PRÁCTICA 1
(2) CMC

Ángel Igualada Moraga

A continuación, tenemos un esquema del código dividido por partes, cada parte es explicada en las páginas posteriores coincidiendo el numero de partes con el número de páginas de explicación

PARTE 1

```
CYK::argerr = "La palabra contiene simbolos que no pertenecen a los terminales de la gramática -> `1` ";
CYK[G1_List, palabra_List] := Module[{regla, matriz, letra, caracteresQueProducenElPrefijo, caracteresQueProducenElSufijo, antecedentes},
  If[! ContainsAll[G1[[2]], palabra], Return Message[CYK::argerr, Complement[palabra, G1[[2]]]]; Return[False]];
  matriz = SparseArray[{fila_, col_} /; col + fila > Length[palabra] + 1 → Null, {Length[palabra], Length[palabra]}] (*//MatrixForm*);
  matriz = Normal[matriz];
```

PARTE 2

```
  (*PRIMERA PARTE i =1  *)
  For[i = 1, i <= Length[palabra], i++,
   letra = palabra[[i]];
   matriz[[1]][[i]] = Flatten[Cases[G1[[3]], {z_, y_List} /; Cases[y, {letra}] =!= {} → z]];
  ];
  (*FIN PRIMERA PARTE *)
```

PARTE 3

```
  (*INICIO SEGUNDA PARTE *)
  For[j = 2, j <= Length[palabra], j++,
   For[i = 1, i ≤ (Length[palabra] - j + 1), i++,
     matriz[[j]][[i]] = {};
     For[k = 1, k <= j - 1, k++,
      caracteresQueProducenElPrefijo = matriz[[k, i]];
      caracteresQueProducenElSufijo = matriz[[j - k]][[i + k]];
```

PARTE 4

```
      antecedentes = Flatten[Cases[G1[[3]], {z_, y_List} /; ! MemberQ[matriz[[j]][[i]], z[[1]]] && Cases[y, {y1_, y2_} /; MemberQ[caracteresQueProducenElPrefijo, y1] &&
                MemberQ[caracteresQueProducenElSufijo, y2]] =!= {} → z]];
      If[antecedentes =!= {}, matriz[[j]][[i]] = Join[matriz[[j]][[i]], antecedentes]];
     ];
    ];
  ];
  (*FIN SEGUNDA PARTE *)
  MemberQ[matriz[[Length[palabra], 1]], G1[[4]]]
]
```

```
CYK::argerr = "La palabra contiene simbolos que no pertenecen a los terminales de la gramática -> `1` ";
CYK[G1_List, palabra_List] := Module[{regla, matriz, letra, caracteresQueProducenElPrefijo, caracteresQueProducenElSufijo, antecedentes},
  If[! ContainsAll[G1[[2]], palabra], Return Message[CYK::argerr, Complement[palabra, G1[[2]]]]; Return[False]];
  matriz = SparseArray[{fila_, col_} /; col + fila > Length[palabra] + 1 → Null, {Length[palabra], Length[palabra]}] (*//MatrixForm*);
  matriz = Normal[matriz];
```

Comenzaremos el análisis del código con la declaración de la función y la creación de la estructura de datos que lo soportará.

En la **línea 1**, observamos un mensaje auxiliar de la función al que se le pasa un argumento, este mensaje se explicará más concisamente más adelante.

En la **línea 2** declaramos la función que implementa el algoritmo CYK (CYK) , esta función tomará dos argumentos (G1 y palabra) ambos de tipo lista y en ella declaramos las 6 variables a usar.

En la **línea 3** comprobamos que todos los signos de la palabra están entre los terminales de la gramática, si no lo están, retornamos un mensaje, concretamente el de la **línea 1** y le pasamos como argumento que símbolos incumplen esta condición, tras esto retornamos False (no puede pertenecer al lenguaje) y acabamos la ejecución. A continuación, tenemos un ejemplo:

```
In[13]:= CYK[GramaticaFNC1, {b, a, d, b, a}]
  ··· CYK: La palabra contiene simbolos que no pertenecen a los terminales de la gramática -> {d}
Out[13]= False
```

En la **línea 4**, se crea mediante un SparseArray, una estructura que sigue el patrón dado.

En primer lugar, definimos dos variables para definir con ellas el patrón `{fila_, col_}`

Tras esto, utilizamos el operador Condición (/;) para establecer que si nuestro patrón se cumple, esa posición deberá tener el valor Null `/; col + fila > Length[palabra] + 1 → Null,` Con esto, conseguimos poner a Null las posiciones imposibles del algoritmo.

Por último, fijamos el tamaño de filas y columnas con la longitud de la palabra `{Length[palabra], Length[palabra]}`

En la **línea 5** convertimos el SparseArray en un Array normal para trabajar más cómodamente.

Así pues para una palabra de 5 caracteres tendríamos →

```
Print[matriz // MatrixForm];
escribe                    forma de matriz
⎛0  0    0    0    0⎞
⎜0  0    0    0  Null⎟
⎜0  0    0  Null Null⎟
⎜0  0  Null Null Null⎟
⎝0 Null Null Null Null⎠
```

```
(*PRIMERA PARTE i =1  *)
For[i = 1, i <= Length[palabra], i++,
 letra = palabra[[i]];
 matriz[[1]][[i]] = Flatten[Cases[G1[[3]], {z_, y_List} /; Cases[y, {letra}] =!= {} → z]];
];
(*FIN PRIMERA PARTE *)
```

Esta parte del código implementará el primer bucle del algoritmo que carácter por carácter, calcula los símbolos que lo producen recorriendo las producciones.

Estamos pues, calculando la primera fila de la matriz (j=1).

En la **línea 1** creamos el bucle for para recorrer todos los caracteres de la palabra.

En la **línea 2** obtenemos el carácter a buscar y lo asignamos a la variable letra.

En la **línea 3** obtenemos una lista con los antecedentes que pueden producir ese carácter, tras esto, ponemos la lista en la posición de la matriz que estamos analizando.

Explicación detallada: `Flatten[Cases[G1[[3]], {z_, y_List} /; Cases[y, {letra}] =!= {} → z]];`

Dentro de las producciones queremos obtener las que tienen como consecuente letra:

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

Obtenemos los casos en que las reglas de producción están formadas por algo que llamamos “z” (podríamos forzar a que fuera de tipo List)

Y algo que llamamos “ y ” (en este caso si forzamos a tipo List).

Esto nos devolvería una lista con todas las reglas de producción

Para filtrar las reglas de producción utilizamos el operador Condición:

```
Cases[G1[[3]], {z_, y_List} /; Cases[y, {letra}] =!= {}
```

Esta condición dicta que devolveremos las reglas de producción en las que dentro de la lista de consecuentes existe uno que es { letra }

Como lo que queremos son los consecuentes, añadimos la última parte `→ z` para en lugar de devolver una lista con elementos {z_,y_} ,

devolvamos solo listas con las z (los antecedentes) .

Por último cuando ya tenemos una lista con los antecedentes, esta será una lista de listas asique la aplanamos con Flatten

```
Flatten[Cases[G1[[3]], {z_, y_List} /; Cases[y, {letra}] =!= {} → z]];
```

Tras la primera parte del algoritmo, tendríamos la primera fila de la matriz rellenada, un ejemplo podría ser:

{B}	{A, C}	{A, C}	{B}	{A, C}
0	0	0	0	Null
0	0	0	Null	Null
0	0	Null	Null	Null
0	Null	Null	Null	Null

```
(*INICIO SEGUNDA PARTE *)
For[j = 2, j <= Length[palabra], j++,
 For[i = 1, i ≤ (Length[palabra] - j + 1), i++,
   matriz[[j]][[i]] = {};
   For[k = 1, k <= j - 1, k++,
    caracteresQueProducenElPrefijo = matriz[[k, i]];
    caracteresQueProducenElSufijo = matriz[[j - k]][[i + k]];
```

En esta parte del algoritmo, para cada posible longitud de una subcadena, para cada posición de inicio de subcadena y para cada longitud de prefijo-sufijo ,calcularemos el resto de posiciones posibles de la matriz.

En la **línea 1**, creamos un bucle for que seleccionará la fila de la matriz, como ya hemos analizados las subcadenas de longitud 1, empezamos el bucle en 2.

En la **línea 2**, creamos un bucle for que dada la longitud de la subcadena irá hasta su máxima posición inicial posible.

En la **línea 3**, inicializamos la posición de la matriz a analizar a lista vacía.

En la **línea 4**, creamos un bucle for que recorrerá todas las posibles combinaciones de longitud de prefijo y por tanto de sufijo.

En la **línea 5**, obtenemos la lista de los símbolos que producen el prefijo y la guardamos en caracteresQueProducenElPrefijo.

En la **línea 6**, obtenemos la lista de los símbolos que producen el sufijo y la guardamos en caracteresQueProducenElSufijo.

```
    antecedentes = Flatten[Cases[G1[[3]], {z_, y_List} /; ! MemberQ[matriz[[j]][[i]], z[[1]]] && Cases[y, {y1_, y2_} /; MemberQ[caracteresQueProducenElPrefijo, y1] &&
            MemberQ[caracteresQueProducenElSufijo, y2]] =!= {} → z]];
    If[antecedentes =!= {}, matriz[[j]][[i]] = Join[matriz[[j]][[i]], antecedentes]];
   ];
  ];
 ];
 (*FIN SEGUNDA PARTE *)
 MemberQ[matriz[[Length[palabra], 1]], G1[[4]]]
]
```

La **línea 1** guarda en antecedentes la lista de símbolos que producen el prefijo de longitud k y el sufijo de longitud j-k sin incluir los símbolos que ya hubiéramos encontrado anteriormente (sin añadir repetidos).

Comenzaremos con el calculo de los elementos que se añaden a la matriz:

```
Flatten[Cases[G1[[3]], {z_, y_List} /; ! MemberQ[matriz[[j]][[i]], z[[1]]] && Cases[y, {y1_, y2_} /; MemberQ[caracteresQueProducenElPrefijo, y1] &&  MemberQ[caracteresQueProducenElSufijo, y2]] =!= {} → z]];
```

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

Obtenemos las reglas de producción que cumplen que están formadas por algo (una lista de un elemento) y una lista. (Esto lo cumplen todas)

```
Cases[G1[[3]], {z_, y_List} /; (*Condiciones adicionales*)
```

Obtendríamos las reglas anteriores y que cumplen las condiciones establecidas con el operador Condición (/;)

```
Cases[G1[[3]], {z_, y_List} /; ! MemberQ[matriz[[j]][[i]], z[[1]]] && Cases[y, {y1_, y2_}
```

Obtenemos las reglas de la forma { z, y } donde z no pertenece a la posición de la matriz que además dentro de y tienen alguna producción de dos elementos

```
Cases[G1[[3]], {z_, y_List} /; ! MemberQ[matriz[[j]][[i]], z[[1]]] && Cases[y, {y1_, y2_} /; MemberQ[caracteresQueProducenElPrefijo, y1] &&  MemberQ[caracteresQueProducenElSufijo, y2]] =!= {}
```

Obtenemos todas las reglas que tienen un antecedente que no está en la matriz , que tienen alguna producción de dos elementos y que el elemento1 pertenece a caracteresQueProducenElPrefijo y el elemento2 pertenece caracteresQueProducenElSufijo.

Tras obtener la lista de reglas que cumplen lo anterior, añadimos `→ z` para en lugar de devolver una lista con las reglas que lo cumplen, devolver una lista con los antecedentes que lo cumplen. Tendríamos una lista de la forma { {ant1}, {ant2}…}.

Aplanamos la lista mediante Flatten y obtenemos una lista de la forma { ant1, ant2, …} .Por último, realizamos un Join de esta lista con la de matriz en la posición actual.

Tras esto ya tenemos calculada la matriz y solamente debemos comprobar si el símbolo inicial esta entre los que pueden producir la subcadena ==cadena

, esta información está en `matriz[[Length[palabra], 1]]` , por lo que la función devuelve como resultado `MemberQ[matriz[[Length[palabra], 1]], G1[[4]]]`