Capítulo 6

Programación de gramáticas clausales en Prolog

Fernando Soler Toscano[1]

En este trabajo presentamos una introducción, a modo de tutorial, a la escritura de *gramáticas de cláusulas definidas* (conocidas como DCG, por su nombre inglés, *Definite Clause Grammars*) en Prolog. No pretende ser un trabajo técnico ni sobre Programación Lógica ni sobre Lingüística, sino una pequeña ayuda para quien desee conocer de un modo práctico algunas de las técnicas elementales de la programación de gramáticas.

6.1. Introducción

Como introducción a las gramáticas formales, mostramos a continuación la jerarquía que proporciona Chomsky [2] de las diferentes clases de gramáticas generativas en virtud del tipo de reglas que las configuran. Una gramática $G = \langle V_n, V_t, R, O \rangle$ se compone de los siguientes cuatro elementos:

- Un conjunto de *símbolos no terminales*, representado por V_n, que está formado por las *categorías gramaticales*. En las gramáticas de ejemplo que veremos en las siguientes secciones, pertenecerán a este conjunto elementos tales como *sintagma nominal*, *artículo*, etc.

[1]Este trabajo se enmarca en el proyecto de investigación *Lógica y lenguaje: información y representación* (HUM2004-01255) del Ministerio de Educación y Ciencia.

- Un conjunto de *símbolos terminales*, reunidos en V_t, que más o menos viene a ser el *diccionario* del lenguaje para el que escribimos la gramática.

- Un conjunto de *reglas de transformación*, al que llamamos R, que tienen la forma $\alpha \rightarrow \beta$, donde α y β son cadenas (secuencias) de elementos de V_n y V_t. Cada una de tales reglas nos permite transformar la cadena α en β, como más adelante veremos.

- Un *símbolo raíz*, que representamos mediante O, que debe pertenecer a V_n (es decir, se trata de un símbolo no terminal). Representa la categoría superior de la gramática, y debe constituir la parte izquierda de al menos una de las reglas de R. Cuando la gramática G es la de una lengua natural[2], O se corresponde con la categoría *oración*.

Según las restricciones que se impongan respecto de la forma de cada elemento $\alpha \rightarrow \beta$[3] de R, son posibles cuatro tipos de gramáticas[4]:

- *Tipo 0*, formado por las *gramáticas irrestrictas*, o *recursivamente enumerables*, que no imponen restricciones a las estructuras de α y β.

- *Tipo 1*, de gramáticas *dependientes del contexto*. En este caso, $\alpha = \eta A \lambda$ y $\beta = \eta \gamma \lambda$, donde A es un símbolo no terminal, y η, λ y γ cadenas de símbolos de V_n y V_t. En el caso de η y λ, se permite que sean vacías; no así con γ.

- *Tipo 2*, de gramáticas *independientes* (o *libres*) *del contexto*, en las que α debe ser un símbolo de V_n y β cualquier cadena de *términos* (elementos tanto de V_n como de V_t).

- *Tipo 3*, de gramáticas *regulares* o *de estados finitos*. En este caso, α debe ser un símbolo de V_n y β puede ser o bien un símbolo de V_t o bien la concatenación de un símbolo de V_t y otro de V_n.

[2] Este paradigma es usado en muchos contextos. Por ejemplo, es frecuente la descripción de los lenguajes de programación según la forma conocida como de Backus-Naur (BNF, por *Backus-Naur Form*, en referencia a John Backus y Peter Naur), que no es más que una gramática con la misma estructura que hemos descrito.

[3] Como hemos explicado α y β representan cadenas (secuencias de uno o más elementos) de *términos*. Se llama *término* a todo elemento que pertenece a V_n o V_t.

[4] Mostramos sólo los rasgos más importantes de cada tipo, y omitimos algunos de los detalles más técnicos, como el tratamiento que cada uno de ellos hace de la cadena vacía ϵ.

La idea de *jerarquía de lenguajes* se debe a que las restricciones que cada uno de los cuatro tipos impone son progresivamente más estrictas, de modo que

$$\text{Tipo } 3 \subset \text{Tipo } 2 \subset \text{Tipo } 1 \subset \text{Tipo } 0$$

es decir, toda gramática de tipo $n+1$ (para n entre 0 y 2), es también una gramática de tipo n.

Veamos un ejemplo ilustrativo de una gramática $G = \langle V_n, V_t, R, O \rangle$ libre del contexto, compuesta por:

- V_n, que en este caso es el conjunto formado por o (que representa la categoría *oración*), sn (sintagma nominal), sv (sintagma verbal), det (determinante), n (nombre), vt (verbo transitivo) y vi (verbo intransitivo).

- V_t, que se compone de: *el*, *un*, *una*, *perro*, *hueso*, *ladra* y *muerde*.

- R, que contiene las siguientes reglas:

$$\begin{aligned}
o &\rightarrow sn,\ sv && (6.1)\\
sn &\rightarrow det,\ n && (6.2)\\
sv &\rightarrow vt,\ sn && (6.3)\\
sv &\rightarrow vi && (6.4)\\
det &\rightarrow el && (6.5)\\
det &\rightarrow un && (6.6)\\
det &\rightarrow una && (6.7)\\
n &\rightarrow perro && (6.8)\\
n &\rightarrow hueso && (6.9)\\
vi &\rightarrow ladra && (6.10)\\
vt &\rightarrow muerde && (6.11)
\end{aligned}$$

- O, que en este caso es o.

Veamos cómo nuestra gramática permite derivar la cadena “el perro muerde un hueso”. A continuación mostramos las transformaciones que deben aplicarse sucesivamente, comenzando por el símbolo no terminal o, para obtener la oración anterior:

o	$\rightarrow$	$sn,\ sv$	regla (6.1)
o	$\rightarrow$	$det,\ n,\ sv$	regla (6.2)
o	$\rightarrow$	$det,\ n,\ vt,\ sn$	regla (6.3)
o	$\rightarrow$	$det,\ n,\ vt,\ det,\ n$	regla (6.2)
o	$\rightarrow$	$el,\ n,\ vt,\ det,\ n$	regla (6.5)
o	$\rightarrow$	$el,\ n,\ vt,\ un,\ n$	regla (6.6)
o	$\rightarrow$	$el,\ perro,\ vt,\ un,\ n$	regla (6.8)
o	$\rightarrow$	$el,\ perro,\ vt,\ un,\ hueso$	regla (6.9)
o	$\rightarrow$	$el,\ perro,\ muerde,\ un,\ hueso$	regla (6.11)

En cada transformación se indica, a la derecha, la regla que se ha usado sobre el paso anterior. Podríamos haber empleado las reglas en otro orden para llegar a "$o \rightarrow el,\ perro,\ muerde,\ un,\ hueso$", oración que queríamos generar. Con esta gramática podemos producir otras oraciones como "un perro ladra". Igualmente, nos resulta imposible generar "el perro ladra un hueso". Sin embargo, permite oraciones agramaticales, como "una perro muerde una hueso". Más adelante veremos cómo podemos añadir a las gramáticas restricciones de concordancia.

6.2. Nuestra primera gramática en Prolog

Prolog, el lenguaje más popular de Programación Lógica, puede usarse para escribir *parsers*, analizadores gramaticales que se componen de:

- Una *gramática*. Construiremos las gramáticas simplemente a partir de su conjunto de reglas, porque ahí se encontrarán, implícitos, los conjuntos de símbolos terminales y no terminales.

- Un *mecanismo de búsqueda*, que sirve para recorrer las reglas de la gramática al generar o analizar producciones correctas. En nuestro caso, el mecanismo de búsqueda será el que proporciona Prolog, basado en las operaciones lógicas de unificación y resolución[5].

En este trabajo usaremos las gramáticas clausales para programar *parsers* gramaticales. Sin, embargo, las gramáticas DCG también sirven para implementar *parsers* morfológicos. Así, el sistema GRAMPAL [6], emplea las DCG de Prolog para escribir gramáticas de formación de palabras que sirven tanto para la generación como para el análisis morfológico.

[5]Para profundizar en los fundamentos lógicos de Prolog, recomendamos especialmente el manual de P. Flach [4].

Vamos a comenzar explicando los elementos mínimos que nos permitirán implementar gramáticas clausales en Prolog. Comencemos por el final. La gramática que mostramos en el ejemplo anterior puede programarse en Prolog escribiendo tan sólo lo siguiente, en un fichero de texto que podríamos guardar, por ejemplo, con el nombre `gramatica1.pl`:

```
o    -->  sn,  sv.
sn   -->  det, n.
sv   -->  vt,  sn.
sv   -->  vi.
det  -->  [el].
det  -->  [un].
det  -->  [una].
n    -->  [perro].
n    -->  [hueso].
vi   -->  [ladra].
vt   -->  [muerde].
```

Como puede observarse, la sintaxis es la misma que ya habíamos usado. Sólo debemos tener en cuenta la forma de la flecha de transformación "`-->`", que todas las reglas terminan en punto, que a la izquierda de cada regla siempre hay un solo símbolo no terminal, y que los símbolos terminales van encerrados entre los corchetes "`[`" y "`]`". Por lo demás, podemos dar a los símbolos los nombres que elijamos.

Ahora bien, ¿cómo hacemos para que nuestra gramática "funcione"? Necesitamos un *compilador* de Prolog, un sistema que permite a nuestro ordenador comprender los programas (las gramáticas que escribiremos en este trabajo son en realidad programas lógicos) escritos en Prolog. Recomendamos el uso de SWI-Prolog, pues es el que hemos usado para crear las gramáticas, y todo el código que aparece en este trabajo funciona en dicho compilador[6]. Puede descargarse gratis desde *http://www.swi-prolog.org/*, donde también podemos encontrar abundante documentación. Una vez instalado[7], podemos abrirlo[8]:

[6] A diferencia de lo que ocurre con otros lenguajes de programación, no existe un estándar rígido en Prolog. Aunque se suele tomar [3] como referencia, cada compilador lo desarrolla de forma diferente, lo que hace que los programas escritos para un sistema deban adaptarse en mayor o menor medida para que funcionen con otros compiladores.

[7] SWI-Prolog dispone de versiones para Linux, MS-Windows y MacOS X. En cualquier caso, la instalación es igual a la de cualquier programa para tales sistemas operativos.

[8] Generalmente basta con teclear *pl* en una consola, tal como hacemos en el ejemplo.

```
> pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.4.3)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free
software, and you are welcome to redistribute it under certain
conditions. Please visit http://www.swi-prolog.org for details

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

Entonces compilamos nuestra gramática. Sólo tenemos que escribir la orden "`[gramatica1].`" tras el signo "`?-`" con el que podemos dar instrucciones a Prolog. Veamos:

```
?- [gramatica1].
% gramatica1 compiled 0.01 sec, 3,060 bytes
Yes
```

De nuevo debemos observar que las instrucciones que damos terminan siempre en punto. Lo anterior sólo funciona si el fichero "`gramatica1.pl`" que contiene nuestra gramática se encuentra en la misma ruta desde la que ejecutamos SWI-Prolog en la consola.

Una forma más sencilla para escribir y compilar nuestras gramáticas es usar el editor que proporciona SWI-Prolog, que podemos abrir mediante "`?- emacs.`". Desde ahí disponemos de menús que permiten abrir, editar y compilar nuestros programas. Además, ofrece facilidades de edición como el coloreado de la sintaxis.

Tras compilar "`gramatica1.pl`" volvemos a encontrar, en la consola de SWI-Prolog, el signo "`?-`". Ya podemos experimentar con nuestra gramática:

```
?- phrase(o, [el, perro, ladra]).
Yes
?- phrase(o, [el, perro, Palabra, un, hueso]).
Palabra = muerde
Yes
?- phrase(o, [el, perro, ladra, un, hueso]).
No
?- phrase(o, [el, perro, muerde, una, hueso]).
Yes
```

Hemos usado "`phrase`", que es el predicado que emplea Prolog para realizar las transformaciones gramaticales. También ahora hemos terminado cada instrucción en un punto. Además, "`phrase`" toma dos argumentos, separados por comas y encerrados entre paréntesis. Las oraciones las representamos como listas de Prolog, con los elementos separados por comas y encerrados entre corchetes. Por último, sólo las *variables* (más adelante comentaremos más sobre ellas) pueden comenzar por una letra mayúscula. Para comprender el funcionamiento de "`phrase/2`" (lo escribimos así por ser un predicado de dos argumentos), comentemos los ejemplos anteriores. Mediante "`?- phrase(o, [el, perro, ladra])`" lo que hacemos es preguntar a Prolog si "el perro ladra" es una transformación correcta (gramatical) de la categoría "`o`" (por supuesto, siempre referido a la gramática que hemos compilado). Prolog nos responde que sí y vuelve a aceptar una nueva orden. Entonces (segunda llamada a `phrase/2`) le preguntamos si hay alguna oración gramatical tipo "el perro `Palabra` un hueso", para algún valor de `Palabra` (se trata de una variable) y que nos diga cual es dicho valor. Entonces nos responde que sí, la cadena es gramatical si `Palabra` vale "muerde"[9]. En los dos últimos ejemplos nos ha dicho que "el perro ladra un hueso" no es gramatical, pero que sí lo es "el perro muerde una hueso".

6.3. Concordancia de género y número

La gramática que hemos construido no contiene ninguna restricción de concordancia gramatical, por ello genera oraciones como "el perro muerde una hueso". Una solución posible a este problema es:

```
o           --> sn(_Gen2,Num),  sv(Num).
sn(Gen,Num) --> det(Gen,Num), n(Gen,Num).
sv(Num)     --> vt(Num),  sn(_Gen,_Num2).
sv(Num)     --> vi(Num).
det(f,sg)   --> [la]; [una].
det(f,pl)   --> [las]; [unas].
det(m,sg)   --> [el]; [un].
det(m,pl)   --> [los]; [unos].
```

[9]Cuando Prolog devuelve `Palabra = muerde` aún no escribe `Yes`, a la espera de que le pidamos o no más soluciones (más valores alternativos para `Palabra`). Lo que nosotros hemos hecho es pulsar la tecla *enter* en ese momento, por ello Prolog responde `Yes`. Pero podríamos haber escrito también "`;`" (seguido o no de *enter*, según la consola que usemos) para pedir nuevos valores de `Palabra`. En este último caso, la respuesta final habría sido `No`. Cuando hay valores alternativos, los devuelve tras cada pulsación del punto y coma.

```
vi(sg)      --> [ladra].
vi(pl)      --> [ladran].
vt(sg)      --> [muerde].
vt(pl)      --> [muerden].
n(f,sg)     --> [perra].
n(f,pl)     --> [perras].
n(m,sg)     --> [perro]; [hueso].
n(m,pl)     --> [perros]; [huesos].
```

Lo que hemos hecho es convertir las categorías gramaticales en predicados de Prolog con argumentos que indican el género y el número. Por ejemplo, la regla

```
sn(Gen,Num) --> det(Gen,Num), n(Gen,Num).
```

indica que un sintagma nominal tiene género "`Gen`" (que será "`m`" para masculino y "`f`" para femenino) y número "`Num`" ("`sg`" para singular y "`pl`" para plural) si está compuesto por "`det(Gen,Num)`" y "`n(Gen, Num)`", es decir, por un determinante y un nombre cuyos género y número no sólo deben coincidir entre sí, sino que serán el género y el número que se *transfieran* a la oración.

En algunas categorías, de las que sólo nos ha interesado marcar el número, hemos usado un solo argumento. Lo importante es que la gramática guarde coherencia interna. Es decir, el número de argumentos, así como su orden y contenido (género, número, etc.) debe ser fijo para cada categoría en toda la gramática.

Algo importante para que funcione la gramática es que debe hacerse un uso correcto de las variables de Prolog[10], que usaremos para expresar *rasgos* que pueden tomar distintos valores. Esta idea del uso de *rasgos* con contenido gramatical, y mecanismos de unificación (como hace Prolog) para imponer restricciones de concordancia, está presente en numerosas teorías lingüísticas [8]. Las gramáticas DCG de Prolog pueden emplearse para implementar muchas de tales teorías, como la Gramática Léxico-Funcional [11].

Hemos usado la *variable universal* de Prolog, que se representa mediante cualquier cadena de caracteres que comience por el guión bajo "`_`". Por ejemplo, en la regla

```
sv(Num)     --> vt(Num),  sn(_Gen,_Num2).
```

[10] Como hemos comentado, son variables de Prolog las cadenas de letras que empiezan por al menos una letra mayúscula.

la variable universal nos sirve para expresar que el sintagma verbal toma el número de su verbo, independientemente de los valores que tengan género y número de su sintagma nominal. En realidad, cuando usamos la variable universal no tiene importancia lo que escribamos tras el guión.

Por último, queremos comentar el uso que se ha hecho del punto y coma ";", que en Prolog tiene un valor disyuntivo. En realidad, una regla como

```
n(m,sg)     --> [perro]; [hueso].
```

equivale a

```
n(m,sg)     --> [perro].
n(m,sg)     --> [hueso].
```

con lo que podrían haberse incluido las dos últimas en vez de la primera y la gramática sería equivalente (en el sentido de que generaría y reconocería las mismas oraciones). Sin embargo, para que sean más cortos los ficheros hemos preferido emplear el punto y coma.

Ya sólo queda usar nuestra gramática[11]. Veamos:

```
?- phrase(o,[el, perro, muerde, un, hueso]).
Yes
?- phrase(o,[la, perro, muerde, un, hueso]).
No
?- phrase(o,[las, perras, muerde, un, hueso]).
No
?- phrase(o,[las, perras, muerden, un, hueso]).
Yes
?- phrase(o,[las, perras, muerden, un, huesos]).
No
?- phrase(o,[las, perras, muerden, unos, huesos]).
Yes
```

Ahora sí hemos conseguido implementar la concordancia de género y número. Pero las gramáticas no sólo sirven para reconocer oraciones correctas, también pueden generarlas:

```
?- phrase(o, Oración).
Oración = [la, perra, muerde, la, perra] ;
Oración = [la, perra, muerde, una, perra] ;
```

[11] Como antes, para ello debemos guardarla en un fichero con extensión *pl*, como `gramatica2.pl`, y compilarla de la forma indicada.

```
Oración = [la, perra, muerde, las, perras] ;
Oración = [la, perra, muerde, unas, perras]
Yes
```

Ahora lo que se ha hecho es pedir a Prolog que devuelva posibles valores de "`Oración`", y como lo pedimos mediante "`phrase(o, Oración)`" (sin olvidar el punto al final cuando lo escribamos en el compilador) exigimos que "`Oración`" sea una transformación correcta de la categoría "`o`" en nuestra gramática. Prolog comienza a devolver diferentes valores (como vemos hay cierto orden en las oraciones que devuelve, lo que se debe al modo en que recorre las reglas de la gramática) mientras nosotros introduzcamos "`;`". Si no hubiéramos parado, habría devuelto un total de 156 oraciones, todas gramaticales, pero alguna sin sentido, como "los huesos ladran". Se trata de la *cobertura* de nuestra gramática.

6.4. Construcción de árboles sintácticos

En ocasiones nos interesa no sólo que los sistemas determinen si cierta oración es o no gramatical, sino que además nos devuelvan un análisis sintáctico. Esto es lo que vamos a hacer en esta sección, añadir a nuestra gramática la posibilidad de construir árboles sintácticos de las oraciones que genera. Veamos cómo queda:

```
o(o(SN,SV))             --> sn(SN,_Gen,Num),  sv(SV,Num).
sn(sn(DET,N),Gen,Num) --> det(DET,Gen,Num), n(N,Gen,Num).
sv(sv(VT,SN),Num)       --> vt(VT,Num),  sn(SN,_Gen,_Num).
sv(sv(VI),Num)          --> vi(VI,Num).
det(det(X),f,sg)        --> [X], {member(X,[la,una])}.
det(det(X),f,pl)        --> [X], {member(X,[las,unas])}.
det(det(X),m,sg)        --> [X], {member(X,[el,un])}.
det(det(X),m,pl)        --> [X], {member(X,[los,unos])}.
vi(vi(ladra),sg)        --> [ladra].
vi(vi(ladran),pl)       --> [ladran].
vt(vt(muerde),sg)       --> [muerde].
vt(vt(muerden),pl)      --> [muerden].
n(n(perra),f,sg)        --> [perra].
n(n(perras),f,pl)       --> [perras].
n(n(X),m,sg)            --> [X], {member(X,[perro,hueso])}.
n(n(X),m,pl)            --> [X], {member(X,[perros,huesos])}.
arbol(Orac):- arbol(0,Orac).
```

```
arbol(Tab,R):-
   nl,tab(Tab),R =.. Li,
   (Li = [_,_] ->
     format('~w',[R]);
       (Li = [S|Otros],format('~w(',[S]),
       atom_length(S,Ls),Tab2 is Tab+Ls+1,
       findall(_,(member(X,Otros),arbol(Tab2,X)),_)),
       format(')')).
```

Hay varias cosas que debemos explicar. Lo más importante es que ahora los predicados que representan las categorías gramaticales toman un argumento más. Los argumentos segundo y tercero no tienen ninguna novedad, son los que usamos en la gramática de la sección anterior. La clave está en el primero. Para cada categoría, el primer argumento recoge su análisis. Por ejemplo, la regla

```
sn(sn(DET,N),Gen,Num) --> det(DET,Gen,Num), n(N,Gen,Num).
```

indica que el análisis de un sintagma nominal es `sn(DET,N)` si está formado por un determinante cuyo análisis es `DET` y por un nombre que tiene por análisis `N`. Los restantes argumentos de cada categoría implementan las restricciones de concordancia, y se comportan como ya hemos explicado. De esta forma, se van anidando los análisis de los diferentes componentes de la oración. El resultado es que, para una oración como "el perro muerde unos huesos", el análisis que se genera es:

```
o(sn(det(el),n(perro)),sv(vt(muerde),sn(det(unos),n(huesos))))
```

Puede ser difícil seguir la estructura del análisis anterior debido al gran anidamiento de paréntesis que contiene. Por ello, hemos incluido en la gramática el predicado `arbol/1`, que sirve para representar gráficamente estructuras como la anterior de una forma más sencilla de visualizar. Veamos un ejemplo de su uso:

```
?- phrase(o(A),[el,perro,muerde,unos,huesos]), arbol(A).
o(
  sn(
     det(el)
     n(perro))
  sv(
     vt(muerde)
     sn(
```

```
        det(unos)
        n(huesos))))
A = o(sn(det(el), n(perro)),
      sv(vt(muerde), sn(det(unos), n(huesos))))
Yes
```

En este ejemplo hemos pedido a Prolog que compruebe si "el perro muerde unos huesos" es o no una oración correcta, cuyo análisis sea `A` (puesto que hemos lanzado la pregunta usando `phrase/2`, y poniendo `o(A)` como la categoría de la que deseamos comprobar si la frase propuesta es una transformación correcta o no; como `A` es una variable estamos pidiendo los valores que debe tomar para que sea verdad nuestra pregunta). En este caso, no hemos puesto un punto al final de la pregunta, sino una coma y otro predicado, `arbol(A)`, seguido de un punto. Con esto lo que hacemos es que, una vez obtenido el valor de `A`, dibuje un árbol que represente su estructura.

No vamos a explicar cómo funciona el predicado `arbol/1`; para ver más detalles sobre esto, se puede acudir a la ayuda de SWI-Prolog (a la que es posible acceder mediante "`?- help.`"), donde se da cuenta de todos los predicados contenidos en su definición.

Terminamos esta sección comentando uno de los recursos más interesantes que se han empleado en esta gramática. Se trata de las llaves "`{`" y "`}`" que aparecen en algunas reglas. Aquí está la clave de la gran expresividad que pueden tener estas gramáticas, puesto que con estas llaves resulta posible añadir a las reglas condiciones adicionales, usando para ello todos los recursos del lenguaje Prolog. Como Prolog es un lenguaje de Programación universal, es decir, podemos implementar en él cualquier algoritmo, podremos hacer gramáticas tan expresivas como queramos (por supuesto, pagando el correspondiente precio en complejidad).

En la gramática anterior, hemos usado los corchetes sólo en reglas del tipo

```
det(det(X),f,sg) --> [X], {member(X,[la,una])}.
```

y lo que encierran en su interior es una llamada al predicado `member/2` de Prolog, en este caso para señalar que `X` debe ser un miembro de la lista formada por `la` y `una`. Así, sólo necesitamos una regla para los determinantes de género femenino y número singular. En la sección 6.6 se hará un mayor uso de este recurso.

6.5. Pequeño sistema de traducción

La traducción automática es probablemente la aplicación más importante de la Lingüística Computacional. En esta sección vamos a mostrar cómo podemos emplear las gramáticas para realizar traducciones. Nos centraremos en el conjunto de oraciones simples de tipo *Sujeto-Verbo-Objeto* usadas en el juego "piedra, papel, tijeras". Las cláusulas que definen nuestra gramática son:

```
% Inglés
sentence(s(S,V,O))  --> nom_p(S,N), verb(V,N), nom_p(O,_).
nom_p(np(M,S),N)    --> modifier(M), noun(S,N).
modifier(m(art))    --> [the].
noun(n(n_1),sg)     --> [stone].
noun(n(n_2),sg)     --> [paper].
noun(n(n_3),pl)     --> [scissors].
verb(v(v_1),sg)     --> [cuts].
verb(v(v_2),sg)     --> [wraps].
verb(v(v_3),sg)     --> [breaks].
verb(v(v_1),pl)     --> [cut].
verb(v(v_2),pl)     --> [wrap].
verb(v(v_3),pl)     --> [break].

% Español
oración(s(S,V,O))  --> sint_n(S,N), verbo(V,N), sint_n(O,_).
sint_n(np(M,S),N)  --> artículo(M,G,N), nombre(S,G,N).
artículo(m(art),f,sg)    --> [la].
artículo(m(art),m,sg)    --> [el].
artículo(m(art),f,pl)    --> [las].
nombre(n(n_1),f,sg)      --> [piedra].
nombre(n(n_2),m,sg)      --> [papel].
nombre(n(n_3),f,pl)      --> [tijeras].
verbo(v(v_1),sg)         --> [corta].
verbo(v(v_2),sg)         --> [envuelve].
verbo(v(v_3),sg)         --> [rompe].
verbo(v(v_1),pl)         --> [cortan].
verbo(v(v_2),pl)         --> [envuelven].
verbo(v(v_3),pl)         --> [rompen].
```

Como vemos, la gramática está dividida en dos secciones. Una, que sirve para generar oraciones en inglés, que desarrolla las transformaciones posibles

del símbolo no terminal *sentence*, y otra, para el español, que hace posibles las transformaciones del símbolo *oración*. Cada una de las dos partes por separado se parece a la gramática que construimos en la sección 6.4. Además, cada gramática tiene sus propios símbolos terminales y no terminales[12]. No podemos ni siquiera hacer correspondencias exactas entre símbolos, pues por ejemplo *nombre*, en la gramática del español, toma tres argumentos, mientras que *noun*, en la del inglés, tiene tan solo dos. Entonces, ¿qué nos permitirá hacer traducciones? Pues que las dos gramáticas comparten el análisis sintáctico, que ahora no es el propio ni del español ni del inglés, sino que se trata de un código artificial (algo así como una *interlingua*) que hemos creado a propósito. Veamos,

```
?- phrase(oración(A),[las, tijeras, cortan, el, papel]).
A = s(np(m(art), n(n_3)), v(v_1), np(m(art), n(n_2)))
Yes
?- phrase(sentence(B),[the, scissors, cut, the, paper]).
B = s(np(m(art), n(n_3)), v(v_1), np(m(art), n(n_2)))
Yes
```

Si pedimos el análisis de "las tijeras cortan el papel" como transformación de `oración(A)` y de "the scissors cut the paper" de `sentence(B)`, en ambos casos `A` y `B` contienen el mismo análisis (véase el ejemplo anterior). Se trata de un término formado por el símbolo `s` (que en español representará `oración` y en inglés `sentence`) que toma dos argumentos (primero y tercero) construidos mediante el símbolo `np` (que representa un sintagma nominal, lo que en la gramática del español se ha representado como `sint_n` y en la del inglés como `nom_p`) y otro mediante `v` (`verbo` o `verb`). Por lo demás, `m` representa un artículo y `n` un sustantivo. En cuanto al léxico, para este código común hemos usado unas etiquetas que después hacemos corresponden con palabras o conjuntos de palabras de cada lengua. Así, `v_1` representa en español tanto "corta" como "cortan" (que se use una forma u otra, en cada oración, vendrá impuesto por las restricciones propias de la gramática del español) y en inglés "cuts" o "cut". Es significativo lo que ocurre con `art`; mientras que en la gramática del inglés sólo se corresponde con "the", en español puede adquirir varias formas, según el género y número.

Podemos usar estas gramáticas de varias formas:

```
?- phrase(sentence(A),[the, stone, breaks, the, scissors]),
```

[12] Por ello hemos podido programar ambas gramáticas en el mismo fichero. En otro caso habría que haber usado módulos (más información sobre esto en la documentación de SWI-Prolog).

```
   phrase(oración(A),Español).
A = s(np(m(art), n(n_1)), v(v_3), np(m(art), n(n_3)))
Español = [la, piedra, rompe, las, tijeras]
Yes
?- phrase(oración(B),[el, papel, envuelve, la, piedra]),
   phrase(sentence(B),Inglés).
B = s(np(m(art), n(n_2)), v(v_2), np(m(art), n(n_1)))
Inglés = [the, paper, wraps, the, stone]
Yes
?- phrase(oración(C),Español), phrase(sentence(C),Inglés).
C = s(np(m(art), n(n_1)), v(v_1), np(m(art), n(n_1)))
Español = [la, piedra, corta, la, piedra]
Inglés = [the, stone, cuts, the, stone] ;

C = s(np(m(art), n(n_1)), v(v_1), np(m(art), n(n_2)))
Español = [la, piedra, corta, el, papel]
Inglés = [the, stone, cuts, the, paper]
Yes
```

En el primer caso, hemos pedido que analice "the stone breaks the scissors" como una transformación de *sentence* (por lo que empleará la gramática inglesa) y guarde el análisis en `A`, para a continuación pedir que genere, en español, una *oración* que tenga `A` por análisis. Usamos la variable `Español` para pedir dicha oración. Prolog realiza estas tareas, y devuelve para `Español` el valor "la piedra rompe las tijeras". El segundo ejemplo es similar, aunque esta vez traduce del español al inglés. En el último ejemplo, pedimos que genere una *oración* en español que tenga `C` como análisis y `Español` como transformación. Como ambos argumentos son variables, Prolog buscará la primera oración en español que encuentre. A continuación, pedimos que genere la traducción en inglés para dicha oración, puesto que mediante `phrase(sentence(C),Inglés)` pedimos que tome el análisis `C` y lo transforme en `Inglés`, una expansión de *sentence*. Cuando Prolog nos devuelve la primera solución (con los valores correspondientes para `C`, `Español` e `Inglés`) pedimos otra mediante ";".

6.6. Una gramática más compleja

A continuación presentamos una gramática más compleja que las anteriores, en que las reglas de producción recurren frecuentemente a predicados de Prolog. El propósito de esta gramática es generar, para cada hora expre-

sada numéricamente, como 4 : 45, una expresión en español que se refiera a la misma hora, como "las cinco menos cuarto". Igualmente, para cada expresión verbal de una hora, deberá generar la hora correspondiente en forma numérica. Así veremos que la posibilidad de analizar las oraciones generadas por gramáticas DCG sirve para mucho más que para obtener árboles sintácticos. En realidad, lo que hará esta gramática es relacionar pares $H : M$, en que H es una hora entre 1 y 12 y M un número de minutos entre 0 y 59, con secuencias de palabras que expresan cada hora. La cobertura será completa, es decir, para cada una de las 720 posibles horas (el producto de 12, número de horas, y 60, número de minutos) habrá una cadena de palabras correspondiente. Veamos el programa:

```
hora(H:M)      --> {between(1,12,H), between(0,59,M)},
                   artículo(H,M), qué_hora(H,M), minutos(M).
artículo(H,M) --> [la], {sg_pl(H,M,sg)}.
artículo(H,M) --> [las], {sg_pl(H,M,pl)}.
qué_hora(H,M) --> [Hora], {hora_base(H,M,HB),h(HB,_,_,Hora)}.
minutos(0)     --> [en, punto].
minutos(M)     --> [y], exp_mins(M), {between(1,30,M)}.
minutos(M)     --> [menos],exp_mins(M2),
                   {between(31,59,M),h(M2,M,_,_)}.
exp_mins(15)   --> [cuarto].
exp_mins(30)   --> [media].
exp_mins(M)    --> [Pal], num_mins(M),{h(M,_,Pal,_),
                   (between(1,14,M); between(16,29,M))}.
num_mins(1)    --> [minuto].
num_mins(M)    --> [minutos], {between(2,29,M)}.

sg_pl(H,M,sg) :- hora_base(H,M,1).
sg_pl(H,M,pl) :- hora_base(H,M,HB), between(2,12,HB).
hora_base(H,M,H)  :- between(0,30,M).
hora_base(12,M,1) :- between(31,59,M).
hora_base(H,M,HB) :- between(31,59,M),
   between(2,12,HB), succ(H,HB).
h(1, 59, un, una).
h(Num1, Num2, Pal, Pal) :-
   between(2,29,Num1),plus(Num1,Num2,60),
   nth1(Num1,[uno,dos,tres,cuatro,cinco,seis,siete,ocho,nueve,
   diez,once,doce,trece,catorce,quince,dieciséis,diecisiete,
   dieciocho,diecinueve,veinte,veintiun,veintidós,veintitres,
```

```
veinticuatro,veinticinco,veintiséis,veintisiete,veintiocho,
veintinueve],Pal).
```

El símbolo raíz de la gramática es `hora(H:M)`. En su argumento, `H:M`, hemos usado los dos puntos ":", que están predefinidos en Prolog como operador. En la regla de producción correspondiente, se realizan llamadas (dentro de las llaves, para imponer a las reglas restricciones basadas en predicados de Prolog) a `between(N1,N2,M)`, predicado que impone que el número `M` esté comprendido entre los dos números `N1` y `N2`. Con ello, restringimos las expresiones tipo `H:M` a valores de `H` entre 1 y 12 y de `M` entre 0 y 59. Entonces, `hora(H:M)` se reescribe como:

- `artículo(H,M)`, que será o bien "la" (como en "*la* una menos cuarto") o "las" ("*las* cuatro y media"), según sean `H` y `M`.

- `qué_hora(H,M)`, que será el nombre de la hora que se expresa, y depende no sólo de `H` sino también de `M`, pues por ejemplo para `3:50` diremos "las *cuatro* menos diez minutos".

- `minutos(M)`, que según sea `M` tomará valores tales como "y media", "en punto", "menos diez minutos", "y un minuto", etc.

Comencemos con `artículo(H,M)`. Las dos reglas de producción para este símbolo imponen que se reescriba como "la" o como "las" según sea "`sg`" o "`pl`" el valor que devuelva `sg_pl(H,M,X)` para `X`. En realidad, se trata de que "la" se reserve para "la una..." y en todos los demás casos se use "las". El predicado `sg_pl/3` se encarga de este cometido. No comentaremos los predicados auxiliares empleados en esta gramática. Están todos construidos a partir de un pequeño número de predicados predefinidos en SWI-Prolog (puede encontrarse documentación sobre estos predicados en la ayuda del compilador).

En cuanto a `qué_hora(H,M)`, se reescribe como la palabra correspondiente a la hora que va a aparecer en la expresión correspondiente a `H:M`. La regla de producción empleada recurre a los predicados `hora_base(H,M,HB)`, que dadas `H` y `M` devuelve el número `HB` de la hora que se escribirá (debe tenerse en cuenta que si `M` es mayor a 30 entonces `HB` es la hora siguiente a `H`, y en otro caso es `H`; además, la hora siguiente a `H` no siempre se obtiene sumándole una unidad, pues cuando `H` es 12 la siguiente hora es la 1), y el predicado `h(HB, _, _, Hora)` (algunos argumentos aparecen con la variable universal porque su contenido no nos interesa ahora) que dada la hora `HB` devuelve su nombre `Hora`, que será una palabra como "una", "dos", etc.

El símbolo `minutos(M)` se transformará o bien en "en punto", si `M` es 0, o en una expresión que comienza por "y" (si `M` está entre 1 y 30) o por "menos" (para valores de `M` mayores a 30 y que termina por el número de minutos, resultado de transformar `exp_mins(M)`, cuando `M` es menor o igual a 30, o bien `exp_mins(M2)`, donde `M2` es la diferencia entre 60 y `M` (para encontrar `M2` se emplea `h/4` también), cuando `M` es mayor a 30. Esto es así porque, por ejemplo, para 5 : 35 decimos "las seis menos veinticinco minutos", pues $60 - 35 = 25$.

La transformación de `exp_mins(M)` es "cuarto" (si `M` es 15), "media" (si `M` es 30) o bien la palabra que expresa el número de minutos `M`. Ahora, obtenemos esta palabra con el tercer argumento de `h/4`, en vez de con el cuarto, que usamos para las palabras correspondiente a las horas. Esto es así para respetar la concordancia de género entre de "horas" con las formas femeninas de los números ("la *una*") y de "minutos" con las masculinas ("y *un* minuto"). Por último, la transformación de `exp_mins(M)` termina con `num_mins(M)`.

Finalmente, `num_mins(M)` se transforma en "minuto" si `M` es 1 y en "minutos" en otro caso.

Terminamos con algunos ejemplos. En los dos primeros Prolog nos devuelve la frase correspondiente a una hora. En el último, nos dice la hora que corresponde a una frase:

```
?- phrase(hora(12:40),F).
F = [la, una, menos, veinte, minutos]
Yes
?- phrase(hora(15:25),F).
F = [las, tres, y, veinticinco, minutos]
Yes
?- phrase(hora(H), [las, dos, y, cuarto]).
H = 2:15
Yes
```

6.7. Comparación con las gramáticas independientes del contexto

Las gramáticas independientes del contexto han recibido siempre una atención especial, debido a la cantidad de fenómenos lingüísticos que permiten modelar, así como por su tratabilidad computacional[13], debido a su

[13] A medida que subimos en la jerarquía de Chomsky, y nos acercamos a las gramáticas de tipo 0, las gramáticas son más expresivas, es decir, permiten generar mayor cantidad de

moderada complejidad. Sin embargo, el propio Chomsky [2] observa que estas gramáticas no tienen suficiente expresividad para dar cuenta de ciertas estructuras propias de las lenguas naturales. Por ejemplo, no es posible crear una gramática independiente del contexto que genere todas las cadenas tipo $a^n b^n c^n$, es decir, cadenas compuestas por n repeticiones de a seguidas por otras tantas de b y finalmente de c, para cualquier valor arbitrario de n. Pues bien, esta limitación impide a las gramáticas independientes del contexto dar cuenta de oraciones que en ciertas lenguas naturales son gramaticales y que siguen esta misma estructura.

Sin embargo, es posible escribir en Prolog una gramática DCG que genere todas y sólo todas las cadenas tipo $a^n b^n c^n$. Veamos:

```
o         --> [a],   s(i).      % 1
s(I)      --> nb(I), nc(I).     % 2
s(I)      --> [a],   s(i(I)).   % 3
nb(i(I)) --> [b],   nb(I).     % 4
nb(i)     --> [b].              % 5
nc(i(I)) --> [c],   nc(I).     % 6
nc(i)     --> [c].              % 7
```

Si pedimos mediante "`phrase(o,C)`" que genere sucesivamente todas las transformaciones posibles para la categoría "`o`" ocurre lo siguiente:

```
?- phrase(o,C).
C = [a, b, c] ;
C = [a, a, b, b, c, c] ;
C = [a, a, a, b, b, b, c, c, c] ;
C = [a, a, a, a, b, b, b, b, c, c, c, c]
Yes
```

Como vemos, se generan sólo las cadenas que queríamos. Con esto observamos que las gramáticas DCG de Prolog son más expresivas que las gramáticas independientes del contexto.

¿Cómo consigue la gramática anterior generar todas las cadenas tipo $a^n b^n c^n$? La clave está en el símbolo no terminal "`s(I)`", que funciona como un contador. Al inicio, la única transformación posible es la que permite la

lenguajes. Pero a la vez, los *parsers* que pueden manejar tales gramáticas (para analizar o generar oraciones correctas) son más complejos, en el sentido de que necesitan más recursos (en un ordenador, memoria y tiempo) para funcionar. Por ello, uno de los temas de investigación más recurrentes en Lingüística Computacional es la búsqueda de las gramáticas menos complejas que puedan dar cuenta de los fenómenos lingüísticos propios de las lenguas naturales.

regla `1` (nos referimos a cada regla por el número que aparece tras "`%`"; este signo se usa para hacer *comentarios* en el programa, anotaciones que el compilador no interpreta), que transforma "`o`" en la secuencia que contiene "`a`" y "`s(i)`". Este es único símbolo al que es posible ahora aplicar una transformación; en este momento tiene el valor inicial del contador. Entonces, mientras se apliquen ahora sucesivas transformaciones según la regla `3`, se van añadiendo nuevas "`a`" a la cadena, pero a la vez el valor del contador se incrementa, pasando de `s(i)` a `s(s(i))`, a `s(s(s(i)))` y así sucesivamente. Finalmente, cuando se aplica al contador la regla `2`, se escriben tantas "`b`" y tantas "`c`" como sea su valor, mediante las reglas `4`–`7`.

El recurso que nos ha permitido superar la expresividad de las gramáticas independientes del contexto ha sido, como hemos comentado, el uso que hemos hecho de los mecanismos propios de Prolog. En resumen, los elementos de las DCG que extienden las gramáticas independientes del contexto, son:

- La posibilidad de usar *predicados* (en el sentido de Prolog) para los símbolos terminales y no terminales. Este recurso, que hemos usado en todas las gramáticas presentadas a partir de la sección 6.3., ha mostrado su utilidad para implementar desde la concordancia de género y número hasta la traducción entre lenguas.

- La posibilidad de añadir condiciones adicionales a las reglas. Esto lo hemos hecho en la gramática de la sección 6.4 (donde se explicó el uso de las llaves "`{`" y "`}`" a este propósito) y en la sección 6.6.

- La disponibilidad de los mecanismos de control propios de Prolog. De hecho, estas gramáticas son programas lógicos en Prolog, por lo que pueden usarse recursos como la disyunción "`;`" o el corte "`!`"[14].

Bibliografía

[1] Patrick Blackburn, Johan Bos, y Kristina Striegnitz. Learn Prolog now! University of the Saarland, 2001. Publicación electrónica en http://www.coli.uni-sb.de/~kris/learn-prolog-now/.

[14] No hemos comentado nada del corte "`!`" de Prolog, puesto que va más allá de los objetivos de este trabajo. Los lectores que deseen profundizar en Prolog pueden hacerlo a través de las referencias que proporcionamos en la bibliografía. Recomendamos especialmente el curso de P. Blackburn, J. Boss y K. Striegnitz [1], al que puede accederse libremente a través de Internet.

[2] Noam Chomsky. *Syntactic Structures*. Mouton, La Haya, 1957. Trad. esp.: *Estructuras sintácticas*, México, Siglo XXI, 1974.

[3] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog, The Standard: Reference Manual*. Springer, 1996.

[4] Peter Flach. *Simply Logical. Intelligent Reasoning by Example*. John Wiley, 1994.

[5] Joaquín Garrido Medina. *Lógica y Lingüística*. Síntesis, 1994.

[6] Antonio Moreno y José M. Goñi. GRAMPAL: a morphological processor for Spanish implemented in Prolog. En *Proceedings of the Joint Conference on Declarative Programming (GULP-PRODE'95)*, páginas 321–331, 1995.

[7] Antonio Moreno Sandoval. *Lingüística computacional*. Ed. Síntesis, 1998.

[8] Antonio Moreno Sandoval. *Gramáticas de unificación y rasgos*. A. Machado Libros, 2001.

[9] David Poole, Alan Mackworth, y Randy Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998.

[10] Stuart Russell y Peter Norvig. *Inteligencia Artificial: un enfoque moderno*. Prentice Hall, 1996.

[11] Hideki Yasukawa. LFG System in Prolog. En *COLING*, páginas 358–361, 1984.