PRÁCTICA 1
(1) CMC

Ángel Igualada Moraga

Índice:

Ejercicio 1 Versión 1

```
func::listerr = "`1`  debe ser una lista de longitud 4  --->  G=(N,T,P,S) ";
getSimbolos::argerr = "La gramática proporcionada debe ser una lista: `1`";
getSimbolos[G1_List] := Module[{simbolos, listaP, produccion, antecedente,
    listaConsecuentes, noTerminales, Terminales, consecuente},
   simbolos = {};
   If[Length[G1] =!= 4,
    Return Message[func::listerr, G1]; simbolos = $Failed,
    simbolos = {};
    noTerminales = G1[[1]];
    Terminales = G1[[2]];
    listaP = G1[[3]];
    For[i = 1, i ≤ Length[listaP], i++,
     produccion = listaP[[i]];
     antecedente = produccion[[1]];
     listaConsecuentes = produccion[[2]];
     For[x = 1, x ≤ Length[listaConsecuentes], x++,
      If[ContainsNone[listaConsecuentes[[x]], noTerminales],
        AppendTo[simbolos, antecedente[[1]]]; Break[]];
     ];
    ];
   ];
   simbolos
  ];
getSimbolos[x1_] := (Message[getSimbolos::argerr, x1]; $Failed);
```

Ejercicio 1 Versión 2

```
getSimbolosV2[G1_List] := Module[{},
   Return[Flatten[Cases[G1[[3]], {z_, y_List} /; Cases[y, elem_ /; ContainsNone[elem, G1[[1]]]] =!= {} → z]]];
  ];
```

Ejercicio 2 Versión 1

```
func::listerr = "`1`  debe ser una lista de longitud 4  --->  G=(N,T,P,S) ";
getSimbolosNoGenerativos::argerr = "El argumento de getSimbolosNoGenerativos debe ser una lista: `1`";
getSimbolosNoGenerativos[G1_List] := Module[{simbolos, reglasP, regla, antecedente, listaConsecuentes, consecuente},
   If[Length[G1] =!= 4,
    Return Message[func::listerr, G1]; simbolos = $Failed,
    simbolos = {};
    reglasP = G1[[3]];
    For[i = 1, i ≤ Length[reglasP], i++,
     regla = reglasP[[i]];
     antecedente = regla[[1]];
     listaConsecuentes = regla[[2]];
     For[x = 1, x ≤ Length[listaConsecuentes], x++,
      If[! MemberQ[listaConsecuentes[[x]], antecedente[[1]]], Break[]];
     ];
     If[x == Length[listaConsecuentes] + 1, AppendTo[simbolos, antecedente[[1]]]];
    ];
   ];
   simbolos
  ];
getSimbolosNoGenerativos[x1_] := (Message[getSimbolosNoGenerativos::argerr, x1]; $Failed);
```

Ejercicio 2 Versión 2

```
getSimbolosNoGenerativosV2[G1_List] := Module[{},
   Return [Flatten[Cases[G1[[3]], {z_, y_List} /; Length[Cases[y, elem_ /; MemberQ[elem, z[[1]]]]] == Length[y] → z]]];
  ];
```

Ejercicio 3 Versión 1

```
func::listerr = "`1` debe ser una lista de longitud 4  --->  G=(N,T,P,S) ";
estaEnFNC::argerr = "Bad argument given to estaEnFNC: `1`";
estaEnFNC[lista_List] := Module[{consecuenteDeUnSignoTerminal, consecuenteDeDosSignosYSonNoTerminales,
    noTerminales, terminales, reglasP, res, regla, listaConsecuentes, consecuente},
   res = True;
   If[Length[lista] =!= 4,
    Return Message[func::listerr, lista]; res = $Failed,
    terminales = lista[[2]];
    reglasP = lista[[3]];
    For[i = 1, i ≤ Length[reglasP] && res, i++,
     regla = reglasP[[i]];
     listaConsecuentes = regla[[2]];
     For[j = 1, j ≤ Length[listaConsecuentes], j++,
      consecuente = listaConsecuentes[[j]];
      consecuenteDeUnSignoTerminal = (Length[consecuente] == 1 && MemberQ[terminales, consecuente[[1]]]);
      consecuenteDeDosSignosYSonNoTerminales = (Length[consecuente] == 2 && ContainsNone[consecuente, terminales]);
      If[ ! ( consecuenteDeDosSignosYSonNoTerminales || consecuenteDeUnSignoTerminal), res = False; Break[];];
     ];
    ];
   ];
   res
  ];
estaEnFNC[x3_] := (Message[estaEnFNC::argerr, x3]; $Failed);
```

Ejercicio 3 Versión 2

```
estaEnFNCV2[G1_List] := Module[{res},
   res = Flatten[Cases[G1[[3]], {z_, y_List} /; (Length[Cases[y, {e11_, e22_} /; MemberQ[G1[[1]], e11] && MemberQ[G1[[1]], e22]]] +
           Length[Cases[y, {eSolo_} /; MemberQ[G1[[2]], eSolo]]]) == Length[y] → z]];
   Length[res] == Length[G1[[3]]]
  ];
```

Ejercicio 4

```
makeProduction[ant_, con_List] := {{ant}, {con}};
getGramaticaLinealDerecha[G1_List] := Module[{letras, letrasDisponibles, reglasP, regla, antecedente,
    listaConsecuentes, antecedentes, consecuente, GramaticaResultado,
    nuevosSimbolos, produccionesAntecedenteActual, produccionesCreadas, consecuentesIntermedios,
    numGeneraciones, produccionesNuevosAntecedentes, tipo},
   letras = Map[Symbol, ToUpperCase[Complement[Alphabet[], {"i", "e"}]]];
   letrasDisponibles = Complement[letras, G1[[1]]];
   GramaticaResultado = {G1[[1]], G1[[2]], {}, G1[[4]]};
   numGeneraciones = 2;
   reglasP = G1[[3]];
   For[i = 1, i ≤ Length[reglasP], i++,
    regla = reglasP[[i]];
    antecedente = regla[[1]];
    listaConsecuentes = regla[[2]];
    produccionesAntecedenteActual = {};
    produccionesNuevosAntecedentes = {};
    For[j = 1, j ≤ Length[listaConsecuentes], j++,
     consecuente = listaConsecuentes[[j]];
     If[MemberQ[G1[[1]], Last[consecuente]], tipo = 2, tipo = 1];
     While[Length[letrasDisponibles] < Length[consecuente] - tipo,
      letrasDisponibles = Join[letrasDisponibles, Flatten[Thread[{letras^numGeneraciones}]]];
      numGeneraciones += 1;
     ];
     If[Length[consecuente] > tipo,
      (* Si el consecuente debe ser dividido *)
      {nuevosSimbolos, letrasDisponibles} = TakeDrop[letrasDisponibles, Length[consecuente] - tipo];
      GramaticaResultado[[1]] = Join[GramaticaResultado[[1]], nuevosSimbolos];
      antecedentes = Prepend[nuevosSimbolos, antecedente[[1]]];
      consecuentesIntermedios = Thread[{Take[consecuente, Length[consecuente] - tipo], nuevosSimbolos}];
      AppendTo[consecuentesIntermedios, Take[consecuente, -tipo]];
      produccionesCreadas = MapThread[makeProduction, {antecedentes, consecuentesIntermedios}];
      produccionesNuevosAntecedentes = Join[produccionesNuevosAntecedentes, Drop[produccionesCreadas, 1]];
      produccionesAntecedenteActual = Join[produccionesAntecedenteActual, produccionesCreadas[[1]][[2]]];
      ,
      (* Si el consecuente no debe ser dividido *)
      AppendTo[produccionesAntecedenteActual, consecuente];
     ];
    ];
    AppendTo[GramaticaResultado[[3]], {antecedente, produccionesAntecedenteActual}];
    GramaticaResultado[[3]] = Join[GramaticaResultado[[3]], produccionesNuevosAntecedentes];
   ];
   GramaticaResultado
  ];
```

Explicación código ejercicio 1 Versión 1

```
getSimbolos[x1_] := (Message[getSimbolos::argerr, x1]; $Failed);
func::listerr = "`1`  debe ser una lista de longitud 4  --->  G=(N,T,P,S) ";
getSimbolos::argerr = "La gramática proporcionada debe ser una lista: `1`";
   If[Length[G1] =!= 4,
    Return Message[func::listerr, G1]; simbolos = $Failed,
```

En primer lugar, cabe destacar que tenemos líneas cuya utilidad únicamente es informar de que o no se pasó una lista (sobrecarga getSimbolos) o que se pasó una lista de longitud menor a 4 (instrucción If).

```
            For[i = 1, i ≤ Length[listaP], i++,
             produccion = listaP[[i]];
             antecedente = produccion[[1]];
             listaConsecuentes = produccion[[2]];
             For[x = 1, x ≤ Length[listaConsecuentes], x++,
              If[ContainsNone[listaConsecuentes[[x]], noTerminales],
                AppendTo[simbolos, antecedente[[1]]]; Break[]];
             ];
            ];
           simbolos
```

La verdadera función de este módulo, se desarrolla mediante dos bucles, un primer bucle externo, recorre cada elemento del tercer componente de la gramática, es decir, recorre una por una todas las reglas de producción.

Dentro de este bucle, tenemos otro bucle, que dentro de cada regla de producción, recorre todos los consecuentes de esa regla.

```
              If[ContainsNone[listaConsecuentes[[x]], noTerminales],
                AppendTo[simbolos, antecedente[[1]]]; Break[]];
             ];
```

Con esta instrucción, comprobamos si el consecuente en el que nos encontramos no tiene ningún símbolo no terminal, si en efecto no contiene ningún símbolo no terminal, lo añadimos a la lista de símbolos buscados (símbolos) y como no necesitamos ver más consecuentes de esta regla, hacemos un Break[] para salir del bucle externo y pasar a comprobar otra regla de producción directamente.

Explicación código ejercicio 1 Versión 2

```
getSimbolosV2[G1_List] := Module[{},
   Return[Flatten[Cases[G1[[3]], {z_, y_List} /; Cases[y, elem_ /; ContainsNone[elem, G1[[1]]]] =!= {} → z]]];
  ];
```

Para explicar esta línea, la dividiremos en partes:

```
Cases[G1[[3]], {z_, y_List}
```

→ Devuelve una lista de las reglas de producción que cumplan el patrón {algo,lista} (podríamos haber forzado a lista lista), con esto recuperaríamos todas las reglas de producción.

```
Cases[G1[[3]], {z_, y_List} /; Cases[y, elem_ ]
```

→ Recuperamos todas las anteriores en las que los consecuentes son cualquier cosa (todas)

```
Cases[G1[[3]], {z_, y_List} /; Cases[y, elem_ /; ContainsNone[elem, G1[[1]]]] =!= {}
```

→ Recuperamos una lista con todas las reglas de producción en las que algún consecuente no contiene ningún no terminal (son todos terminales ó λ)

```
Cases[G1[[3]], {z_, y_List} /; Cases[y, elem_ /; ContainsNone[elem, G1[[1]]]] =!= {} → z
```

→En lugar de recuperar una lista de reglas enteras, recuperamos solo los antecedentes. Teniendo así, una lista de listas (de un único antecedente).

Por último, mediante la función Flatten, aplanamos la lista obtenida y la devolvemos.

Explicación código ejercicio 2 Versión 1

De nuevo, tenemos instrucciones meramente informativas.

```
For[i = 1, i ≤ Length[reglasP], i++,
 regla = reglasP[[i]];
 antecedente = regla[[1]];
 listaConsecuentes = regla[[2]];
 For[x = 1, x ≤ Length[listaConsecuentes], x++,
  If[! MemberQ[listaConsecuentes[[x]], antecedente[[1]]], Break[]];
 ];
 If[x == Length[listaConsecuentes] + 1, AppendTo[simbolos, antecedente[[1]]]];
];
```

Este módulo, recorre todas las reglas de producción mediante el bucle externo, mientras que el interno recorre todos los consecuentes de la regla en la que nos encontramos.

```
If[! MemberQ[listaConsecuentes[[x]], antecedente[[1]]], Break[]];
```

En esta ocasión, necesitamos que todos los consecuentes de una regla contengan al antecedente para aceptar el símbolo, por lo que dentro del bucle, comprobamos si el antecedente no lo contiene, hacemos un Break[ ] y pasamos a intentarlo con la siguiente regla, ya que esta no nos sirve.

```
If[x == Length[listaConsecuentes] + 1, AppendTo[simbolos, antecedente[[1]]]];
```

Al salir del bucle interior, ya sea por Break[] o por haber completado el bucle, comprobamos si el valor del contador del bucle es igual al número de consecuentes más uno, si es así, sabemos que ha realizado todas las iteraciones y que todos los consecuentes contienen al antecedente y podemos añadir el antecedente a la lista de símbolos a devolver (símbolos)

```
simbolos
```

Por último, devolvemos los símbolos devolviendo la variable simbolos.

Explicación código ejercicio 2 Versión 2

```
getSimbolosNoGenerativosV2[G1_List] := Module[{},
   Return [Flatten[Cases[G1[[3]], {z_, y_List} /; Length[Cases[y, elem_ /; MemberQ[elem, z[[1]]]]] == Length[y] → z]]];
  ];
```

De nuevo, dividiremos la línea en subinstrucciones:

```
Cases[G1[[3]], {z_, y_List} /; Length[Cases[y, elem_ /; MemberQ[elem, z[[1]]]]] == Length[y]
```

→Devolvemos una lista con las reglas de producción que cumplen el patrón {algo,lista} y tal que la longitud de la lista de los consecuentes de su lista de consecuentes en los que el antecedente está contenido es igual a la longitud a la lista de consecuentes, es decir, el antecedente está contenido en todos los consecuentes.

```
Cases[G1[[3]], {z_, y_List} /; Length[Cases[y, elem_ /; MemberQ[elem, z[[1]]]]] == Length[y] → z]
```

→Devolvemos una lista con el primer elemento de las reglas de producción (antecedente) en las que este, está contenido en todos los consecuentes, como el antecedente es una lista, tendremos una lista de listas.

```
Flatten[Cases[G1[[3]], {z_, y_List} /; Length[Cases[y, elem_ /; MemberQ[elem, z[[1]]]]] == Length[y] → z]];
```

→Aplanamos la lista con Flatten y la devolvemos

Explicación código ejercicio 3 Versión 1

```
For[i = 1, i ≤ Length[reglasP] && res, i++,
 regla = reglasP[[i]];
 listaConsecuentes = regla[[2]];
 For[j = 1, j ≤ Length[listaConsecuentes], j++,
```

De nuevo, tenemos un bucle for externo que recorre todas las reglas de producción, siempre y cuando la variable res sea True.

El segundo bucle for, recorre cada consecuente de las reglas en la que nos encontremos.

```
consecuenteDeUnSignoTerminal = (Length[consecuente] == 1 && MemberQ[terminales, consecuente[[1]]]);
```

En la variable consecuenteDeUnSignoTerminal almacenamos un valor booleano que indica si el consecuente solo está conformado por un símbolo y si este es terminal.

```
consecuenteDeDosSignosYSonNoTerminales = (Length[consecuente] == 2 && ContainsNone[consecuente, terminales]);
```

En la variable consecuenteDeDosSignosYSonNoTerminales almacenamos un valor booleano que indica si el consecuente está conformado exactamente por dos símbolos y sino contiene ningún símbolo terminal.

```
If[ ! ( consecuenteDeDosSignosYSonNoTerminales || consecuenteDeUnSignoTerminal), res = False; Break[];];
```

Si el consecuente no está en ninguna de las formas de la forma normal de Chomsky, automáticamente, la gramática no está en la forma normal de Chomsky por lo que ponemos res a False , realizamos un Break para salir del bucle interno, y dado que no se cumple la condición del bucle (res = False)

Salimos de este y devolvemos res.

Si todos los consecuentes cumplieran una de las dos posibles formas, nunca pondríamos rea a False y devolveríamos True.

Explicación código ejercicio 3 Versión 2

```
estaEnFNCV2[G1_List] := Module[{res},
   res = Flatten[Cases[G1[[3]], {z_, y_List} /; (Length[Cases[y, {e11_, e22_} /; MemberQ[G1[[1]], e11] && MemberQ[G1[[1]], e22]]] +
           Length[Cases[y, {eSolo_} /; MemberQ[G1[[2]], eSolo]]]) == Length[y] → z]];
   Length[res] == Length[G1[[3]]]
  ];
```

```
Cases[G1[[3]], {z_, y_List}]
```

Obtenemos una lista con todas las reglas

```
Cases[y, {e11_, e22_} /; MemberQ[G1[[1]], e11] && MemberQ[G1[[1]], e22]]
```

Devuelve los consecuentes que cumplen el patrón { e11_,e22_}, es decir los consecuentes que estás formados por dos símbolos y en los cuales el primer símbolo pertenece a los símbolos no terminales y el segundo símbolo pertenece también a los no terminales.

```
Cases[y, {eSolo_} /; MemberQ[G1[[2]], eSolo]]
```

Obtenemos los consecuentes que solo tienen un símbolo y este símbolo es terminal

```
Cases[G1[[3]], {z_, y_List} /; (Length[Cases[y, {e11_, e22_} /; MemberQ[G1[[1]], e11] && MemberQ[G1[[1]], e22]]]
                                                    +
                                Length[Cases[y, {eSolo_} /; MemberQ[G1[[2]], eSolo]]])
                                                    == Length[y] → z]];
```

Obtenemos una lista con las reglas en las que se cumple que el número de consecuentes de un símbolo terminal + el número de consecuentes de dos símbolos auxiliares es igual al número total de consecuentes y devolvemos únicamente el antecedente de estas reglas.

Tras aplanar la lista (que será una lista en la que cada elemento será una lista con un elemento) , devolvemos la comparación del número de reglas cuyos consecuentes cumplen las formas de la FNC y el número de reglas totales.

Explicación código ejercicio 4

```
letras = Map[Symbol, ToUpperCase[Complement[Alphabet[], {"i", "e"}]]];
```

Para comenzar, creamos un alfabeto base, a partir del cual generaremos más símbolos si fuera necesario, para ello, usamos Alphabet[] que nos devuelve el alfabeto inglés en minúsculas.

Como más tarde veremos, la generación de símbolos, se basa en la operación " ^ ", de modo que si necesitásemos nuevos símbolos, por ejemplo en el caso de B, aparecerían primero B , después B^2 , tras este B^3…

Esta idea, es válida siempre y cuando usemos variables, si usásemos no obtendríamos lo esperado, del alfabeto inglés eliminamos la "i" y "e" ya que Mathematica, por defecto, tiene asignado un valor numérico para I y E (lo que hace que en lugar de Tener E^2 tengamos un número.

Tras eliminar i y e del alfabeto, lo pasamos a mayúsculas y aplicamos Symbol[] a cada letra para transformar los strings en variables.

El objetivo de trabajar con variables, es doble, ya que la entrada de la gramática será con variables, así podremos saber si algún símbolo de nuestro conjunto se ha usado ya y además podremos aplicar ^ para generar nuevos símbolos.

```
letras = Map[Symbol, ToUpperCase[Complement[Alphabet[], {"i", "e"}]]];
letrasDisponibles = Complement[letras, G1[[1]]];
GramaticaResultado = {G1[[1]], G1[[2]], {}, G1[[4]]};
numGeneraciones = 2;
```

Tras tener nuestro conjunto de símbolos base p a partir del cual podremos generar más, en una variable letrasDisponibles, guardamos los símbolos que todavía no se han usado (si los hay).

Creamos la que será la gramática resultado y inicializamos una variable llamada numGeneraciones a dos, la función de esta variable es ser el exponente de los nuevos símbolos generados, (ya que ^ 0 siempre es 1 y ^ 1 es la variable, empezamos en 2 para distinguir los símbolos nuevos de los antiguos.

```
If[MemberQ[G1[[1]], Last[consecuente]], tipo = 2, tipo = 1];
```

Por la especificación del problema, los consecuentes pueden ser una cadena de símbolos terminales o una cadena de símbolos terminales y un símbolo auxiliar al final, comprobando si el ultimo símbolo del consecuente pertenece a los símbolos auxiliares, sabremos en que caso estamos.

Si el ultimo símbolo es auxiliar, estamos en el caso 2, en el que el consecuente final (tras el proceso de normalización) tendrá una longitud de dos.

Sino, estamos en el caso 1, en el que el consecuente final (tras el proceso de normalización) tendrá una longitud de uno.

```
While[Length[letrasDisponibles] < Length[consecuente] - tipo,
 letrasDisponibles = Join[letrasDisponibles, Flatten[Thread[{letras^numGeneraciones}]]];
 numGeneraciones += 1;
];
```

En cada pasada del bucle interno, tendremos este bucle While que generará nuevos símbolos si la longitud del consecuente – tipo (número de símbolos de la producción final , teniendo en cuenta que el antecedente actual será el primer símbolo en usarse para normalizar la regla de producción).

Para la generación, utilizamos el comando Thread que intercala cada elemento de la lista letras con numGeneraciones usando ^.Obtenemos lo siguiente:

$$\{\{A^2\},\{B^2\},\{C^2\},\{D^2\},\{F^2\},\{G^2\},\{H^2\},\{J^2\},\{K^2\},\{L^2\},\{M^2\},\{N^2\},\{O^2\},\{P^2\},\{Q^2\},\{R^2\},\{S^2\},\{T^2\},\{U^2\},\{V^2\},\{W^2\},\{X^2\},\{Y^2\},\{Z^2\}\}$$
$$\{\{A^3\},\{B^3\},\{C^3\},\{D^3\},\{F^3\},\{G^3\},\{H^3\},\{J^3\},\{K^3\},\{L^3\},\{M^3\},\{N^3\},\{O^3\},\{P^3\},\{Q^3\},\{R^3\},\{S^3\},\{T^3\},\{U^3\},\{V^3\},\{W^3\},\{X^3\},\{Y^3\},\{Z^3\}\}$$

Tras obtener esto, aplanamos la lista y juntamos los nuevos símbolos con los que teníamos disponibles. Cuando en caso de ser necesario, hayamos generado nuevos símbolos, pasaremos a analizar el consecuente.

```
If[Length[consecuente] > tipo, (* Si el consecuente debe ser dividido *), (* Si el consecuente no debe ser dividido *) ];
```

Si el consecuente es de tipo 2 (cadena de terminales y un auxiliar), lo trataremos si su longitud es mayor de 2 , esto se puede extrapolar al tipo 1, sino , no necesitamos hacer nada.

```
If[Length[consecuente] > tipo,
 (* Si el consecuente debe ser dividido *)
 {nuevosSimbolos, letrasDisponibles} = TakeDrop[letrasDisponibles, Length[consecuente] - tipo];
 GramaticaResultado[[1]] = Join[GramaticaResultado[[1]], nuevosSimbolos];
 antecedentes = Prepend[nuevosSimbolos, antecedente[[1]]];
```

Si debemos tratar el consecuente, necesitaremos usar tantos símbolos (para ser nuevos antecedentes) como la longitud del consecuente menos lo que mida su consecuente final.

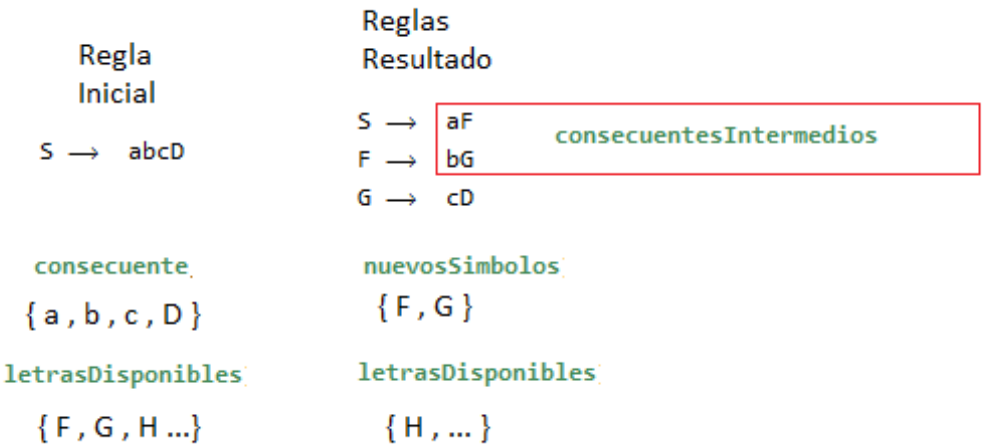
Tomamos los símbolos que necesitemos y los guardamos en nuevosSímbolos, y el resto, lo dejamos en letras disponibles.

Como vamos a usar estos nuevos símbolos, los añadimos a la lista de símbolos auxiliares de la que será la gramática resultado.

Serán antecedentes de las nuevas producciones, en primer lugar el antecedente actual, y en reglas sucesivas, los nuevosSímbolos

```
consecuentesIntermedios = Thread[{Take[consecuente, Length[consecuente] - tipo], nuevosSimbolos}];
```

Con esta instrucción, intercalaremos cada letra del consecuente a tratar (menos la parte final) con uno de los nuevos símbolos, con esto, lo que perseguimos , es crear los consecuentes de las nuevas reglas.




```
AppendTo[consecuentesIntermedios, Take[consecuente, -tipo]];
```

Al último consecuente no hay que añadirle nada asique lo añadimos a consecuentes intermedios.

```
produccionesCreadas = MapThread[makeProduction, {antecedentes, consecuentesIntermedios}];

                      makeProduction[ant_, con_List] := {{ant}, {con}};
```

Con MapThread, aplicamos la función makeProduction que nos creará una lista de reglas de producción que tendrá como antecedente un elemento de antecedentes y como consecuente un elemento de consecuentes intermedios.

```
produccionesNuevosAntecedentes = Join[produccionesNuevosAntecedentes, Drop[produccionesCreadas, 1]];
produccionesAntecedenteActual = Join[produccionesAntecedenteActual, produccionesCreadas[[1]][[2]]];
```

En producciones nuevos antecedentes, guardaremos las producciones de los símbolos que generamos y en produccionesAntecedenteActual, guardaremos el consecuente de la producción que hemos creado.

Del antecedente que producía el consecuente que hemos convertido en nuevas reglas, nos guardamos solo el consecuente porque puede que tengamos que analizar mas consecuentes con ese antecedente y crearemos una regla de producción con todos sus consecuentes.

```
AppendTo[GramaticaResultado[[3]], {antecedente, produccionesAntecedenteActual}];
GramaticaResultado[[3]] = Join[GramaticaResultado[[3]], produccionesNuevosAntecedentes];
```

Al finalizar todas las pasadas del bucle interno, ya hemos analizado todos los consecuentes de esta regla, y sabremos ya que consecuentes debe tener por lo que en este momento añadimos una regla con el antecedente y todos sus consecuentes a la gramática resultado.

Cabe destacar que el añadir las reglas creadas en cada pasada del bucle interno, es solo por mantener el orden al devolver la gramatica y podrían acumularse todas y hacer el Join a acabar el bucle externo.