Proyecto 01

Construcción de un Analizador Léxico

Universidad Nacional Autónoma de México

Facultad de Ciencias

Materia: Compiladores

Profesor
Manuel Soto Romero

Ayudantes:
Jose Manuel Evangelista Tiburcio
José Alejandro Pérez Márquez
Fausto David Hérnandez Jasso
Diego Méndez Medina

Equipo Ternunenes

Integrantes:
Hernandez Zavala Ana Sofia - No. Cuenta: 319316717
Mendiola Montes Victor Manuel - No. Cuenta: 320197350
Sanluis Castillo Daniela Alejandra No. Cuenta: 320091179

Fecha de entrega : 23/Octubre/2025

Introducción

Este proyecto tiene como propósito construir un analizador léxico para un lenguaje imperativo (IMP) bastante sencillo. Esto se hizo tomando en cuenta todos los tokens de IMP, con los que generamos sus expresiones regulares, y a partir de ellas hacer una construcción del siguiente modo:

$$\text{ER} \rightarrow \text{AFN}\varepsilon \rightarrow \text{AFN} \rightarrow \text{AFD} \rightarrow \text{AFDmin} \rightarrow \text{MDD} \rightarrow \texttt{lexer}$$

Para esto, decidimos hacer un modulo que se encargará de cada transformación, y fuimos creando nuestro proyecto con dos metodologías hasta que dimos con la indicada para poder presentar este trabajo que funciona para este lenguaje IMP.

Preliminares formales

Lenguajes Formales

- **Alfabeto:** Es un conjunto finito de símbolos. Este se representa con Σ. Por ejemplo, $\Sigma = \{a, b, c\}$
- **Cadena:** Es una secuencia finita de símbolos extraídos de un alfabeto. Por ejemplo, *'abc'*.
- **Lenguaje formal:** Es un conjunto de cadenas formadas por símbolos extraídos de un alfabeto finito. Por ejemplo, $L = \{$ *'abc', 'bca', 'cab'* $\}$

Tokens

- **Token:** Son las unidades significativas más pequeñas del programa. Cada token esta definido como `Token = (Categoria, Lexema)`. La Categoría es el nombre (o, valga de redundancia, la categoria) del token (Por ejemplo, identificador, entero, operador, etc.), y el lexema es la cadena concreta del programa. En el contexto del diseño de un compilador, cada token del lenguaje se describe mediante una expresión regular.

Expresiones Regulares

Es la especificación formal y compacta de los patrones que definen un lenguaje regular. En el contexto de un compilador, se usan para describir los patrones de los tokens.

Dado un alfabeto Σ, una ER se define recursivamente:

- $\varnothing \rightarrow$ Conjunto vacio.
- $\varepsilon \rightarrow$ Representa una cadena vacia y denota el conjunto $\{\varepsilon\}$.
- a $\rightarrow$ Para cualquier símbolo a en Σ, a es una expresión regular y denota el conjunto $\{a\}$.

Ahora supongamos que tenemos dos expresiones regulares r y s, que vienen del lenguaje R y S respectivamente. También podemos construir expresiones regulares de forma recursiva con r y s.

- $(r + s) \rightarrow$ Representa la unión de los lenguajes de r y s.
- $(rs) \rightarrow$ Representa la concatenación de los lenguajes r y s
- $(r)^* \rightarrow$ Representa la cerradura de Kleene del lenguaje r

Recordemos que la cerradura de Kleene de una expresión regular representa a todas las cadenas que puedes formar concatenando cero, una o más cadenas de un lenguaje L.

Autómatas

Todo lenguaje regular L (y expresión regular l de L) puede representarse mediante un autómata, que a grandes rasgos, es un modelo matemático de una máquina que procesa cadenas símbolo por símbolo y va cambiando de estado según reglas previamente definidas.

Un autómata finito esta conformado por la siguiente tupla: $M = (\ \Sigma\ , Q\ , q_0\ , F\ , \delta\ )$. Σ representa el alfabeto del lenguaje regular que acepta M. Q es el conjunto de estados. q_0 es el estado inicial (por donde empieza). F es el conjunto de estados de aceptación (que toman una cadena como válida). δ representa las reglas de transición, que son las reglas que nos dicen a que estado movernos dentro de M dependiendo del símbolo leído.

Tenemos 4 tipos de autómatas:

- **Autómata Finito No Determinista con Transiciones ε (AFNε)**: En las reglas de transición de este autómata se aceptan las transiciones con la cadena vacía ε. Es decir, nos podemos mover hacia ciertos estados sin consumir símbolos de la cadena de entrada.
- **Automata Finito No Determinista (AFN)**: En las reglas de transición de este autómata se permiten múltiples transiciones desde un mismo estado con el mismo símbolo. Es decir, que podemos podríamos ir a más de un estado leyendo un mismo símbolo de la cadena de entrada.
- **Automata Finito Determinista (AFD)**: En las reglas de transición de este autómata cada estado tiene exactamente una transición para cada símbolo del alfabeto. Apartir de este autómata se pueden implementar analizadores léxicos eficientes garantizando un comportamiento determinista, sin embargo, podemos optimizar esta implementación.
- **Autómata Finito Determinista Minimizado (AFDmin)**: A partir del AFD se aplica un proceso de minimización, que agrupa los estados equivalentes y los inalcanzables para eliminarlos sin afectar el lenguaje que acepta el AFD).El resultado es un AFD optimizado, pues cuenta con el menor número posible de estados que reconoce el mismo lenguaje, lo que reduce la huella de memoria del analizador y simplifica las tablas de transición.

Máquina Discriminadoras Deterministas y Analizador Léxico

Una **Máquina Discriminadora Determinista (MDD)** esta derivada de un AFD (en nuestro caso, un AFDmin) y es una herramienta fundamental en el análisis léxico, pues su característica principal es que tiene una función que asigna categorías léxicas a sus estados de aceptación.

Gracias a esto, la MDD acepta una cadena y la categoriza. Podríamos decir que es como un puente entre la teoría de autómatas y la implementación práctica del análisis léxico.

El **Analizador Léxico (Lexer)** es un componente esencial de un compilador. Su función es transformar nuestro código fuente en una secuencia estructurada tokens que sean reconocibles el analizador sintáctico. La MDD define el comportamiento del lexer.

Podemos decir que cada una de estas representaciones (desde ER hasta lexer), se obtiene a partir de la anterior, garantizando que el lenguaje se conserva en cada transformación:

$$ER \Rightarrow AFN\varepsilon \Rightarrow AFND \Rightarrow AFD \Rightarrow AFD_{min} \Rightarrow MDD.$$

Lenguaje IMP

El Lenguaje IMP es el lenguaje que usaremos para poder probar nuestro analizador léxico. Este es un lenguaje de programación imperativo muy simple, que esta definido de la siguiente manera:

- **Expresiones Aritméticas (Aexp):**
  - $n \rightarrow$ Un número entero.
  - $x \rightarrow$ Un identificador, o variable.
  - $Aexp + Aexp \rightarrow$ La suma de dos expresiones aritméticas.
  - $Aexp - Aexp \rightarrow$ La resta de dos expresiones aritméticas.
  - $Aexp * Aexp \rightarrow$ El producto de dos expresiones aritméticas.
- **Expresiones Booleanas (Bexp):**
  - $true \rightarrow$ Valor true.
  - $false \rightarrow$ Valor false.
  - $Aexp = Aexp \rightarrow$ Igualdad de dos expresiones aritméticas.
  - $Aexp \leq Aexp \rightarrow$ Menor o igual para dos expresiones aritméticas.
  - $not\ Bexp \rightarrow$ Negación de una expresión booleana.
  - $Bexp\ and\ Bexp \rightarrow$ Conjunción de dos expresiones booleana.
- **Comandos (Com):**
  - $skip \rightarrow$ Comando vacio.
  - $x := Aexp \rightarrow$ Asignación de una expresión aritmética a una variable.
  - $Com\ ;\ Com \rightarrow$ Secuencia de comandos.
  - $if\ Bexp\ then\ Com\ else\ Com \rightarrow$ Estructura del if.
  - $while\ Bexp\ do\ Com \rightarrow$ Estructura del ciclo while.

Metodología y su implementación

Antes de explicar nuestra implemetación y metodología, quisiera poner la estructura de nuestro proyecto, que también se encuentra en el README.md

```
Proyecto/
|
|-- docs/
|     | --  Reporte.pdf (Este documento)
|
|-- main/
|    |-- Main.hs (Programa principal)
|    |-- implementacion.imp (Archivo de entrada para Main)
|
|-- src/
|    |-- AFD.hs (Logica para pasar de AFN a AFD)
|    |-- ADFmin.hs (Logica para pasar de AFD a AFDmin)
|    |-- AFN.hs (Logica para pasar de AFNepsilon a AFN)
|    |-- AFNEp.hs (Logica para pasar de ER a AFNepsilon)
|    |-- ER.hs (Logica para construir una ER)
|    |-- Gramatica.hs (Gramatica del lenguaje IMP)
|    |-- Lexer.hs (Construcción de las MDD's)
|    |-- MDD.hs (Lógica para construir una MDD con AFDmin)
|
|-- test/
|    |-- Test.hs (Pruebas unitarias)
|
|-- package.yaml
|-- Proyecto01-Analizador-L-xico.cabal
|-- README.md (Este archivo)
|-- stack.yaml
```

(1) Expresiones Regulares

El objetivo de esta fase es usar la construcción recursiva de las ER para traducir las reglas del lenguaje IMP en una ER. Para esto, lo primero que hicimos fue definir una estructura de datos capaz de representar esta definición recursiva. Esto se logró en el módulo `ER.hs` con un tipo de dato algebraico:

```
-- (Modulo ER.hs)

data Expr = Term Char          -- Caso base 'a'
          | Or Expr Expr       -- (r + s)
          | And Expr Expr      -- (rs)
          | Kleene Expr        -- (r)^*
          | Range Char Char    -- Definición de un rango de símbolos
  deriving (Eq)
```

Adicionalmente, incluimos un constructor *Range Char Char* para facilitarnos la vida (y la complejidad del proyecto). Aunque todas letras de la *a* a la *z* podría definirse como *Term 'a'* | *Term 'b'* | ... | *Term 'z'*, nuestro constructor *Range* nos permite representar rangos de caracteres de forma compacta, que nos es muy útil para definir rangos como [0-9] o [a-z] de manera eficiente.

Una vez definido `ER.hs`, vamos a utilizarlo en un nuevo modulo llamado `Gramatica.hs`, que nos servirá para construir las ER del lenguaje IMP. Este módulo importa ER y lo utiliza para crear valores de tipo Expr que representan cada token. A continuación explicamos nuestra construcción de cada expresión regular.

- **Palabras Reservadas:** Nosotros tomamos a `true, false, not, and, skip, if, then, else, while` ý `do` como palabras reservadas. Nuestra idea es unir a todas estas palabras con el operador + para las ER, pues una vez que se declaro una palabra reservada, no podemos usar otra en seguida de esa (como concatenarlas):

  ```
  true + false + not + and + skip + if + then + else + while + do
  ```

  En nuestra implementación de haskell, para generar cada palabra simplemente concatenamos todos los símbolos (letras) de la palabra. Por ejemplo, con nuestra definición de `ER.hs`, la palabra *true* queda como:

  ```
   -- Palabra true
  true_ = And (Term 't') (And (Term 'r') (And (Term 'u') (Term 'e')))
  ```

  Hacemos esto para cada palabra reservada y al final las unimos a todas con el operador OR para nuestras ER.

  ```
  palabrasReservadas :: Expr
  palabrasReservadas = Or true_ (Or false_ (Or not_ (Or and_ ... )))
  ```

- **Identificadores:** En clase vimos que para definir un identificador de IMP siempre debe de iniciar con una letra (ya sea min o MAY) y después, si así se quiere hacer, se pueden agregar más letras minúsculas, mayúsculas y dígitos. Pensándolo asi, tenemos la siguiente ER:

  $$([a\text{-}z] + [A\text{-}Z])([a\text{-}z] + [A\text{-}Z] + [0\text{-}9])^*$$

  Esto obliga a que nuestros identificadores siempre inicien con una letra y despues pongamos cualquier letra o digito (si queremos).

  En haskell, lo primero que podemos hacer para ahorrarnos tiempo y código, es definir los rangos de las letras mayusculas, minusculas y de los digitos. Esto con nuestra definición de `Range` en `ER.hs`

  ```
  letraMayus, letraMinus, digito, :: Expr
  letraMayus = Range 'A' 'Z'   -- Todas las letras mayúsculas.
  letraMinus = Range 'a' 'z'   -- Todas las letras minúsculas.
  digito = Range '0' '9'       -- Números del 0 al 9.
  ```

  Ahora, solo nos queda hacer la ER en Haskell con nuestros operadores AND y OR.

  ```
  identificador :: Expr
  identificador =  And (Or letraMayus letraMinus)
                       (Kleene (Or (Or letraMayus letraMinus) digito))
  ```

- **Enteros:** Este fue nuestro ejemplo más visto en clases, y definimos que un entero podría ser 0, un entero positivo o un entero negativo. Esto lo declaramos como:

  $$0 + ([1\text{-}9][0\text{-}9]^*) + -([1\text{-}9][0\text{-}9]^*)$$

  En haskell, vamos a tener dos rangos: Los digitos del 0 al 9 (que ya definimos) y los digitos sin el cero:

  ```
  digitoSinC = Range '1' '9'    -- Números del 1 al 9.
  ```

  Solo nos queda implementarlo en Haskell:

```
enteros :: Expr
enteros = Or (Term '0')
             (Or (And (digitoSinC) (Kleene digito))
                 (And (Term '-') (And (digitoSinC) (Kleene digito)))
             )
```

- **Operadores:** Los operadores que tenemos son `+, -, *, =, <=` y `:=`. Al igual que en las palabras reservadas, pues solo podemos usar un operador:

  `+ | - | * | = | <= | :=`

  La implementación en haskell es análoga a las palabras reservadas:

  ```
  operadores :: Expr
  operadores = Or (Or (Or (Term '+')(Term '-'))(Term '*'))...(Term '='))
  ```

- **Delimitadores:** En IMP, solo contamos con `';'` y con los paréntesis `'('` `')'`, asi que solamente hacemos la unión entre ellos:

  `; | ( | )`

  En haskell quedará muy rápido:

  ```
  delimitadores :: Expr
  delimitadores =   Or (Term ';')(Or (Term '(') (Term ')'))
  ```

- **Espacio en blanco:** Un espacio en blanco se puede representar de 3 formas: Literalmente un espacio en blanco `' '`, tabular `\t`, y un salto de línea `\n`. Casi análogo al caso de los identificadores, nosotros podemos tener 1 ó más espacios en blanco sin que esto altere el funcionamiento en IMP, asi que tenemos la siguiente expresión:

  `(' ' + \t + \n)(' ' + \t + \n)`*

  En nuestra implementación, primero definimos la representación de un espacio en blanco (espacio, tab o salto de linea):

  ```
  espacioBlanco :: Expr
  espacioBlanco = Or (Term ' ') (Or (Term '\t') (Term '\n') )
  ```

Luego, simplemente pasamos la expresión a haskell:

```
espacios :: Expr
espacios = And espacioBlanco (Kleene espacioBlanco)
```

- **Comentarios:** Por último, tenemos los comentarios, que nosotros quisimos que se iniciaran con dos diagonales seguidas de cualesquiera símbolos en el teclado. Para declarar el final de un comentario, se tiene que hacer un salto de línea \n:

  `// ([Simbolos en teclado])* \n`

  Para nuestra implementación, primero hicimos una definición de todos los símbolos en el teclado a partir de *Range*:

  ```
  alfabeto :: Expr
  alfabeto = Range ' ' '~'    -- Todos los símbolos del teclado.
  ```

  Luego, solo hacemos una concatenación para obtener nuestra expresión regular:

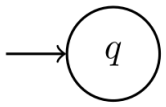
  ```
  comentarios :: Expr
  comentarios = And (Term '/')
                    (And (Term '/') (And (Kleene alfabeto) (Term '\n')))
  ```

(2) ER $\rightarrow$ AFNε

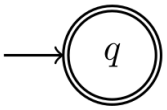
El objetivo de esta fase es traducir una ER a su AFNε equivalente. Esta transformación es la primer traducción fundamental en nuestro analizador léxico y se implementa en el módulo `AFNEp.hs`

Para esta conversión, implementamos el algoritmo de Thompson. Podemos resumir muy rápido los pasos recursivos:

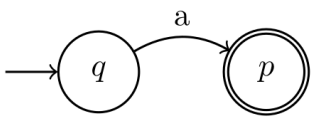
- **Casos Base:**
  - Si la expresión regular es $\varnothing$, el autómata resultante será:



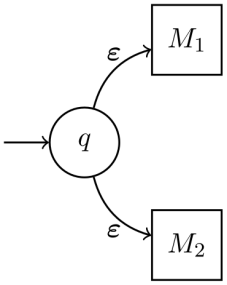
  - Si la expresión regular es ε, el autómata resultante será:



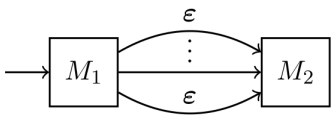
  - Si la expresión regular es un símbolo a, tal que $a \in \Sigma$, el autómata resultante será:



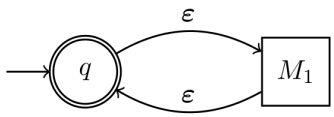
- **Casos Recursivos:** Para esta parte, supongamos que tenemos dos autómatas M_1, M_2 que reconocen los lenguajes de las expresiones regulares l y r respectivamente.
  - Si la expresión regular es $l + r$, el autómata resultante será:



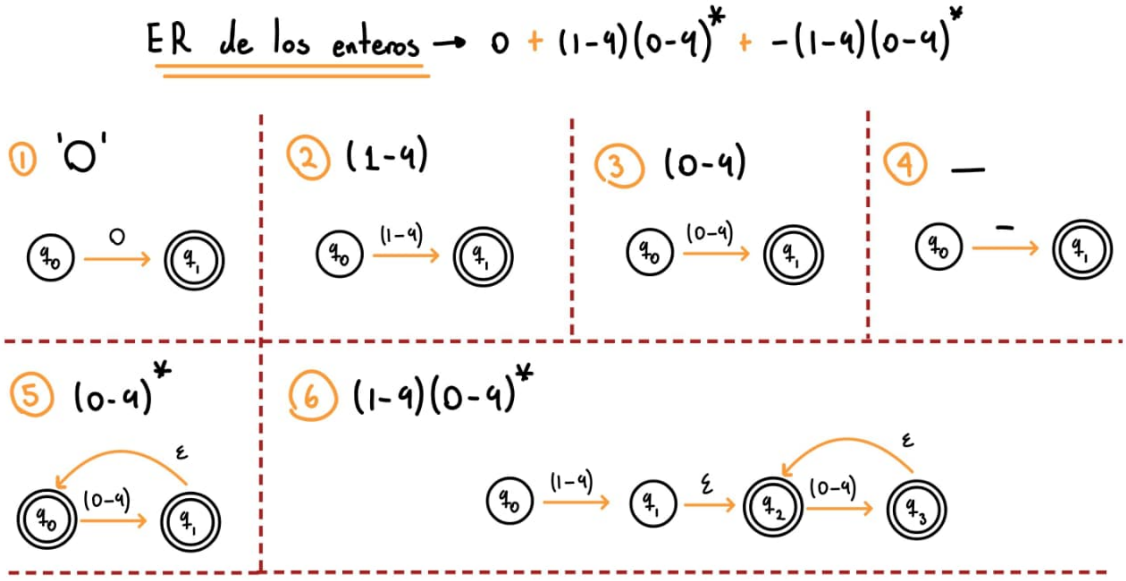
- Si la expresión regular es lr, el autómata resultante será:

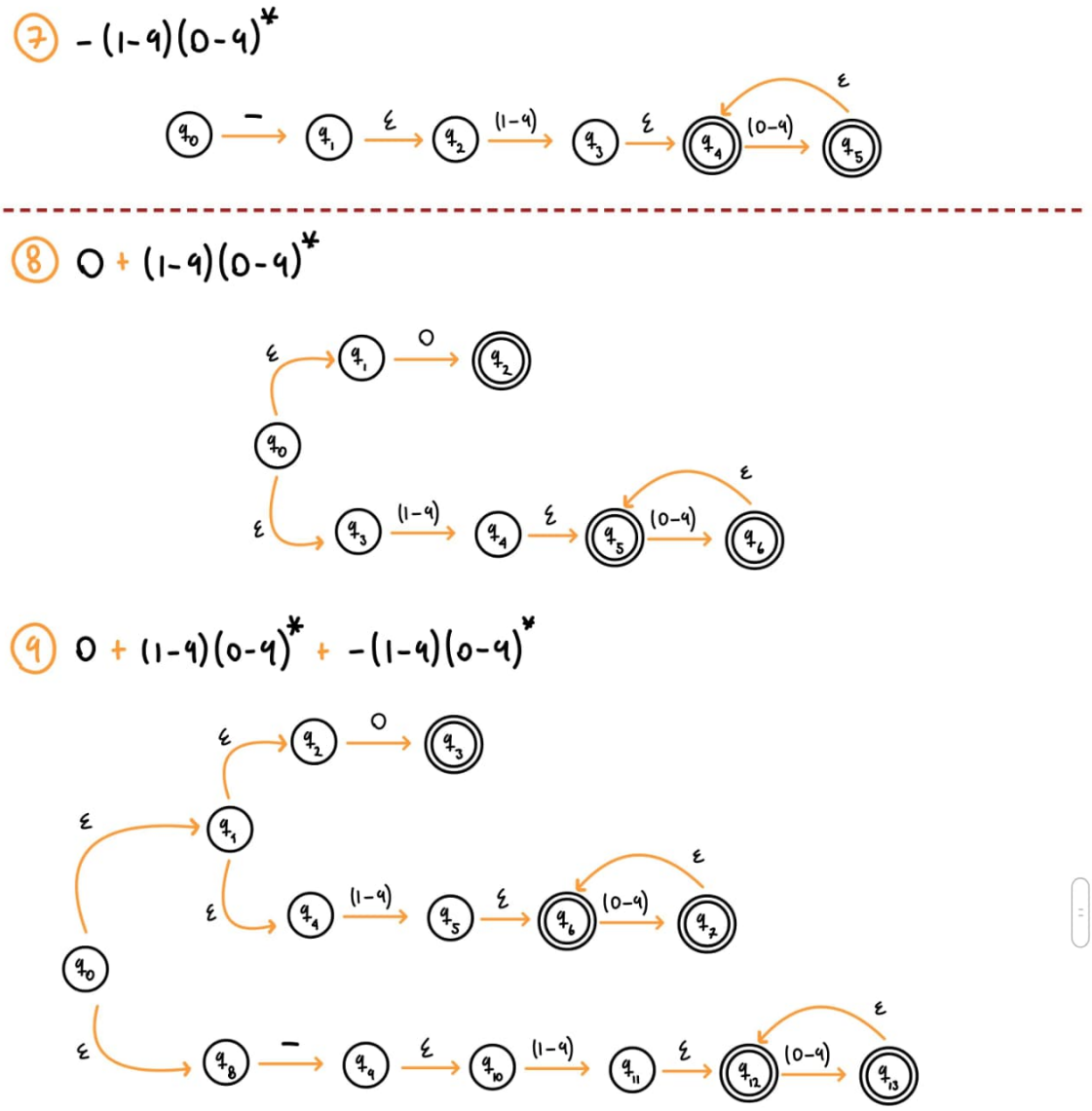


- Si la expresión regular es l^*, el autómata resultante será:



`Por ejemplo`, si queremos traducir nuestra expresión regular de los enteros a una AFNϵ, nos queda de la siguiente manera:





Ahora, para pasar esto a Haskell primero tenemos que implementar la definición de un AFNε. Primero, la definición de los estados y de las transiciones:

```
-- Estados. String para que sea más facil de leer.
type Estado = String

-- Transiciones en AFNe
-- Estado origen -> Símbolo o nada (epsilon) -> lista de Estados destino.
type Trans_eps = (Estado, Maybe Char, [Estado])
```

Con estos dos tipos definidos, ya podemos definir un AFNε

```
-- Def de AFNe
data AFNEp = AFNEp {
    estados :: [Estado],          -- Lista de todos los estados.
    alfabeto :: [Char],           -- Alfabeto que acepta el autómata.
    transiciones :: [Trans_eps],  -- Reglas de transición.
    inicial :: Estado,            -- Estado inicial.
    finales :: [Estado]           -- Lista de estados finales.
    } deriving (Eq)
```

La lógica principal reside en la función `expr_to_AFNEp`. Esta función recorre recursivamente la `Expr` que recibe como entrada y construye el `AFNEp` correspondiente. Para asegurar que cada nuevo estado creado sea único, la función lleva un contador de estados. Entonces la función toma una `Expr` y el contador actual, y devuelve el par del `AFNEp` recién creado y el nuevo valor del contador.

```
-- Función recursiva para pasar de ER a AFNEp
expr_to_AFNEp :: Expr -> Int -> (AFNEp, Int)
```

Dentro de esta función, tenemos nuestra implementación recursiva de Thompson, nosotros definimos 5 casos: 2 bases y 3 recursivos.

- **(A) - (Term c)**: Es nuestro el autómata que reconoce un solo símbolo. Lo que hacemos es crear dos estados, q_n y q_{n+1} (inicial y final), con una única transición entre ellos etiquetada con el símbolo.

  ```
  expr_to_AFNEp (Term c) nuevesito =
      let q0 = estadoNuevo nuevesito        -- Estado inicial q0.
          q1 = estadoNuevo (nuevesito + 1)  -- Estado final q1.
          transicion = (q0, Just c, [q1])   -- La única transición.
      in (AFNEp [q0, q1] [c] [transicion] q0 [q1], nuevesito + 2)
  ```

- **(B) - Range a b**: Este es un caso "base" similar. En lugar de una sola transición, creamos múltiples transiciones desde q_0 a q_1, una por cada símbolo en el rango de a a b.

```
expr_to_AFNEp (Range a b) nuevesito =
    let simbolos = [a..b]                 -- Lista de símbolos.
        q0 = estadoNuevo nuevesito        -- Estado inicial q0.
        q1 = estadoNuevo (nuevesito + 1)  -- Estado final q1.
        -- Lista de transiciones.
        transicion = [(q0, Just c, [q1]) | c <- simbolos]
    in (AFNEp [q0, q1] simbolos transicion q0 [q1], nuevesito + 2)
```

- **(C) - Or expr1 expr2**: Construimos recursivamente los autómatas M_1 y M_2 para $expr1$ y $expr2$. Luego, se crea un nuevo estado inicial q_0, y se añaden dos transiciones ε desde q_0 a los estados iniciales de M_1 y M_2. Los estados finales del nuevo autómata son la unión de los estados finales de M_1 y M_2.

```
expr_to_AFNEp (Or expr1 expr2) nuevesito =
    let (m1, estado1) = expr_to_AFNEp expr1 nuevesito -- Construimos m1.
        (m2, estado2) = expr_to_AFNEp expr2 estado1   -- Construimos m2.
        -- Nuevo estado inicial q0.
        q0 = estadoNuevo estado2
        -- Nuevas transiciones epsilon para conectarlos.
        transEp = [ (q0, Nothing, [inicial m1]),
                    (q0, Nothing, [inicial m2])]
        -- El resto de componentes...
        in (AFNEp nuevosEstados nuevoAlfabeto
                  nuevasTransiciones q0 nuevosFinales, estado2 + 2)
```

- **(D) - And expr1 expr2**: Construimos M_1 y M_2. El estado inicial del nuevo autómata es el de M_1. Los estados finales son los de M_2. La conexión se realiza añadiendo transiciones ε desde cada estado final de M_1 hacia el estado inicial de M_2.

```
expr_to_AFNEp (And expr1 expr2) nuevesito =
    let (m1, estado1) = expr_to_AFNEp expr1 nuevesito  -- Construimos m1.
        (m2, estado2) = expr_to_AFNEp expr2 estado1    -- Construimos m2.
        -- Nuevas transiciones epsilon desde M1 a M2.
        transEp = [(qf_m1, Nothing, [inicial m2]) | qf_m1 <- finales m1]
        -- El resto de componentes...
    in (AFNEp nuevosEstados nuevoAlfabeto
              nuevasTransiciones (inicial m1) (finales m2), estado2)
```

- **(E) - Kleene expr**: Se construye recursivamente el autómata M para $expr$. Añadimos transiciones ε desde cada estado final de M de vuelta al estado inicial. Por último, hacemos que el estado inicial ahora también sea final.

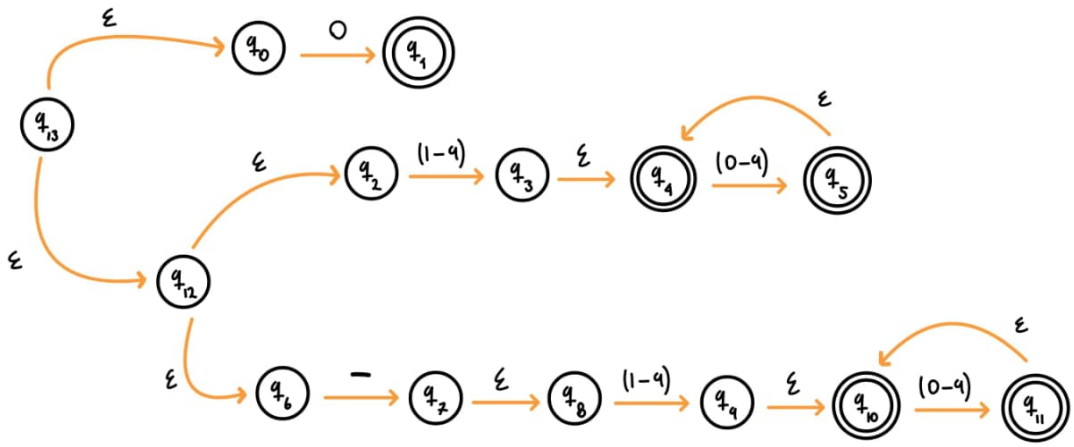
```
expr_to_AFNEp (Kleene expr) nuevesito =
    -- Construimos el autómata m para expr.
    let (m, estado) = expr_to_AFNEp expr nuevesito

        -- Transiciones epsilon desde el estado final al estado inicial.
        transEp = [(qf_m, Nothing, [inicial m]) | qf_m <- finales m]
        -- ...
        -- El estado inicial de m ahora es un estado final.
        nuevosFinales = nub (inicial m : finales m)

    in (AFNEp nuevosEstados nuevoAlfabeto
              nuevasTransiciones (inicial m) nuevosFinales, estado)
```

Para poder probar nuestra implementación, hicimos unos tests rápidos en el módulo `TestAFNEp.hs`. Dicho modulo, nos imprime los AFNε de las siguientes ER : $0(0+1)^*$, $(ab+b)^*ab^*$ ý $0 + [1-9][0-9]^* + -[1-9][0-9]^*$.

Para seguir con el ejemplo de los enteros, el AFNε resultante del programa es:

```
-----------------------------------------------------------------------
                ***** Imprimiendo AFNEp *****

>> estados      = ["q13","q0","q1","q12","q2","q3","q4","q5","q6","q7","q8","q9","q10","q11"]

>> alfabeto     = "0123456789-"

>> transiciones = [("q0",Just '0',["q1"]),("q2",Just '1',["q3"]),("q2",Just '2',["q3"]),("q2",Just '3',["q3"]),("q2",Just '4',["q3"]),("q2",Just '5',["q3"]),("q2",Just
'6',["q3"]),("q2",Just '7',["q3"]),("q2",Just '8',["q3"]),("q2",Just '9',["q3"]),("q3",Nothing,["q4"]),("q4",Just '0',["q5"]),("q4",Just '1',["q5"]),("q4",Just '2',["q5
"]),("q4",Just '3',["q5"]),("q4",Just '4',["q5"]),("q4",Just '5',["q5"]),("q4",Just '6',["q5"]),("q4",Just '7',["q5"]),("q4",Just '8',["q5"]),("q4",Just '9',["q5"]),("q
5",Nothing,["q4"]),("q6",Just '-',["q7"]),("q7",Nothing,["q8"]),("q8",Just '1',["q9"]),("q8",Just '2',["q9"]),("q8",Just '3',["q9"]),("q8",Just '4',["q9"]),("q8",Just '
5',["q9"]),("q8",Just '6',["q9"]),("q8",Just '7',["q9"]),("q8",Just '8',["q9"]),("q8",Just '9',["q9"]),("q9",Nothing,["q10"]),("q10",Just '0',["q11"]),("q10",Just '1',[
"q11"]),("q10",Just '2',["q11"]),("q10",Just '3',["q11"]),("q10",Just '4',["q11"]),("q10",Just '5',["q11"]),("q10",Just '6',["q11"]),("q10",Just '7',["q11"]),("q10",Jus
t '8',["q11"]),("q10",Just '9',["q11"]),("q11",Nothing,["q10"]),("q12",Nothing,["q2"]),("q12",Nothing,["q6"]),("q13",Nothing,["q0"]),("q13",Nothing,["q12"])]

>> inicial      = "q13"

>> finales      = ["q1","q4","q5","q10","q11"]

Listo :D
-----------------------------------------------------------------------
```

Y, si hacemos su representación visual, obtenemos lo siguiente:



Con lo que podemos concluir que nuestra implementación es correcta ☺

(3) AFNε $\rightarrow$ AFN

El objetivo de esta fase es eliminar las transiciones ε de una AFNε para obtener un AFN equivalente. Esta lógica esta implementada en el módulo `AFN.hs`.

Para esta conversión, implementamos el algoritmo basado en las $\varepsilon - cerradura$ de un estado q (denotado $ECLOSURE(q)$ ó $EC(q)$), que es el conjunto de estados de un AFNε a los que se puede llegar desde q por 0, 1 o más transiciones ε.

El algoritmo sigue los siguientes pasos:

1. Calcular $EC(q)$ para cada estado q del AFNε.
2. Definir una nueva función de transición. Supongamos que $q \in Q$ ý $a \in \Sigma$:

$$\delta'(q, a) = EC\ (\ \delta\ (\ EC\ (q),\ a\ ))$$

3. Determinar el nuevo conjunto de estados finales:

$$F' = \ \{\ q \in Q :\ EC(q)\ \cap\ F \neq \varnothing\ \}$$