while}\ \langle E \rangle\ \mathbf{do}\ \langle LS \rangle\ \mathbf{endwhile} \\
\langle S \rangle &\rightarrow \langle Cond \rangle\ \mathbf{fi} \\
\langle S \rangle &\rightarrow \langle Cond \rangle\ \mathbf{else}\ \langle LS \rangle \mathbf{fi} \\
\langle LS \rangle &\rightarrow \langle S \rangle \langle LS \rangle \mid \lambda \\
\langle Cond \rangle &\rightarrow \mathbf{if}\ \langle E \rangle\ \mathbf{then}\ \langle LS \rangle \\
\langle E \rangle &\rightarrow \mathbf{cte} \mid \langle Id \rangle \\
\langle Id \rangle &\rightarrow \mathbf{*} \langle Id \rangle \mid \mathbf{id} \langle A \rangle \\
\langle A \rangle &\rightarrow \mathbf{[} \langle E \rangle \mathbf{]} \mid \lambda
\end{aligned}
$$

EJERCICIO* 23

Si la gramática tiene ciclos, podemos calcular primeros y anulables mediante ecuaciones similares a las del cálculo de siguientes. Escribe esas ecuaciones y pruébalas con la siguiente gramática:

$$\begin{array}{lcl} \langle \mathsf{S} \rangle & \rightarrow & \langle \mathsf{E} \rangle \,;\, \langle \mathsf{S} \rangle \mid \lambda \\ \langle \mathsf{E} \rangle & \rightarrow & \langle \mathsf{E} \rangle + \langle \mathsf{T} \rangle \mid \langle \mathsf{T} \rangle \\ \langle \mathsf{T} \rangle & \rightarrow & \langle \mathsf{T} \rangle \langle \mathsf{M} \rangle \langle \mathsf{F} \rangle \mid \langle \mathsf{F} \rangle \\ \langle \mathsf{M} \rangle & \rightarrow & * \mid \lambda \\ \langle \mathsf{F} \rangle & \rightarrow & \mathbf{id} \mid \text{“(”} \langle \mathsf{E} \rangle \text{“)”} \end{array}$$

5.5. Gramáticas LL(1)

Ahora estamos en condiciones de definir las gramáticas LL(1). Para ello, utilizaremos una estructura auxiliar a la que llamaremos *tabla de análisis*. Esta tabla tiene una fila por cada no terminal y un columna por cada terminal, incluyendo el fin de la entrada. Se rellena de la siguiente manera:

- Averiguar qué no terminales son anulables.
- Calcular los primeros de las partes derechas de las reglas.
- Calcular los siguientes de los no terminales anulables.
- Por cada regla $\langle \mathsf{A} \rangle \rightarrow \alpha$, poner α en las celdas $M[\langle \mathsf{A} \rangle, \mathtt{a}]$ tales que:
  - $\mathtt{a} \in$ primeros(α)
  - o anulable(α) y $\mathtt{a} \in$ siguientes($\langle \mathsf{A} \rangle$).

Intuitivamente, la tabla nos dice para cada no terminal y para cada símbolo de entrada qué parte derecha debe elegir el analizador. En nuestro ejemplo, la tabla es:

	cte	**do**	**id**	**if**	**then**	**while**	*	:=	[	]	**$**
⟨S⟩		⟨I⟩ := ⟨E⟩		**if** ⟨E⟩ **then** ⟨S⟩		**while** ⟨E⟩ **do** ⟨S⟩	⟨I⟩ := ⟨E⟩				
⟨E⟩	**cte**	⟨I⟩					⟨I⟩				
⟨I⟩		**id** ⟨A⟩					* ⟨I⟩				
⟨A⟩	λ				λ			λ	[⟨E⟩]	λ	λ

Es fácil ver la correspondencia entre la tabla y el código que hemos escrito. Cada una de las filas es un método `analiza`. Ante un terminal, si la columna correspondiente está vacía, se produce un error, en caso contrario, se intenta analizar la parte derecha correspondiente.

Cuando la tabla tiene en cada celda como máximo una parte derecha, decimos que la gramática es LL(1). Entonces, podemos estar seguros de que el programa que generamos para el análisis se comportará correctamente.

Algunas propiedades de las gramáticas LL(1):

- No son ambiguas.
- No tienen recursividad por la izquierda.
- Los conjuntos de primeros de las partes derechas de un no terminal son disjuntos.
- Si $\langle \mathsf{A} \rangle \stackrel{*}{\Rightarrow} \lambda$, los siguientes de $\langle \mathsf{A} \rangle$ son disjuntos con los primeros de la parte derecha de cualquier $\langle \mathsf{A} \rangle$-producción. Además, $\langle \mathsf{A} \rangle$ tiene como mucho una parte derecha anulable.

Hasta ahora hemos hablado de gramáticas LL(1). Si un lenguaje tiene una gramática LL(1), decimos que ese lenguaje es LL(1). Existen lenguajes incontextuales que no son LL(1), es decir que no existe ninguna gramática LL(1) capaz de modelarlos. Un ejemplo sería:

$$\{\mathtt{a}^n\mathtt{0}\mathtt{b}^n | n \geq 1\} \cup \{\mathtt{a}^n\mathtt{1}\mathtt{b}^{2n} | n \geq 1\}.$$

Afortunadamente, gran parte de las construcciones habituales de los lenguajes de programación pueden modelarse mediante gramáticas LL(1).

© Universitat Jaume I 2010-2011

5.6. Conflictos LL(1)

Si al rellenar la tabla de análisis aparece una celda con más de una entrada, tenemos lo que se llama un *conflicto LL(1)*. En términos de nuestro analizador, si nos encontramos con dos entradas en la celda del no terminal ⟨A⟩ para el terminal **b**, no sabremos que poner el cuerpo del `if` correspondiente. En estos casos, hay dos alternativas: utilizar un método de análisis más potente, por ejemplo LL(2) o LR(1), o modificar la gramática para evitar estos conflictos. Para las construcciones habituales de los lenguajes de programación suelen bastar ligeras modificaciones. Veremos ahora algunos ejemplos.

Eliminación de la recursividad por la izquierda Si la gramática tiene recursividad por la izquierda es seguro que aparecerá un conflicto (¿por qué?). Hay dos tipos de recursividad por la izquierda:

- Directa: existe al menos una *regla* de la forma $\langle A\rangle \to \langle A\rangle\alpha$.
- Indirecta: existe una *derivación* de la forma $\langle A\rangle \overset{+}{\Rightarrow} \langle A\rangle\alpha$.

En primer lugar, veremos cómo podemos eliminar la recursividad directa. Supongamos que tenemos el siguiente par de producciones:

$$\begin{aligned}\langle A\rangle &\to \langle A\rangle\alpha\\ \langle A\rangle &\to \beta\end{aligned}$$

Las formas sentenciales que pueden generar son del tipo $\beta\alpha^n$ con $n \geq 0$. Podemos sustituirlas por las siguientes reglas, que generan las mismas formas sentenciales:

$$\begin{aligned}\langle A\rangle &\to \beta\langle A'\rangle\\ \langle A'\rangle &\to \alpha\langle A'\rangle\\ \langle A'\rangle &\to \lambda\end{aligned}$$

En algunos casos, la eliminación no es tan sencilla y requiere expandir primero alguna de las reglas para después aplicar la transformación anterior. Existe un algoritmo para eliminar completamente la recursividad por la izquierda. Por desgracia, la gramática resultante no tiene ninguna relación con la original, lo que lo hace poco práctico. Afortunadamente, en la mayor parte de las ocasiones, la recursividad por la izquierda se puede eliminar utilizando gramáticas con partes derechas regulares.

Ejercicio 24

Elimina la recursividad por la izquierda en la siguiente gramática:

$$\begin{aligned}\langle E\rangle &\to \langle E\rangle\texttt{+}\langle T\rangle \mid \langle E\rangle\texttt{-}\langle T\rangle \mid \langle T\rangle\\ \langle T\rangle &\to \langle T\rangle\texttt{*}\langle F\rangle \mid \langle T\rangle\texttt{/}\langle F\rangle \mid \langle F\rangle\\ \langle F\rangle &\to \text{“(”}\langle E\rangle\text{“)”} \mid \texttt{id}\end{aligned}$$

Calcula su tabla de análisis y analiza la cadena `id*(id+id)`.

Ejercicio* 25

¿Cómo podemos eliminar la recursividad directa por la izquierda en las siguientes reglas?

$$\begin{aligned}\langle A\rangle &\to \langle A\rangle\alpha_1 \mid \langle A\rangle\alpha_2 \mid \ldots \mid \langle A\rangle\alpha_n\\ \langle A\rangle &\to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m\end{aligned}$$

Eliminación de prefijos comunes Otra de las causas de aparición de conflictos LL(1) es la existencia de prefijos comunes entre distintas partes derechas, lo que lleva a que los distintos conjuntos

de primeros no sean disjuntos. Para eliminar estos conflictos, podemos aplicar el análogo de "sacar factor común". Si tenemos las reglas:

$$\langle A \rangle \quad \rightarrow \quad \alpha\beta | \alpha\gamma$$

Podemos sustituirlas por:

$$\begin{array}{lcl} \langle A \rangle & \rightarrow & \alpha \langle A' \rangle \\ \langle A' \rangle & \rightarrow & \beta | \gamma \end{array}$$

Puede suceder que los prefijos aparezcan "indirectamente", entonces hay que expandir previamente algún no terminal. Por ejemplo:

$$\begin{array}{lcl} \langle S \rangle & \rightarrow & \textbf{id} := \langle E \rangle \,|\, \langle P \rangle \\ \langle P \rangle & \rightarrow & \textbf{id} \text{ “(”} \langle E \rangle \text{“)”} \end{array}$$

En este caso, deberemos expandir en primer lugar el no terminal $\langle P \rangle$:

$$\begin{array}{lcl} \langle S \rangle & \rightarrow & \textbf{id} := \langle E \rangle \,|\, \textbf{id} \text{ “(”} \langle E \rangle \text{“)”} \\ \langle P \rangle & \rightarrow & \textbf{id} \text{ “(”} \langle E \rangle \text{“)”} \end{array}$$

Después aplicamos la factorización:

$$\begin{array}{lcl} \langle S \rangle & \rightarrow & \textbf{id} \ \langle S' \rangle \\ \langle S' \rangle & \rightarrow & := \langle E \rangle \,|\, \text{“(”} \langle E \rangle \text{“)”} \\ \langle P \rangle & \rightarrow & \textbf{id} \text{ “(”} \langle E \rangle \text{“)”} \end{array}$$

Finalmente, si $\langle P \rangle$ no aparece en ninguna parte derecha, podemos eliminarlo.

Ejercicio 26

Elimina los prefijos comunes de estas producciones:

$$\begin{array}{lcl} \langle S \rangle & \rightarrow & \textbf{try} : \langle S \rangle \ \textbf{except id} : \langle S \rangle \\ \langle S \rangle & \rightarrow & \textbf{try} : \langle S \rangle \ \textbf{except id} : \langle S \rangle \ \textbf{finally} : \langle S \rangle \\ \langle S \rangle & \rightarrow & \textbf{id} := \langle E \rangle ; \,|\, \langle E \rangle ; \\ \langle S \rangle & \rightarrow & \{ \langle LS \rangle \} \\ \langle LS \rangle & \rightarrow & \langle S \rangle \,|\, \langle S \rangle \langle LS \rangle \,|\, \lambda \\ \langle E \rangle & \rightarrow & \textbf{id} \,|\, \textbf{cte} \end{array}$$

Manipulación directa de la tabla de análisis En algunos casos, la mejor manera de resolver un conflicto es cambiar directamente la tabla de análisis. Por ejemplo, sea la gramática:

$$\begin{array}{lcl} \langle S \rangle & \rightarrow & \textbf{if} \ \langle E \rangle \ \textbf{then} \ \langle S \rangle \ \textbf{else} \ \langle S \rangle \\ \langle S \rangle & \rightarrow & \textbf{if} \ \langle E \rangle \ \textbf{then} \ \langle S \rangle \\ \langle S \rangle & \rightarrow & \textbf{sent} \\ \langle E \rangle & \rightarrow & \textbf{id} \end{array}$$

En primer lugar, eliminamos prefijos comunes:

$$\begin{array}{lcl} \langle S \rangle & \rightarrow & \textbf{if} \ \langle E \rangle \ \textbf{then} \ \langle S \rangle \ \langle EI \rangle \\ \langle S \rangle & \rightarrow & \textbf{sent} \\ \langle E \rangle & \rightarrow & \textbf{id} \\ \langle EI \rangle & \rightarrow & \textbf{else} \ \langle S \rangle \,|\, \lambda \end{array}$$

© Universitat Jaume I 2010-2011

Ahora escribimos la tabla de análisis:

	else	id	if	sent	then	$
⟨S⟩			**if** ⟨E⟩ **then** ⟨S⟩ ⟨El⟩	**sent**		
⟨E⟩		**id**				
⟨El⟩	**else** ⟨S⟩ λ					λ

Vemos que hay un conflicto en la celda correspondiente a ⟨El⟩ con el terminal **else**. Si analizamos en qué situaciones podemos llegar al conflicto, vemos que la única manera en la que podemos tener que analizar un ⟨El⟩ es por haber terminado de analizar la ⟨S⟩ de un **if**-**then**. Pero si seguimos la regla de asociar el **else** al **if** más cercano, lo que tenemos que hacer en esta situación es analizar la secuencia **else** ⟨S⟩. Así pues, modificamos la entrada de la tabla y obtenemos:

	else	id	if	sent	then	$
⟨S⟩			**if** ⟨E⟩ **then** ⟨S⟩ ⟨El⟩	**sent**		
⟨E⟩		**id**				
⟨El⟩	**else** ⟨S⟩					λ

En términos de implementación, esto quiere decir que `analizaEl` tendrá la siguiente forma:

```
def analizaEl(self):
  if self.componente.cat== "else":
    # <El> -> else <S>
    self.avanza()
    self.analizaS()
  elif self.componente.cat== "eof":
    # <El> -> lambda
    pass
  else:
    self.error()
```

5.7. Gramáticas RLL(1)

Si la gramática de la que partimos es una GPDR, tenemos dos opciones para construir un analizador. La más obvia es transformarla, utilizando los métodos que presentamos en el punto 4.2, en una gramática incontextual para la que después construiríamos el analizador. Sin embargo, vamos a ver cómo podemos construir los analizadores directamente a partir de las propias GPDR. Esto nos llevará a los analizadores RLL(1), en cierto sentido los equivalentes a los LL(1) para GPDR. Estudiaremos la construcción de manera intuitiva, sin entrar en una formalización rigurosa.

En primer lugar, debemos darnos cuenta de que ahora al transcribir las reglas podrán aparecer no sólo terminales y no terminales, sino también expresiones regulares en forma de clausuras o disyunciones.

Comenzaremos por un ejemplo para entenderlo mejor. Sea la GPDR:

$$\begin{aligned} \langle \mathsf{E} \rangle &\rightarrow \langle \mathsf{T} \rangle (+\langle \mathsf{T} \rangle | -\langle \mathsf{T} \rangle))^* \\ \langle \mathsf{T} \rangle &\rightarrow \mathbf{id} | \mathbf{cte} | \text{“(”} \langle \mathsf{E} \rangle \text{“)”} \end{aligned}$$

Hemos modificado ligeramente la forma de la expresión para que el código tenga un poco más de “miga”.

Para construir el analizador de ⟨E⟩, comenzamos con el siguiente esqueleto:

```
def analizaE(self):
  if self.componente.cat in [ "apar", "cte", "id" ]:
    # <E> -> <T> (+<T>|-<T>)*
    self.analizaT()
    ... # ¿Clausura?
  else:
    self.error()
```

El problema es cómo analizamos la clausura. La solución es utilizar un bucle `while` en el que entraremos mientras el componente leído sea uno de los primeros del cuerpo de la clausura. En nuestro caso:

```
def analizaE(self):
  if self.componente.cat in [ "apar", "cte", "id" ]:
    # <E> -> <T> (+<T>|-<T>)*
    self.analizaT()
    while self.componente.cat in [ "suma", "resta" ]:
      ... # ¿Disyunción?
  else:
    self.error()
```

En general, una clausura la escribimos con un bucle `while` de la siguiente forma:

```
while self.componente.cat in primeros clausura:
  cuerpo clausura
```

Es interesante preguntarse si debemos comprobar al salir del bucle que el componente leído es efectivamente válido. Si queremos tener un informe de error detallado, lo mejor es comprobar después del bucle que efectivamente `self.componente.cat` es un siguiente de la clausura. Sin embargo, si la gramática es RLL(1), no es necesario en el sentido de que no se llamará a `self.avanza` antes de detectar el error.

Para la disyunción, tendremos que hacer algo parecido: decidimos con los primeros cuál de las ramas tomamos:

```
def analizaE(self):
  if self.componente.cat in [ "apar", "cte", "id" ]:
    # <E> -> <T> (+<T>|-<T>)*
    self.analizaT()
    while self.componente.cat in [ "suma", "resta" ]:
      # (+<T>|-<T>)
      if self.componente.cat== "suma":
        # +<T>
        self.avanza()
        self.analizaT()
      elif self.componente.cat== "resta":
        # -<T>
        self.avanza()
        self.analizaT()
      else:
        self.error()
  else:
    self.error()
```

© Universitat Jaume I 2010-2011

En general, cuando tengamos que transcribir una disyunción, tendremos una estructura similar a la que teníamos al escribir el método de análisis de un no terminal: una serie de `if` encadenados preguntado por los primeros de cada opción; en el cuerpo de cada condición está la transcripción de la opción correspondiente; el `else` corresponde a un error. Si alguna opción es anulable, tendremos que incluir también los siguientes de la disyunción en el correspondiente `if`. La similitud no es extraña, ya que podemos ver las distintas reglas de un no terminal como una disyunción.

Como puedes ver, al actuar mecánicamente se introducen algunas comprobaciones redundantes que se pueden eliminar sin problemas.

5.7.1. Cálculo de primeros

Ahora debemos adaptar el algoritmo presentado en el punto 5.3 para las GPDR. La adaptación es sencilla: basta con añadir los casos de la clausura y la disyunción.

¿Cuáles serán los primeros de una forma sentencial $(\rho)^*\beta$? Por un lado, serán los primeros de ρ; además, la clausura puede producir la cadena vacía, así que también tendremos los primeros de β. En cuanto a la disyunción, tendremos que los primeros son la unión de los primeros de cada una de las opciones. Si alguna de las opciones es anulable, habrá que mirar el resto de la forma sentencial.

Teniendo esto en cuenta, el algoritmo queda:

Algoritmo primeros $(\alpha : (N \cup \Sigma)^*)$
  $C = \emptyset$
  si $\alpha = \mathtt{a}\beta$ **entonces**
    $C := \{\mathtt{a}\}$;
  si no si $\alpha = \langle \mathsf{A} \rangle \beta$ **entonces**
    para todo $\langle \mathsf{A} \rangle \to \gamma \in P$ **hacer** $C := C \cup \text{primeros}(\gamma)$; **fin para**
    si anulable($\langle \mathsf{A} \rangle$) **entonces** $C := C \cup \text{primeros}(\beta)$; **fin si**
  si no si $\alpha = (\rho)^*\beta$ **entonces**
    $C := \text{primeros}(\rho) \cup \text{primeros}(\beta)$;
  si no si $\alpha = (\rho_1|\rho_2|\ldots|\rho_n)\beta$ **entonces**
    $C := \bigcup_{i=1}^{n} \text{primeros}(\rho_i)$;
    si anulable$(\rho_1|\rho_2|\ldots|\rho_n)$ **entonces** $C := C \cup \text{primeros}(\beta)$; **fin si**
  fin si
  devolver C;
fin primeros

5.7.2. Cálculo de anulables

Una modificación similar (figura 2) se puede hacer para el algoritmo que averigua si una forma sentencial es anulable. Basta con observar que la clausura siempre es anulable y que una disyunción lo es si lo es alguna de sus opciones.

5.7.3. Cálculo de siguientes

El cálculo de los siguientes se hace de manera similar al caso de las gramáticas incontextuales. Simplemente tendremos que aumentar las ecuaciones, teniendo en cuenta además que ahora tendremos que calcular los siguientes de clausuras y disyunciones.

En el resto de este punto, supondremos que ρ es un no terminal, una disyunción o una clausura. Las ecuaciones de la página 23 se pueden adaptar directamente:

- El fin de la entrada sigue al inicial:

$$\{\$\} \subseteq \text{siguientes}(\langle \mathsf{S} \rangle). \tag{4}$$

Algoritmo anulable ($\alpha : (N \cup \Sigma)^*$)
  si $\alpha = \lambda$ **entonces**
    devolver cierto;
  si no si $\alpha = \mathsf{a}\beta$ **entonces**
    devolver falso;
  si no si $\alpha = \langle \mathsf{A} \rangle \beta$ **entonces**
    si existe γ tal que $\langle \mathsf{A} \rangle \to \gamma \in P$ y anulable(γ) **entonces**
      devolver anulable(β);
    si no
      devolver falso;
    fin si
  si no si $\alpha = (\rho)^*\beta$ **entonces**
    devolver anulable(β);
  si no si $\alpha = (\rho_1|\rho_2|\ldots|\rho_n)\beta$ **entonces**
    si existe i tal que anulable(ρ_i) **entonces**
      devolver anulable(β);
    si no
      devolver falso;
    fin si
  fin si
fin anulable

Figura 2: Cálculo de anulables en GPDR.

- Si $\langle \mathsf{A} \rangle \to \alpha\rho\beta \in P$, entonces:

$$\text{primeros}(\beta) \subseteq \text{siguientes}(\rho); \tag{5}$$

- si, además, anulable(β) entonces:

$$\text{siguientes}(\langle \mathsf{A} \rangle) \subseteq \text{siguientes}(\rho). \tag{6}$$

Además debemos considerar los casos en que ρ esté en una clausura o una disyunción. Podemos entender mejor las ecuaciones que siguen si observamos que $(\alpha\rho\beta)^*$ tiene dos posibles reescrituras: la cadena vacía y $(\alpha\rho\beta)(\alpha\rho\beta)^*$. En este segundo caso, lo que aparece tras ρ es β, por lo que sus primeros son siguientes de ρ. Si β es anulable, detrás de ρ podrá aparecer cualquier cosa que pueda seguir a la clausura o, si consideramos la reescritura $(\alpha\rho\beta)(\alpha\rho\beta)(\alpha\rho\beta)^*$, los primeros de $(\alpha\rho\beta)$. En resumen:

- Si $(\alpha\rho\beta)^*$ aparece en la parte derecha de alguna regla, entonces:

$$\text{primeros}(\beta) \subseteq \text{siguientes}(\rho) \tag{7}$$

- y si anulable(β) entonces

$$\text{primeros}(\alpha\rho\beta) \cup \text{siguientes}((\alpha\rho\beta)^*) \subseteq \text{siguientes}(\rho). \tag{8}$$

Cuando ρ aparece en una disyunción, el cálculo es más sencillo:

- Si $(\alpha_1|\ldots|\beta\rho\gamma|\ldots|\alpha_n)$ aparece en la parte derecha de una regla, entonces:

$$\text{primeros}(\gamma) \subseteq \text{siguientes}(\rho) \tag{9}$$

- y si anulable(γ) entonces

$$\text{siguientes}((\alpha_1|\ldots|\beta\rho\gamma|\ldots|\alpha_n)) \subseteq \text{siguientes}(\rho). \tag{10}$$

Si estas reglas te resultan engorrosas, siempre puedes convertir la GPDR en una gramática incontextual y calcular sobre ella los siguientes.

© Universitat Jaume I 2010-2011

5.7.4. Tabla de análisis

La construcción de la tabla de análisis es análoga a la que hacíamos en el caso de las gramáticas LL(1). La única diferencia reseñable es que tendremos filas para las clausuras y disyunciones presentes en la gramática.

Los pasos que seguimos son:

- Averiguar qué no terminales y disyunciones son anulables.
- Calcular primeros de las partes derechas de las reglas y de los cuerpos de las clausuras y las disyunciones.
- Calcular siguientes de los no terminales anulables, de las clausuras y de las disyunciones anulables.
- Por cada regla $\langle \mathsf{A} \rangle \to \alpha$, poner α en las celdas $M[\langle \mathsf{A} \rangle, \mathtt{a}]$ tales que:
  - $\mathtt{a} \in$ primeros(α)
  - o anulable(α) y $\mathtt{a} \in$ siguientes($\langle \mathsf{A} \rangle$).
- Por cada clausura $(\rho)^*$:
  - Introducir $\rho(\rho)^*$ en las celdas $M[(\rho)^*, \mathtt{a}]$ tales que $\mathtt{a} \in$ primeros(ρ).
  - Si ρ es anulable, introducir $\rho(\rho)^*$ en las celdas $M[(\rho)^*, \mathtt{a}]$ tales que $\mathtt{a} \in$ siguientes($(\rho)^*$).
  - Introducir λ en las celdas $M[(\rho)^*, \mathtt{a}]$ tales que $\mathtt{a} \in$ siguientes($(\rho)^*$).
- Por cada disyunción $(\rho_1|\ldots|\rho_n)$:
  - Introducir ρ_i en las celdas $M[(\rho_1|\ldots|\rho_n), \mathtt{a}]$ tales que $\mathtt{a} \in$ primeros(ρ_i).
  - Introducir ρ_i en las celdas $M[(\rho_1|\ldots|\rho_n), \mathtt{a}]$ tales que $\mathtt{a} \in$ siguientes($\rho_1|\ldots|\rho_n$) y anulable(ρ_i).

Si no hay conflictos en la tabla, decimos que la GPDR es RLL(1).

Vamos a calcular la tabla de análisis de la GPDR del ejemplo:

$$\begin{array}{lcl} \langle \mathsf{E} \rangle & \to & \langle \mathsf{T} \rangle (+\langle \mathsf{T} \rangle | - \langle \mathsf{T} \rangle)^* \\ \langle \mathsf{T} \rangle & \to & \mathbf{id} | \mathbf{cte} | \text{“(”} \langle \mathsf{E} \rangle \text{“)”} \end{array}$$

El único elemento anulable es la clausura $(+\langle \mathsf{T} \rangle | - \langle \mathsf{T} \rangle)^*$. Necesitaremos saber los primeros de cada no terminal y de las clausuras y disyunciones:

Elemento	Primeros
$\langle \mathsf{E} \rangle$	{**id**, **cte**, “(”}
$\langle \mathsf{T} \rangle$	{**id**, **cte**, “(”}
$(+\langle \mathsf{T} \rangle \mid - \langle \mathsf{T} \rangle)$	{+, −}
$(+\langle \mathsf{T} \rangle \mid - \langle \mathsf{T} \rangle)^*$	{+, −}

Para calcular los siguientes, comenzamos con la tabla:

Elemento	Siguientes
$\langle \mathsf{E} \rangle$	{“)”, **$**}
$\langle \mathsf{T} \rangle$	{+, −}
$(+\langle \mathsf{T} \rangle \mid - \langle \mathsf{T} \rangle)$	{**id**, **cte**, “(”}
$(+\langle \mathsf{T} \rangle \mid - \langle \mathsf{T} \rangle)^*$	$\emptyset$

Extendemos los siguientes de ⟨E⟩:

Elemento	Siguientes
⟨E⟩	{“)”, **$**}
⟨T⟩	{+, −, “)”, **$**}
(+⟨T⟩|−⟨T⟩)	{**id**, **cte**, “(”}
(+⟨T⟩|−⟨T⟩)*	{“)”, **$**}

Y ahora ya podemos calcular la tabla (por problemas de espacio, la dibujamos en dos partes):

	id	**cte**	**+**
⟨E⟩	⟨T⟩(+⟨T⟩|−⟨T⟩)*	⟨T⟩(+⟨T⟩|−⟨T⟩)*	
⟨T⟩	**id**	**cte**	
(+⟨T⟩|−⟨T⟩)			+⟨T⟩
(+⟨T⟩|−⟨T⟩)*			(+⟨T⟩|−⟨T⟩)(+⟨T⟩|−⟨T⟩)*

	−	“(”	“)”	**$**
⟨E⟩		⟨T⟩(+⟨T⟩|−⟨T⟩)*		
⟨T⟩		“(” ⟨E⟩ “)”		
(+ ⟨T⟩|−⟨T⟩)	−⟨T⟩			
(+⟨T⟩|−⟨T⟩)*	(+⟨T⟩|−⟨T⟩)(+⟨T⟩|−⟨T⟩)*		λ	λ

5.8. Tratamiento de errores

El tratamiento de errores es una de las partes de la compilación que más depende del ingenio del programador. Siempre es posible encontrar mejoras a la estrategia de recuperación que se sigue y obtener mensajes de error más ajustados al verdadero problema. Lo que veremos son algunas ideas generales que después tendrás que ajustar a tus propias necesidades.

5.8.1. Conceptos generales

Esperamos del analizador que haga algo más que aceptar/rechazar un programa:

- debe informar de la presencia de errores con tanta claridad y precisión como sea posible,
- y debe tratar de recuperarse del error y seguir con el análisis para detectar otros (eventuales) errores, aprovechando así una única ejecución del compilador para detectar el mayor número posible de errores.

Pero, en cualquier caso, la detección de errores y su recuperación no deben afectar significativamente a la complejidad del analizador sintáctico.

5.8.2. ¿Dónde se detecta un error?

Los analizadores LL(1) tienen la propiedad del **prefijo viable**: los errores se detectan en el momento en que se ha leído el prefijo más corto de la entrada que no es válido para ninguna cadena del lenguaje. Por lo tanto, este es el primer punto donde se puede anunciar un error analizando la entrada símbolo a símbolo de izquierda a derecha.

El compilador debe señalar el lugar donde se detecta el error. Típicamente se indica la *línea* donde se ha detectado el error y un código o *mensaje informativo* del tipo de error.

5.8.3. ¿Qué acciones debemos emprender al detectar un error?

En principio, podemos pensar en abandonar la compilación (implícitamente, eso hemos hecho hasta ahora). Sin embargo esto no es adecuado más que en caso de entornos muy interactivos. En general, compilar es un proceso costoso y debemos aprovechar la compilación para detectar tantos errores como sea posible.

© Universitat Jaume I 2010-2011

Otra solución es intentar modificar el programa fuente con la esperanza de adivinar lo que quiso escribir el programador. Generaríamos entonces código de acuerdo con nuestra hipótesis. Esto es muy peligroso:

- En general, es muy difícil saber qué quería exactamente el programador.
- Puede pasar inadvertido que el código generado ha sido creado a partir de un fuente con errores, lo que puede traer malas consecuencias.
- En el mejor de los casos, será una pérdida de tiempo porque tras corregirse el error se volverá a compilar.

Así pues, la generación de código debe detenerse.

Podemos resumir lo anterior diciendo que debemos recuperarnos el error y seguir analizando, pero sin generar código.

Hay diversas estrategias de recuperación de errores:

- Entrar en **modo pánico**: se descartan todos los componentes léxicos de entrada desde la detección del error hasta encontrar un *componente léxico de sincronización.*
- Modelar los errores mediante **producciones de error**: se enriquece la gramática con producciones que modelan los errores más frecuentes y se asocia a éstas una acción de tratamiento de error.
- Tratar el error **a nivel de frase**: se efectúa una corrección local, sustituyendo un prefijo del resto del programa por una secuencia que permita seguir analizando.
- Efectuar una **corrección global**: se busca el programa correcto más próximo al erróneo.

Las dos últimas opciones se pueden considerar únicamente de interés teórico por su elevado coste computacional y el poco beneficio que dan en la práctica.

La segunda opción tiene más interés, aunque obtener unas buenas reglas es bastante costoso. Un ejemplo sería añadir a la gramática para las expresiones aritméticas una regla como $\langle \mathsf{F} \rangle \rightarrow$ "("$\langle \mathsf{E} \rangle$ y después emitir el mensaje de error correspondiente a la falta de paréntesis cuando se detectara este error. Como puedes imaginar, este tipo de reglas dan lugar a todo tipo de conflictos por lo que la escritura del compilador se complica enormemente. A cambio, los mensajes de error que se pueden emitir son muy precisos y la recuperación muy buena.

El método que utilizaremos será el tratamiento en modo pánico.

5.8.4. Tratamiento de errores en modo pánico

Como se ha comentado arriba, el tratamiento de errores en modo pánico consiste en detener el análisis sintáctico e ir leyendo componentes léxicos hasta encontrar algún componente que permita la "sincronización". Si lo consideramos desde la perspectiva del analizador, la sincronización puede conseguirse de tres maneras:

- Podemos proseguir el análisis como si hubiéramos encontrado la categoría léxica que esperábamos. Este es el tipo de tratamiento que haríamos por ejemplo al no encontrar un punto y coma donde debería estar. En este caso podemos sincronizarnos con los siguientes del elemento que esperábamos encontrar y con el propio elemento. Si encontramos el propio elemento al sincronizarnos, lo eliminaremos antes de continuar el análisis.
- Podemos proseguir el análisis como si hubiésemos encontrado la estructura (no terminal, clausura o disyunción) en la que hemos encontrado el error. Esta estrategia supone sincronizarse con alguno de los siguientes de la estructura y devolver (en caso del no terminal) el control como si todo hubiera ido bien. Podríamos aplicarla, por ejemplo, si estamos intentando encontrar una expresión en una llamada a función.
- Finalmente, podemos reintentar analizar la estructura en caso de que encontremos alguno de los primeros de la estructura que estamos analizando.

En cualquier caso, el fin de la entrada debe de ser siempre una de las categorías con las que nos podamos sincronizar.

Para todo esto, es útil disponer de una función de sincronización a la que se le pasa un conjunto S de categorías con las que podemos sincronizarnos. La función sería:

```
def sincroniza(self, sinc):
  sinc|= set("eof") # Nos aseguramos de que esté eof
  while self.componente.cat not in sinc:
    self.avanza()
```

En nuestro ejemplo, una posible estrategia de tratamiento de errores sería:

- Si no encontramos el primero de una ⟨E⟩, hay tres posibilidades:
  - Hemos encontrado el fin de fichero. Emitimos error "Fin de fichero inesperado" y salimos.
  - Hemos encontrado un paréntesis cerrado. Emitimos error "Paréntesis cerrado no esperado", sincronizamos con **id**, **cte** o paréntesis abierto y continuamos el análisis.
  - Hemos encontrado un + o un −. Emitimos error "Falta operando", sincronizamos con **id**, **cte** o paréntesis abierto y continuamos el análisis.
- Si después de analizar $(+\langle T\rangle|-\langle T\rangle)^*$, no hay un paréntesis ni fin de fichero, emitimos error "Falta operador", y nos sincronizamos con **id**, **cte** o paréntesis abierto (que, además, es lo único que puede haber en la entrada).
- Los casos para ⟨T⟩ son análogos a los de ⟨E⟩, salvo que estemos esperando el paréntesis cerrado. Entonces, emitimos error "Falta paréntesis cerrado" y nos sincronizamos con él o con cualquiera de los siguientes de ⟨T⟩.

6. Resumen del tema

- El analizador sintáctico encuentra las estructuras de la entrada.
- Especificamos el analizador mediante gramáticas incontextuales.
- El árbol de análisis muestra cómo derivar la entrada.
- Debemos trabajar con gramáticas no ambiguas o dar reglas para resolver las ambigüedades.
- Extendemos las gramáticas para utilizar partes derechas regulares.
- Dos tipos de análisis: ascendente y descendente.
- Empleamos análisis LL(1) (o RLL(1) para GPDR):
  - En cada momento, tomamos la decisión en base a un solo símbolo de la entrada.
  - No puede haber recursividad a izquierdas.
  - Las distintas alternativas tienen que diferenciarse por sus primeros (o siguientes si son anulables).
- Implementación: mediante analizadores descendentes recursivos.
- Tratamiento de errores:
  - Detiene la generación de código.
  - Recuperación en modo pánico.

© Universitat Jaume I 2010-2011