Prácticas de MC

Índice

Trabajo realizado por Nikita Stetskiy

Práctica 1

Construir un Autómata Finito Determinístico para aceptar cadenas de ceros y de unos tales que el número de ceros no es múltiplo de tres. Para la correcta visualización del autómata usaremos JFLAP.

Solución

Una manera sencilla de conseguir que el autómata funcione, es hacer que los estados del autómata puedan avanzar si leemos un cero, de esa manera si empezamos en el estado inicial, q_0, y finalizamos en el mismo haremos lo contrario de lo que nos pide el ejercicio, pero al intercambiar los estados finales obtendremos la solución original.

Por tanto, nuestra primera aproximación seria de esta manera:

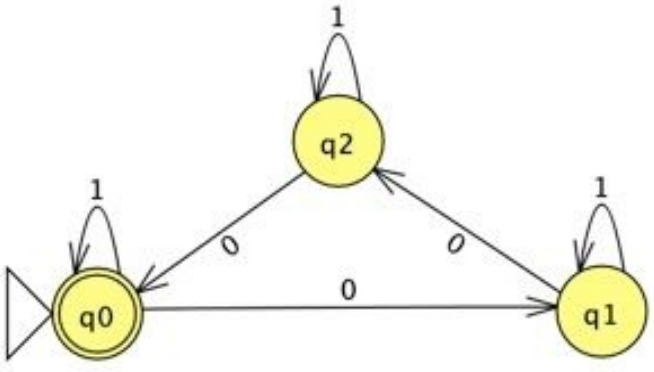

Figura 1

De esta manera si leemos múltiplos de cero o ε avanzaremos a estado final, y si leemos un uno nos quedaremos en el mismo estado. Luego, si cambiamos los estados finales obtendremos esta aproximación a la solución:

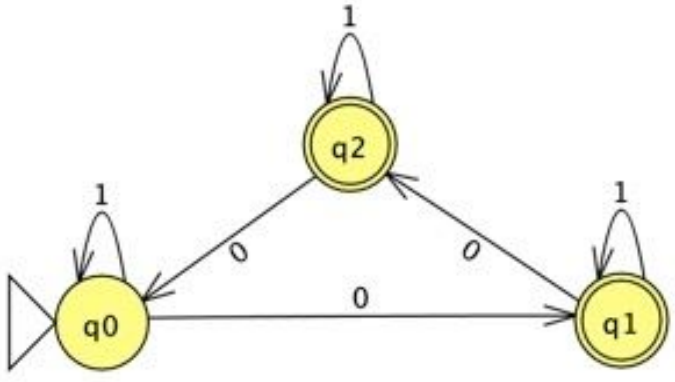

Figura 2

De esta manera rechazaremos (ε, 000, 00101000), ya que si lee un múltiplo de 3 vuelve a q_0, el cual no es estado final, pero aceptaremos (010, 011, 11100010)

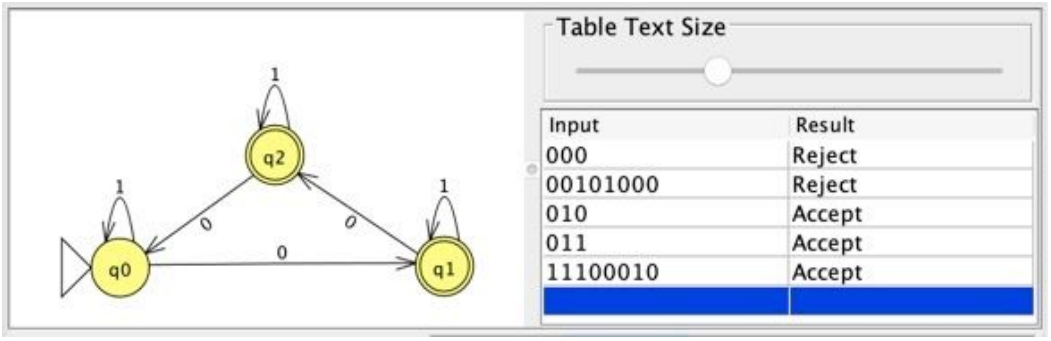

Figura 3

Práctica 2

Elegir un lenguaje regular, para ese lenguaje obtener la expresión regular para representar las cadenas de eses lenguaje y así obtener el autómata. Convertirlo a AFD minimal mediante JFLAP.

Solución

Necesitamos construir un lenguaje regular, para describir un conjunto de cadenas sin enumerar sus elementos. En nuestro caso un lenguaje regular definido en el alfabeto {0,1} que acepta palabras en las que este la subcadena 101.

$$r = (0+1)^*101(0+1)^*$$

- Acepta: 101, 01011, 1010100
- Rechaza: ε, 0, 010, 11000

Introducimos la expresión r, y la convertimos en NFA (Convert → Convert to NFA). El AFND con transiciones nulas obtenido es el siguiente:

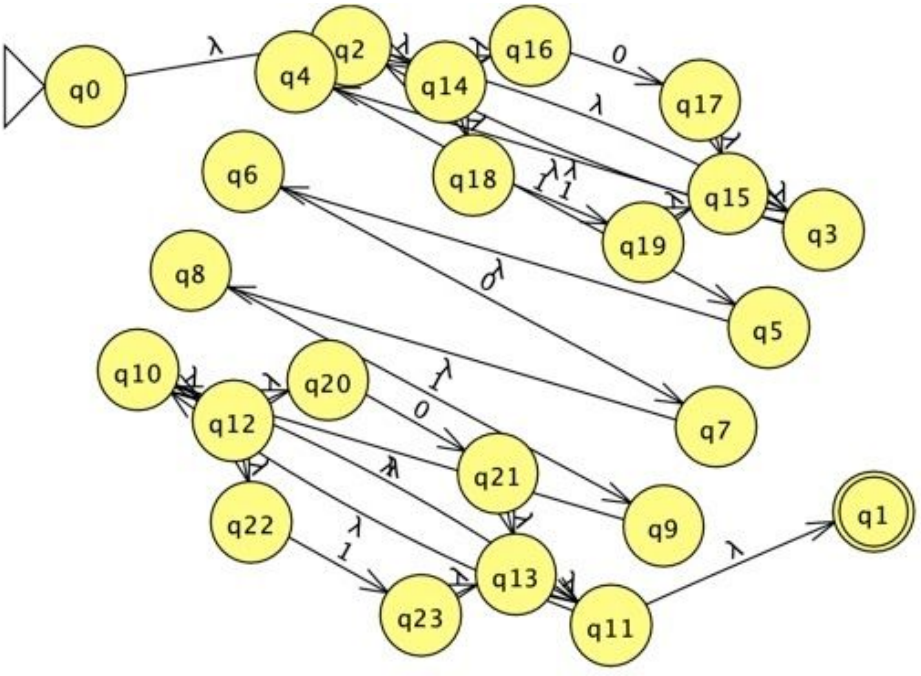

Figura 4

Lo convertimos a AFD, con la función Convert → Convert to DFA:

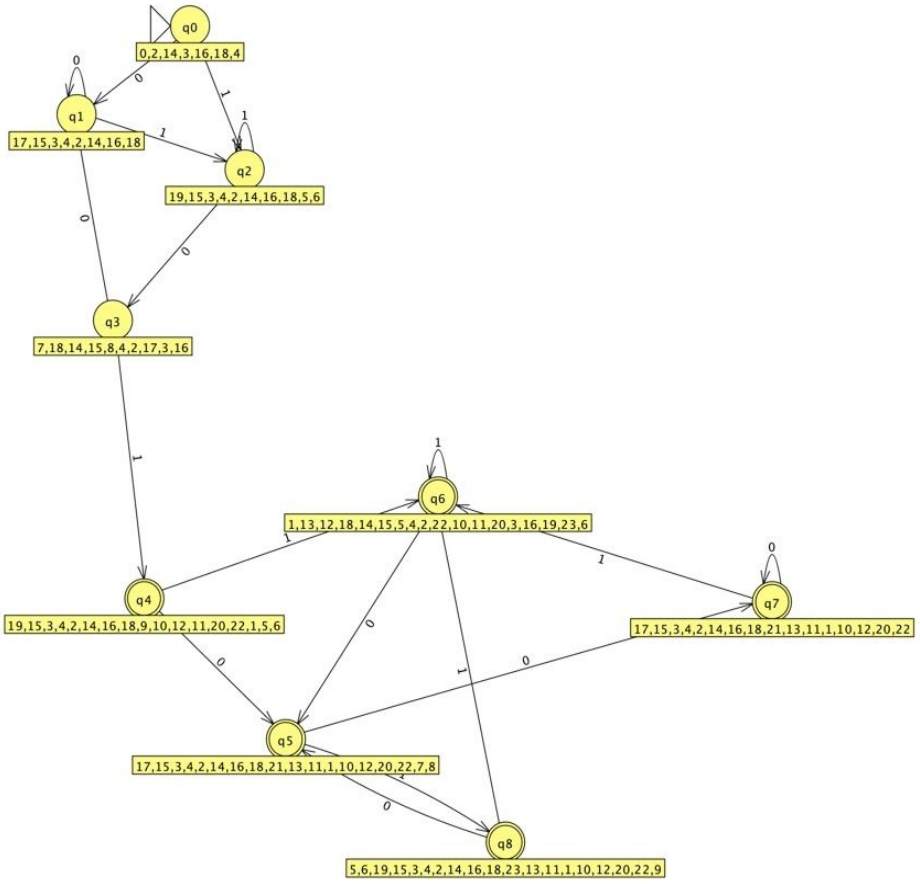

Figura 5

Gracias a la función de JFLAP Convert → Minimize DFA, lo minimizamos:

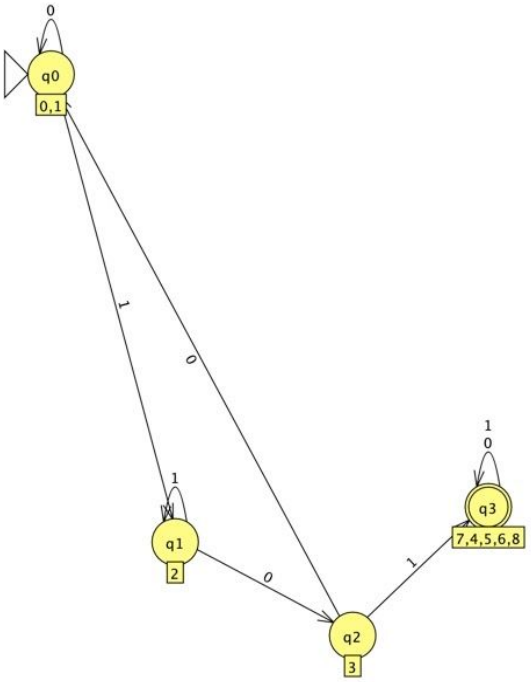

Figura 6

Con la función input, podemos insertar varias cadenas para probar la solución del problema:

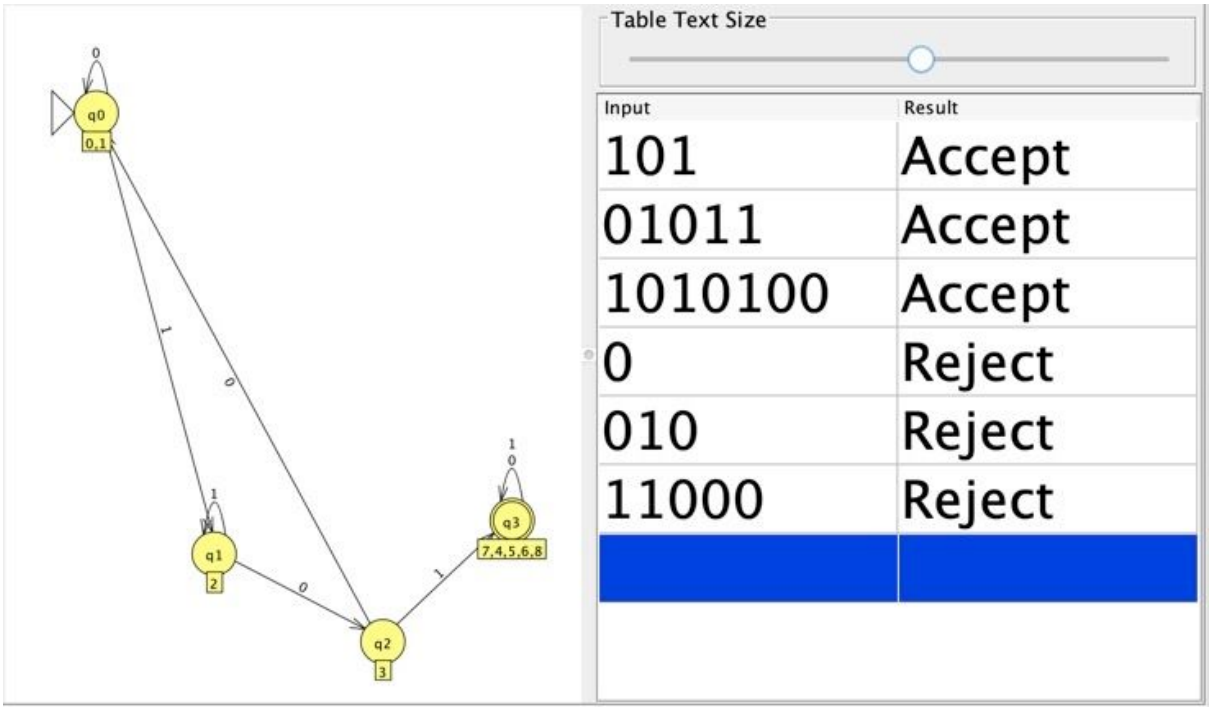

Figura 7

Práctica 3

Elegir un lenguaje regular, para ese lenguaje obtener la expresión regular para representar las cadenas de eses lenguaje con un LEX. Crear un fichero, ejecutar y compilar sobre el LEX.

Solución

Tenemos estados que aceptan a-z y A-Z, 0-9, ambos, el signo +. En el ejemplo de correo pasa de q_0, el cual acepta letras y números, a q_1 solo cuando recibe una “@” y q_2 con un “.” pudiendo repetir el estado q_2 que acepta a-z.

Código:

```
letra [a-zA-Z]
num [0-9]
letra_num [a-z.A-Z.0-9\_]
signo (\+)
telefono_m ({signo}?{num}{11})
telefono_d ({num}{9})
dominio   (\@{letra}+\.)
correo ({letra_num}+{dominio}{letra}{2,})
mejores_peliculas ("Shrek"|"Shrek 2"|"Shrek 3")
%%
.|\n
{telefono_m} {printf("\nTelefono valido\n");}
{telefono_d} {printf("\nTelefono valido\n");}
{correo} {printf("\nCorreo valido\n");}
{mejores_peliculas} {printf("\nMejor pelicula valida\n");}
.+ {printf("\nIncorrecto\n");}
%%
int main(){
   yylex();
   printf("\n");
   return 0;
}
```

En el fichero se definen:

Las definiciones
%%
Las reglas
%%
Las subrutinas del usuario

Hemos definido las letras, números y el signo. Seguidamente de las delimitaciones el bloque de las reglas está compuesto por 5 reglas en la cual escoge si un telefono, correo o pelicula es válida, en caso diferente imprime “Incorrecto”.
En las subrutinas hemos descrito el main que llama a las funciones de las expresiones regulares gracias a yylex() y las ejecuta.

Comprobación:

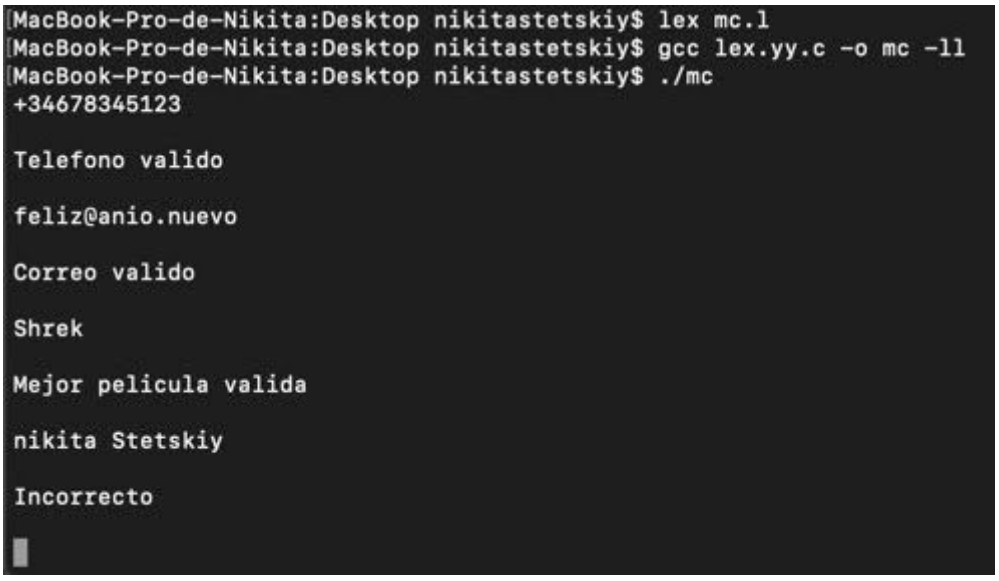

Figura 8

Práctica 4

Usar JFLAP para demostrar que una gramática libre de contexto sea ambigua

$$
\begin{aligned}
E &\Longrightarrow I \\
E &\Longrightarrow I - E \\
E &\Longrightarrow E - I \\
I &\Longrightarrow a|b|c|d
\end{aligned}
$$

En nuestra gramática se escogen estas producciones, para comprobar su ambigüedad se debe encontrar una palabra que pueda admitir dos árboles de derivación diferentes, se aplicará la derivación tanto por la derecha, como por la izquierda. Tomaremos esta palabra para demostrarlo: *a − b − c*

· Definimos las reglas
· Introducimos la gramática en “Grammar” - (*Figura 9)*
· Utilizamos la ventana de “Brute Parser”
· Introducimos la palabra tomada para obtener nuestro árbol - (*Figura 10)*
· Introducimos la palabra cambiando las reglas de orden - (*Figura 12)*

LHS		RHS
E	→	I
E	→	I-E
E	→	E-I
I	→	a|b|c|d

Figura 9

Como conclusión, podemos ver claramente que los árboles son diferentes utilizando la misma palabra, tanto obteniendo el árbol de derivación por la izquierda y derecha, los dos árboles de derivación son distintos, por lo que la gramática es ambigua. Eliminando la regla *E ⇒ I−E* ó *E ⇒ E−I,* la gramática dejaría de ser ambigua.

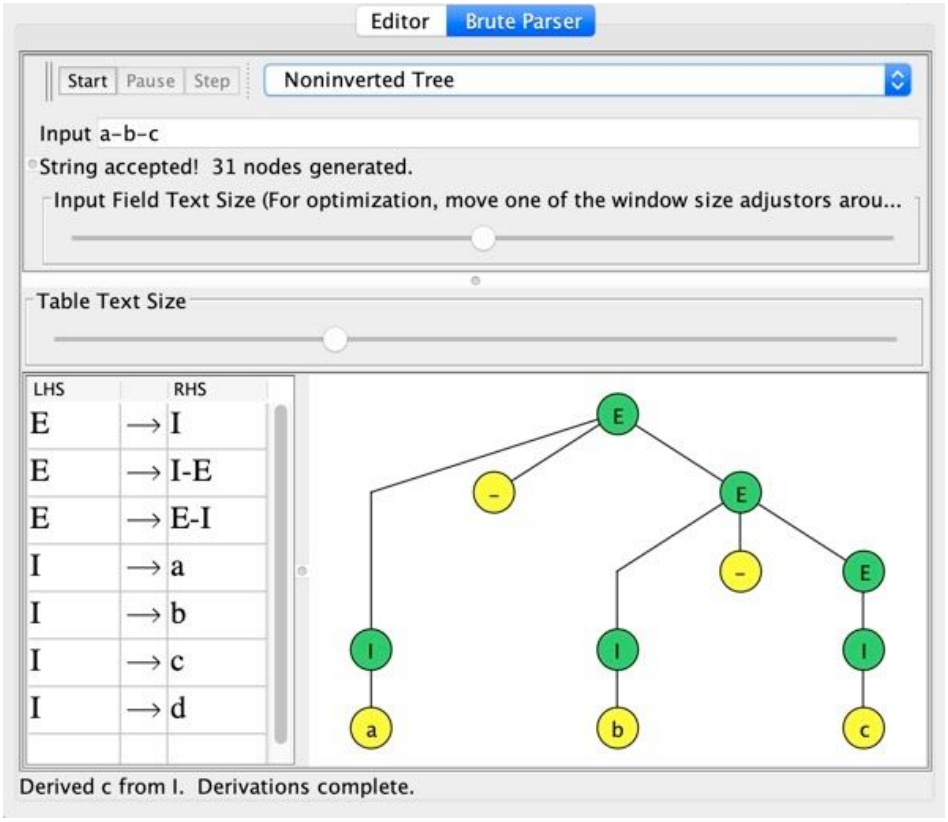

Figura 10

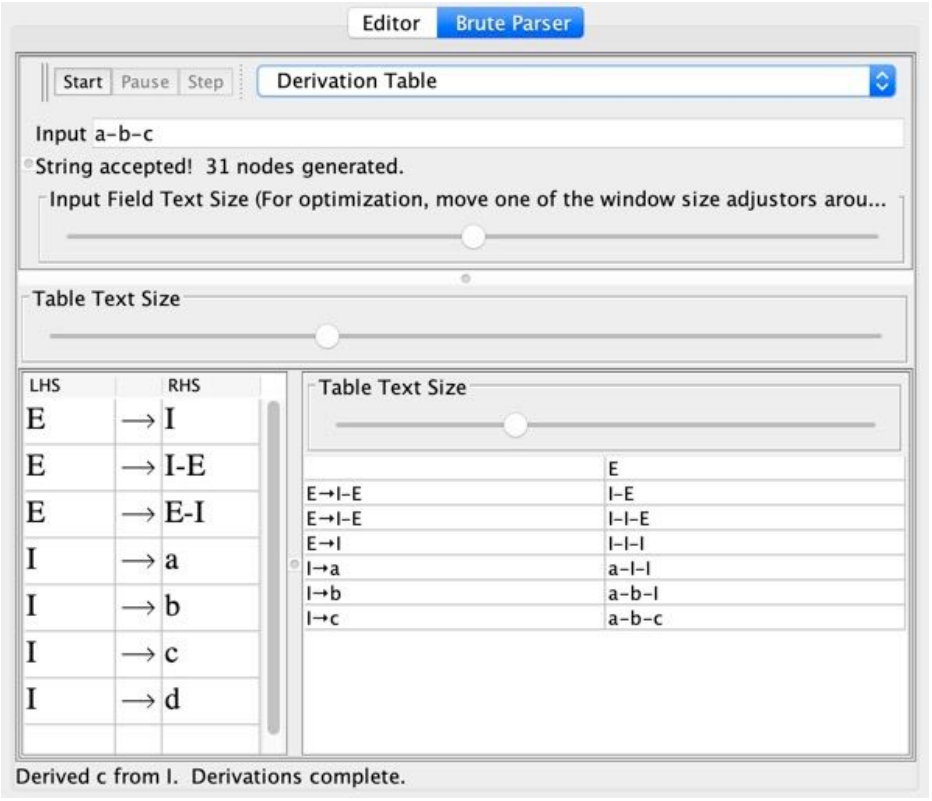

Figura 11

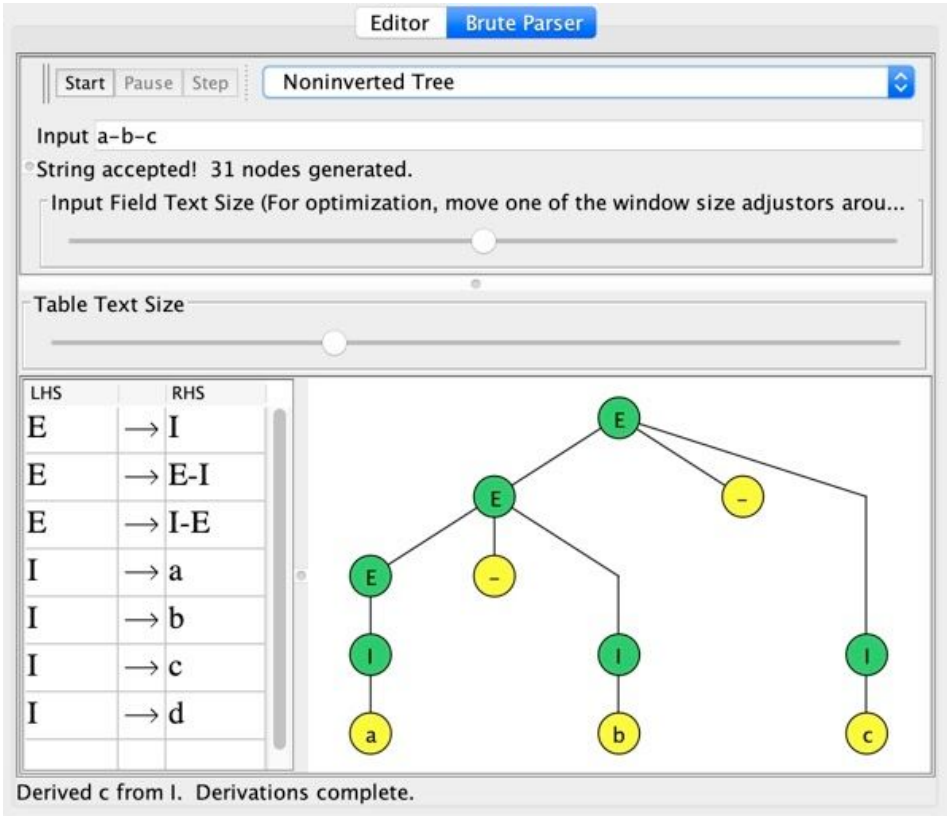

Figura 12

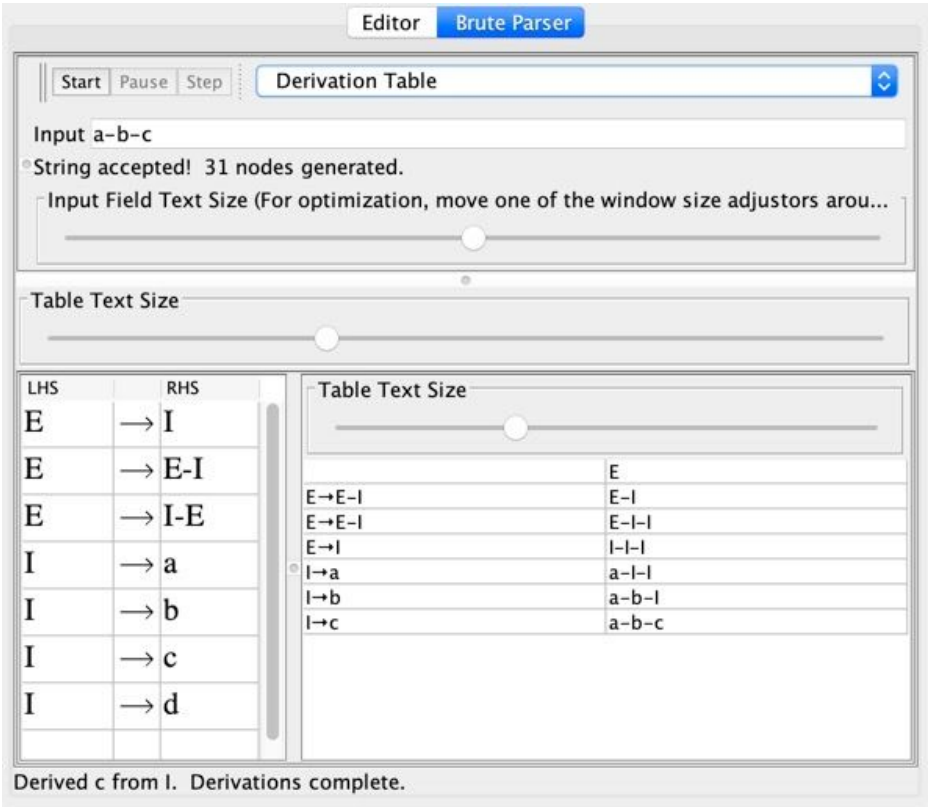

Figura 13