Parsing para Dummies

Teoría de la computación

Agosto 2021

1 Introducción

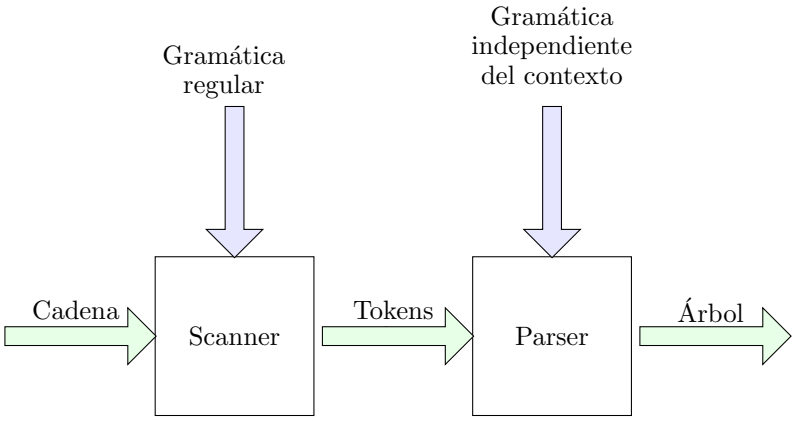
Sea $G = \langle V, \Sigma, R, S \rangle$ una gramática independiente del contexto. Suponga que $w \in L(G)$. El problema de parsing para w es encontrar un árbol de análisis de w en la gramática G. Existen dos enfoques que pueden usarse para este propósito. En el primero, llamado Top-down parsing, el objetivo es comenzar con el símbolo inicial de la gramática e ir reescribiéndolo mediante reglas que permitan generar a w. En el segundo, llamado Bottom-up parsing, el objetivo es recorrer los símbolos terminales que componen a w para ir encontrando las reglas que los fueron generando hasta llegar al símbolo inicial.

Antes de ver algunos algoritmos de parsing, resulta importante considerar el contexto general en el que ocurre este problema.

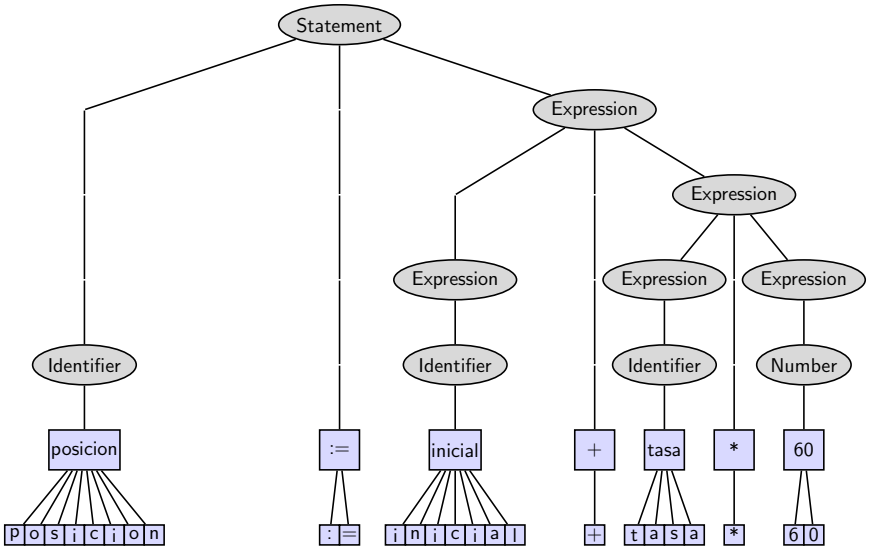
1.1 Scanners y parsers

El problema de parsing puede verse como una función cuyo input es un texto, digamos un pedazo de código en un lenguaje de programación, y cuyo output es un árbol de análisis de dicho texto. Esta función ocurre en dos etapas. En la primera, el texto se procesa para identificar palabras y sus categorías gramaticales. En este paso, se identifican subcadenas de símbolos que, agrupadas, constituyen un solo "token". Por ejemplo, la cadena `tasa*60` (escrita digamos en Python) consta de siete símbolos, pero la subcadena `tasa` es un token de un *identifier*, mientras que `60` es un token de un *number*. El algoritmo que realiza la identificación de tokens se conoce como un *scanner*. Su salida es una secuencia de tokens, que serán procesados por el parser. Es importante observar que son las expresiones regulares las más usadas para clasificar subcadenas dentro de su respectivo token.

En la segunda etapa, la secuencia de categorías gramaticales se procesa para obtener un árbol de análisis. Para encontrar el árbol de análisis se requiere de una gramática, usualmente independiente del contexto.



En el siguiente ejemplo observamos con distinto color el resultado de las dos etapas. La cadena `posicion:=inicial+tasa*60` ha sido procesada por el scanner (cuyo rango de acción ha sido representado en color azul claro) y la salida que arroja es a su vez procesada por el parser (cuyo rango de acción ha sido representado en color gris).



2 Métodos de parsing

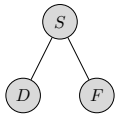
Consideremos ahora los dos enfoques de parsing y algunos de sus algoritmos.

2.1 Top-down

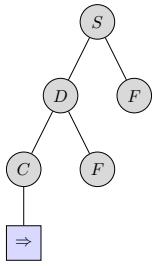
Considere la siguiente gramática:

$$\begin{aligned} P = \{ & S \to NF \mid DF \mid p \mid q \\ & F \to NF \mid DF \mid p \mid q \\ & D \to CF \\ & N \to \neg \\ & C \to \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \} \end{aligned}$$

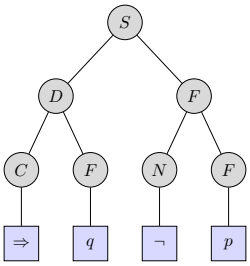
Suponga que queremos encontrar el árbol de análisis de la cadena $\Rightarrow q \neg p$. En un enfoque Top-down, sabemos que debemos partir del símbolo inicial S y encontrar una manera de generar a w. Las posibles reglas para reescribir a S nos permiten generar, por un lado, a p o a q y, por el otro, a NF y a DF. Intentamos con la regla que genera a DF y desde aquí seguimos intentando generar a w.



Ahora, el símbolo D lo podemos cambiar a DF por CFF (usando la única regla posible $D \to CF$). Además, el símbolo C se puede reescribir por $\Rightarrow$.

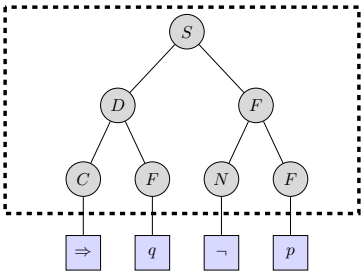


En este punto hemos generado la cadena $\Rightarrow FF$ y como el símbolo F puede reescribirse de varias maneras, tenemos que escoger con cuidado. Un poco de reflexión nos muestra que la primera F debemos reescribirla por q y la segunda por NF, de tal manera que la N se puede reescribir por $\neg$ y la nueva F que hemos generado por el símbolo terminal p; y así concluye la derivación.



2.1.1 Top-down por fuerza bruta

Dada una cadena $w \in \Sigma^*$, siempre es posible responder el problema de parsing para w. En un enfoque Top-down por fuerza bruta podemos ir considerando una a una las posibles combinaciones de aplicaciones de reglas hasta que aparezca w. Pero, ¿qué tantos recursos se consumen en este caso? Para responder más fácilmente esta pregunta, sólo consideraremos gramáticas que se encuentren en Forma Normal de Chomsky (FNC). Ilustraremos el conteo de las posibles combinaciones mediante la gramática G anterior, la cual ya está en FNC. Así pues, nos preguntamos cuántas posibles combinaciones deberíamos probar, en el peor de los casos, para generar nuestra cadena $\Rightarrow q \neg p$. Observe que el árbol de análisis de $p \neg q \Rightarrow$ permite ilustrar el hecho de que para obtener una cadena de longitud n se necesitan n aplicaciones de reglas de tipo $A \to a$. Esto se debe a que, en una gramática en FNC, cada símbolo terminal sólo puede provenir de una regla $A \to a$. Observe también que, al excluir el nivel de los símbolos terminales, tenemos un árbol cuya cantidad de hojas es igual a n (ver recuadro punteado):



Además, las reglas que se usaron para obtener este árbol son de la forma $A \to BC$, las cuales producen dos aristas cada una, así que estamos considerando un árbol binario. Para este tipo de árboles es fácil contar el número de aplicaciones de reglas, el cual es igual a la mitad del número de aristas, y como en todo árbol binario con n hojas el número de aristas es igual a $2(n-1)$, entonces el número de aplicaciones de reglas de la forma $A \to BC$ es $n-1$.

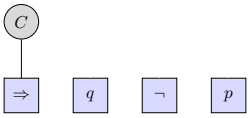
En total, para generar una cadena de longitud n se requieren $n + (n-1) = 2n-1$ aplicaciones de reglas. Así pues, podemos ir considerando una a una las posibles combinaciones de aplicaciones de reglas de longitud $2n-1$ hasta que

aparezca w, y si la gramática tiene m reglas, entonces en el peor de los casos tendremos que considerar todas las m^{2n-1} posibles combinaciones.

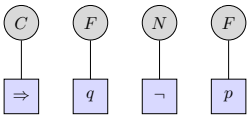
2.2 Bottom-up

Comenzamos por considerar la secuencia de símbolos terminales que componen a la cadena $\Rightarrow q\neg p$:

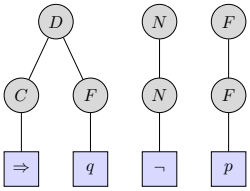
Tomamos el primer símbolo de izquierda a derecha y buscamos qué regla pudo haberlo generado, encontrando la regla $C \rightarrow \Rightarrow$.



Realizamos la misma operación en el segundo caracter, luego en el tercero y en el cuarto. Todos estos casos son muy sencillos, pues cada uno proviene directamente de un símbolo no terminal:

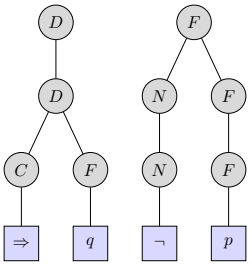


Ahora realizamos la misma operación sobre la nueva capa de símbolos que acabamos de construir, pero aquí nos topamos con una dificultad: en ninguna regla de la gramática encontramos al símbolo C solito y aislado en el lado derecho. Intentamos entonces buscar una regla cuyo lado derecho conste de este símbolo y el que le sigue, a saber, F. En otras palabras, buscamos en las reglas de la gramática cuál de ellas genera la cadena CF. Encontramos la regla $D \rightarrow CF$ y añadimos esto a nuestro arbol (observe que no se ha hecho nada sobre los dos símbolos de la derecha y sólo se han subido un nivel para que la nueva capa quede al mismo nivel):

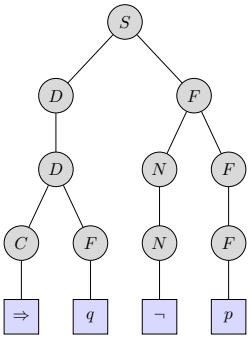


Continuamos el procedimiento sobre esta nueva capa de símbolos. Aquí observamos que ni D, ni DN, ni DNF son el lado derecho de ninguna regla. No obstante, NF sí lo es; ella es el lado derecho de las reglas $S \rightarrow NF$ y $F \rightarrow NF$. El procedimiento debe considerar ambas opciones, toda vez que no hay manera

de saber de antemano cuál de ellas funcionará. No es difícil ver que usar la regla $S \to NF$ conduce a un callejón sin salida, toda vez que la cadena DS no puede procesarse más (en efecto, ella no puede generarse mediante esta gramática). Debemos entonces usar la regla $F \to NF$, obteniendo una nueva capa:



En esta capa se nos presenta de nuevo una elección: la cadena DF puede generarse o bien mediante la regla $S \to DF$ o bien mediante la regla $F \to DF$. Se selecciona la primera, toda vez que con ella queda terminado el árbol de análisis (y la segunda lleva a un callejón sin salida).



3 Bottom-up mediante autómatas de Knuth

3.1 Manijas

Al proceso de recorrer la capa de símbolos y encontrar una regla que la pueda generar se le conoce como la búsqueda de *manijas*. Una manija para una cadena $w = xhy$ es una regla de la forma $A \to h$. Esta regla permite reducir la expresión xhy a la expresión xAy, lo cual suele denotarse mediante el símbolo $xhy \rightarrowtail xAy$. La reducción del ejemplo anterior puede denotarse, mediante esta terminología, de la siguiente manera:

$$\begin{aligned} \underline{\Rightarrow} q \neg p &\rightarrowtail C\underline{q}\neg p \rightarrowtail CF\underline{\neg} p \rightarrowtail CFN\underline{p} \rightarrowtail \underline{CF}NF \\ &\rightarrowtail D\underline{NF} \rightarrowtail \underline{DF} \rightarrowtail S \end{aligned}$$

Observe que en esta reducción se subrayó la subcadena h en cada caso. En los contextos en los cuales la regla específica no es importante, como en la reducción anterior, h también suele llamársе *manija*.

3.2 Autómatas K

¿Hay un algoritmo general para buscar una manija que nos permita reducir la cadena? Hay varios, y esto suele depender de la gramática, pero aquí presentaremos uno muy bonito basado en DFAs. Es importante notar desde el comienzo que este algoritmo es útil sólo para un tipo de gramáticas, las cuales se llaman Gramáticas Independientes del Contexto Deterministas. Estas gramáticas tienen la propiedad de que el algoritmo de búsqueda de una manija es limpio, en el sentido de que cada cadena tiene sólo una manija. De esta manera, el proceso de parsing es muy eficiente. La gramática del ejemplo anterior no es de este tipo, pues hay cadenas que tienen más de una manija. Simplificaremos la gramática para considerar sólo la capa de símbolos no terminales, eliminando también la engorrosa repetición de reglas en la primera y segunda filas, y así obtenemos una gramática determinista. Una prueba mecánica para determinar si una gramática dada es determinista consiste en construir su respectivo autómata *DK*, sobre lo cual volveremos más adelante.

La gramática simplificada se describe a continuación (observe que el símbolo inicial es F):

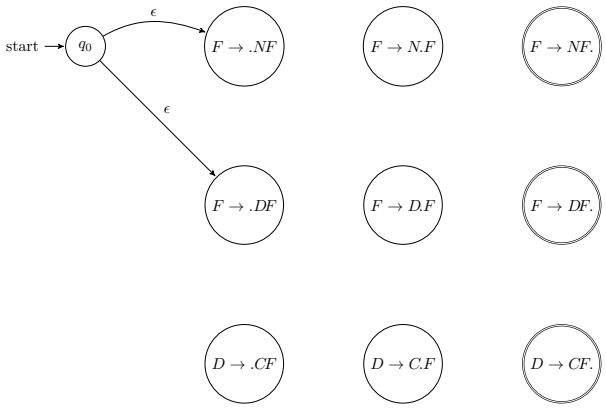
$$P = \{F \to NF \mid DF$$
$$D \to CF\}$$

La idea ahora es construir un NFA que acepte cualquier cadena que termine en una subcadena que sea el lado derecho de alguna de las reglas de la gramática. En este caso, queremos aceptar las cadenas que terminen en NF, DF y CF. Esta construcción no es difícil, toda vez que podemos dejar que el autómata adivine qué terminación buscar. Ahora bien, es importante que el autómata pueda llevar un registro de dónde va en esta comparación. Para ello, usamos las *reglas punteadas*, que no son otra cosa que tomar cada regla de la gramática y poner un punto en el lugar en el cual se está haciendo la comparación. Por ejemplo, la primera regla de nuestra gramática da lugar a las siguientes reglas punteadas:

$$F \to .NF$$
$$F \to N.F$$
$$F \to NF.$$

Los estados del autómata serán un estado inicial q_0 y todos los estados que corresponden a cada regla punteada, como $\boxed{F \to .NF}$. Los estados de aceptación serán las reglas punteadas en las cuales el punto está al final, como $\boxed{\boxed{F \to NF.}}$. Ahora, respecto a las transiciones, dejamos que el autómata adivine cuál regla va a comparar, lo cual se logra enviando una transición ϵ desde el estado inicial a todos los estados que representen una regla que arranca con el símbolo inicial, así:

el costo computacional de adivinar es probar todos los caminos posibles xd.



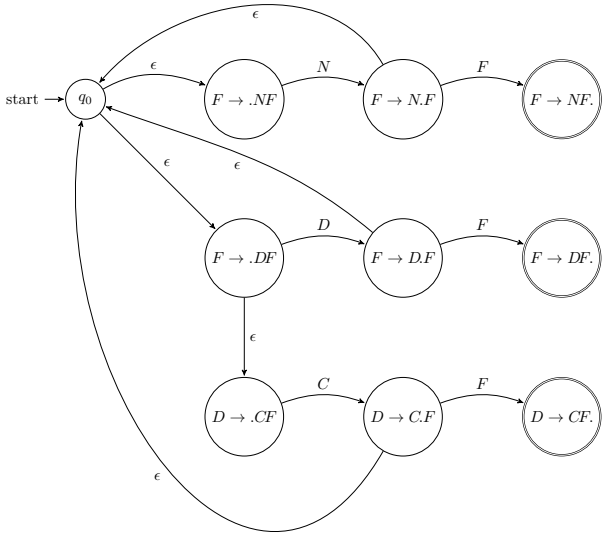
Ahora se consideran dos tipos de transiciones: las *shift-moves* y las ϵ-*moves*. Las primeras se definen entre las reglas punteadas correspondientes a la misma regla de la gramática:

$$\boxed{B \to u.av} \xrightarrow{a} \boxed{B \to ua.v}$$

Las *ϵ-moves* consideran una transición entre dos reglas punteadas precisamente en el punto en el cual el símbolo a comparar en la primera regla, digamos $B \to u.C.v$, sea el comienzo de la otra regla, digamos $C \to .r$. Observe que esta última debe tener el punto al inicio:

$$\boxed{B \to u.Cv} \xrightarrow{\epsilon} \boxed{C \to .r}$$

Observe que cuando C es el símbolo inicial de la gramática, resulta más económico hacer la transición al símbolo inicial del autómata. En nuestro ejemplo, el autómata K es el siguiente:



3.3 Autómatas DK

Recuerde que todo autómata finito no determinista puede transformarse en un autómata finito determinista. Este último es el que se conoce como el autómata DK. Realizar este proceso a mano alzada es descorazonador, toda vez que hay que considerar el conjunto potencia del conjunto de estados de K. Por ejemplo, nuestro anterior K tiene 10 estados, entonces debemos considerar 2^{10} estados para DK. Afortunadamente, existe una manera más directa de encontrar DK:

1. Cree un estado inicial q_0 con todas las reglas con punto al comienzo que arrancan con el símbolo inicial.

2. Si alguna regla en el estado en consideración, digamos q_i, tiene un punto antes de un símbolo no terminal, digamos C, entonces debemos añadir a q_i todas las reglas con punto al comienzo que arrancan con C. Continúe este proceso hasta que no pueda repetirse más.

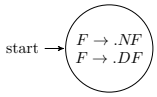
3. Por cada símbolo c que es precedido por un punto en alguna de las reglas de q_i, enviar una transición etiquetada con c al estado q_j, el cual contiene todas las reglas $B \to uc.a$ para las cuales $B \to u.ca$ está en q_i y todas las reglas obtenidas al aplicar el procedimiento 2 (hay que crear este estado q_j si no existiera).

4. Los estados finales son los que contienen por lo menos una regla punteada completa, de la forma $B \to x.$.

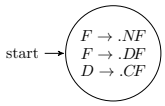
Ilustraremos el procedimiento con nuestra gramática P, que repetimos aquí por conveniencia:

$$P = \{F \to NF \mid DF$$
$$D \to CF\}$$

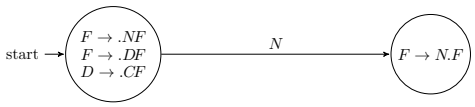
El estado inicial se inicializa incluyendo las dos reglas que arrancan con F (que es el símbolo inicial):



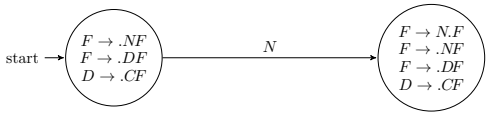
Este es el paso 1. No obstante, hay que aplicar el paso 2 sobre el estado inicial. Como no hay reglas que arranquen con N, no hacemos nada con la regla $F \to .NF$. Pero observe que debemos añadir la regla $D \to .CF$, toda vez que la regla $F \to .DF$ tiene un punto antes del símbolo no terminal D:



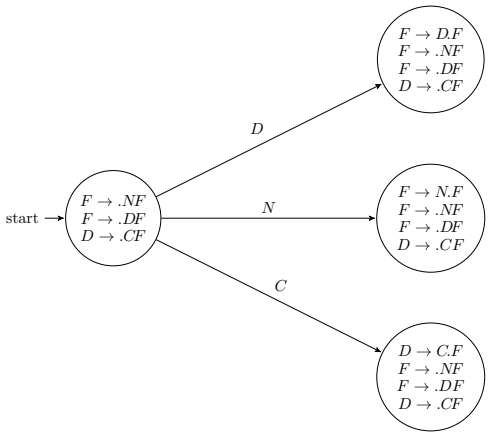
Ahora aplicamos el paso 3 sobre el estado inicial. Consideramos la regla $F \to .NF$, la cual nos pide que enviemos una transición etiquetada con el símbolo N a un estado, el cual debe tener, por lo menos, la regla $F \to N.F$. Para ello debemos crear un nuevo estado:



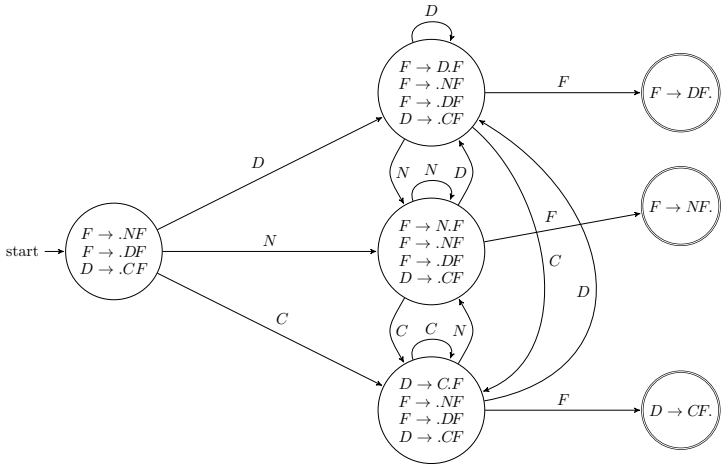
Pero este estado no está completo; aún debemos aplicar el procedimiento 2 sobre él, esto es, debemos incluir todas las reglas que arrancan con F y luego debemos incluir todas las reglas que arrancan con el símbolo precedido por un punto en las anteriores reglas:



Al repetir este proceso sobre todas las reglas del estado inicial, obtenemos lo siguiente:



Pero no hemos terminado, pues debemos aplicar el paso 3 sobre todos los estados, incluyendo los nuevos que se irán formando. Con un poco de paciencia se obtiene el siguiente autómata:



Ilustraremos el uso del autómata *DK* realizando el proceso de parsing para la cadena $\neg \Rightarrow p \wedge qp$. Observe que esta es la notación polaca de la fórmula de la lógica proposicional $\neg\big(p \Rightarrow (q \wedge p)\big)$. Antes que nada, puesto que hemos simplificado la gramática, debemos encontrar las categorías de los símbolos no terminales de la cadena. En otras palabras, realizaremos el parsing sobre la cadena *NCFCFF*. El procedimiento ahora es muy simple: introducimos la cadena en el autómata *DK*. El cómputo de la cadena se detendrá en un estado de aceptación, indicándonos la manija que debemos usar. Luego, reducimos la cadena con esta manija y repetimos el proceso hasta que en la reducción tengamos solamente el símbolo inicial.

Bien, al introducir la cadena *NCFCFF*, el autómata se detendrá en el estado

$\boxed{\boxed{D \to CF.}}$ habiendo procesado la subcadena $N\underline{CF}$, en la cual hemos subrayado la manija. Hacemos la reducción $N\underline{CF}CFF \rightarrowtail NDCFF$ e introducimos esta nueva cadena al autómata. Observe que vuelve a detenerse en el estado $\boxed{\boxed{D \to CF.}}$ habiendo procesado la subcadena $ND\underline{CF}$. Hacemos nuevamente la reducción usando esta manija: $ND\underline{CF}F \rightarrowtail NDDF$ e introducimos esta cadena al autómata. Ahora se detiene en el estado $\boxed{\boxed{F \to DF.}}$ habiendo procesado la subcadena $ND\underline{DF}$. Hacemos la reducción usando esta manija: $ND\underline{DF} \rightarrowtail NDF$. Se deja como ejercicio al lector completar el procedimiento.

Para terminar, observe que si el autómata DK tiene estados en los cuales hay más de una regla punteada completa del estilo $B \to x.$, entonces la gramática correspondiente no es determinista. Tampoco lo es si algún estado contiene una regla del estilo $B \to x.ay$, para a un símbolo terminal. El proceso de parsing para estas gramáticas se encuentra por fuera del alcance de este texto.