



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 Hernández 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:

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

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 está definido como $\text{Token} = (\text{Categoria}, \text{Lexema})$. La Categoría es el nombre (o, valga de redundancia, la categoría) 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:

- $\emptyset \rightarrow$ Conjunto vacío.
- $\varepsilon \rightarrow$ Representa una cadena vacía 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 qué estado movernos dentro de M dependiendo del símbolo leído.

Tenemos 4 tipos de autómatas:

- **Autómata Finito No Determinista con Transiciones ε (AFN ε)**: 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.
- $Bexp \text{ and } Bexp \rightarrow$ 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.
- $if Bexp \text{ then } Com \text{ else } Com \rightarrow$ Estructura del if.
- $while Bexp \text{ do } Com \rightarrow$ 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)
|   | -- Enteros.pdf (Ejemplo de como pasar de ER hasta MDD)
|
|-- 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 (Módulo que ejecuta todas las pruebas de esta carpeta)
|   |-- TestAFNEp.hs (Pruebas para AFNepsilon)
|   |-- TestAFN.hs (Pruebas para AFN)
|   |-- TestAFD.hs (Pruebas para AFD)
|   |-- TestAFDmin.hs (Pruebas para AFD min)
|   |-- TestMDD.hs (Pruebas para MDD)
|   |-- TestLexer.hs (Pruebas finales del Lexer)
|
|-- package.yaml
|-- Proyecto01-Analizador-L-xico.cabal
|-- README.md (Instrucciones de ejecución)
|-- 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` y `do` como palabras reservadas. Nuestra idea es unir a todas estas palabras con el operador `+` para las ER, pues una vez que se declaró 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 así, tenemos la siguiente ER:

$$([a-z] + [A-Z])([a-z] + [A-Z] + [0-9])^*$$

Esto obliga a que nuestros identificadores siempre inicien con una letra y después 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:

$$0 + ([1-9][0-9]^*) + -([1-9][0-9]^*)$$

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:

$$+ \mid - \mid * \mid = \mid <= \mid :=$$

La implementación en haskell es análoga a las palabras reservadas:

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

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

$$; \mid (\mid)$$

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, así 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'))))
```

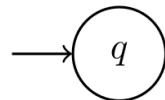
(2) ER \rightarrow AFN ε

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

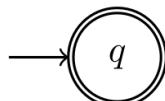
Para esta conversión, implementamos el algoritmo de Thompson. Podemos resumir muy rápido los pasos recursivos:

- **Casos Base:**

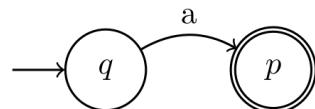
- Si la expresión regular es \emptyset , el autómata resultante será:



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

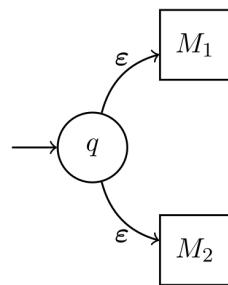


- Si la expresión regular es un símbolo a , tal que $a \in \Sigma$, el autómata resultante será:

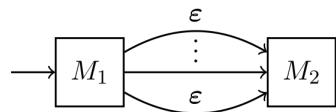


- **Casos Recursivos:** Para esta parte, supongamos que tenemos dos autómatas M_1 , M_2 que reconocen los lenguajes de las expresiones regulares l y r respectivamente.

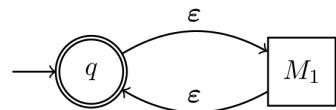
- Si la expresión regular es $l + r$, el autómata resultante será:



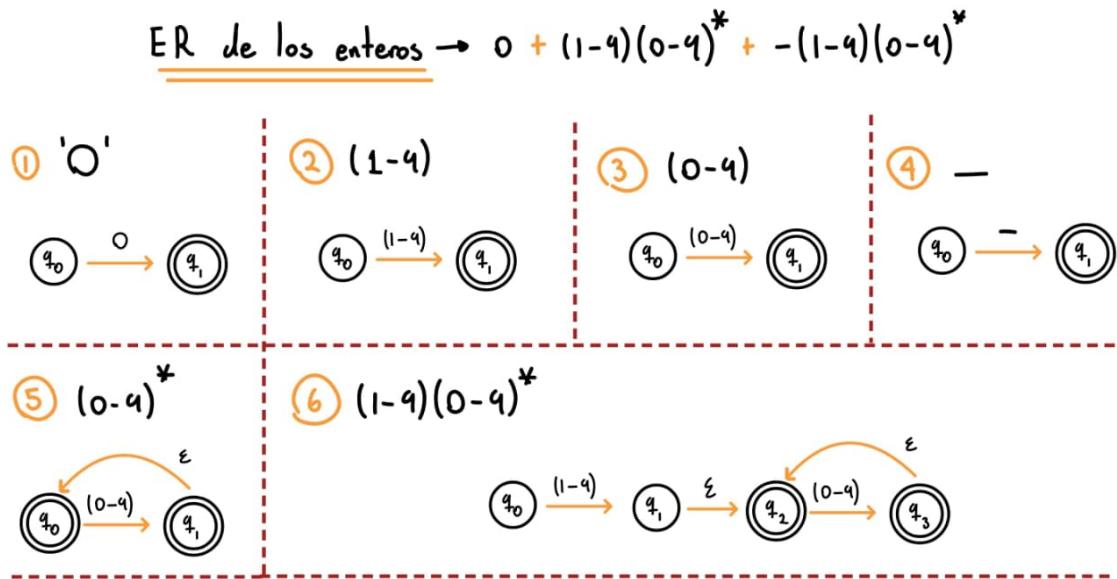
- Si la expresión regular es lr , el autómata resultante será:



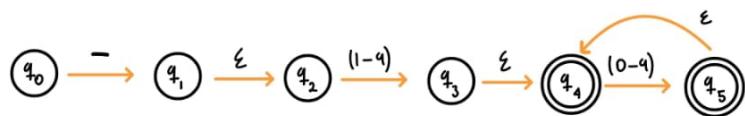
- Si la expresión regular es l^* , el autómata resultante será:



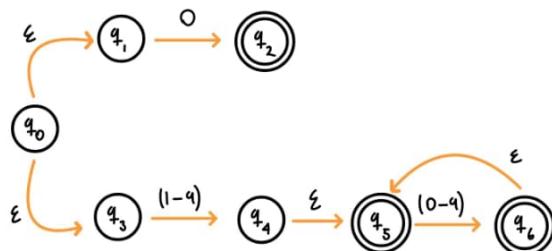
Por ejemplo, si queremos traducir nuestra expresión regular de los enteros a una AFN ϵ , nos queda de la siguiente manera:



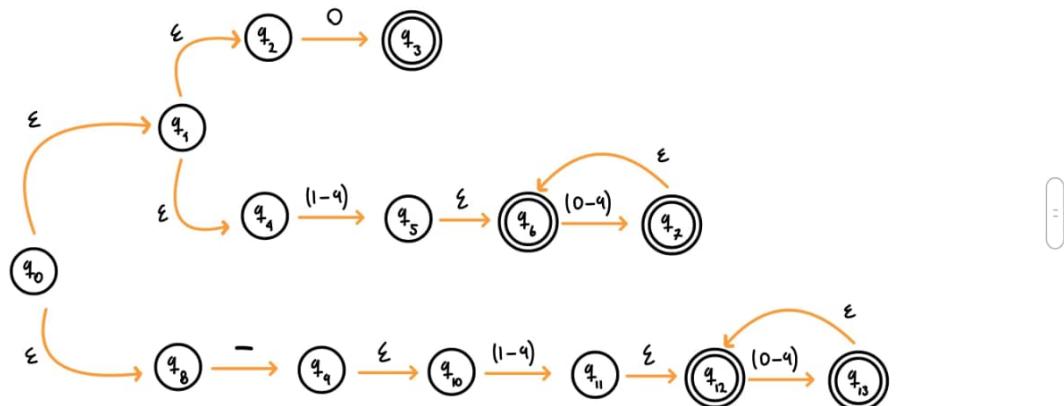
⑦ $-(1-q)(0-q)^*$



⑧ $O + (1-q)(0-q)^*$



⑨ $O + (1-q)(0-q)^* + -(1-q)(0-q)^*$



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

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

-- Transiciones en AFNε
-- 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 AFN ε

```
-- 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)

```

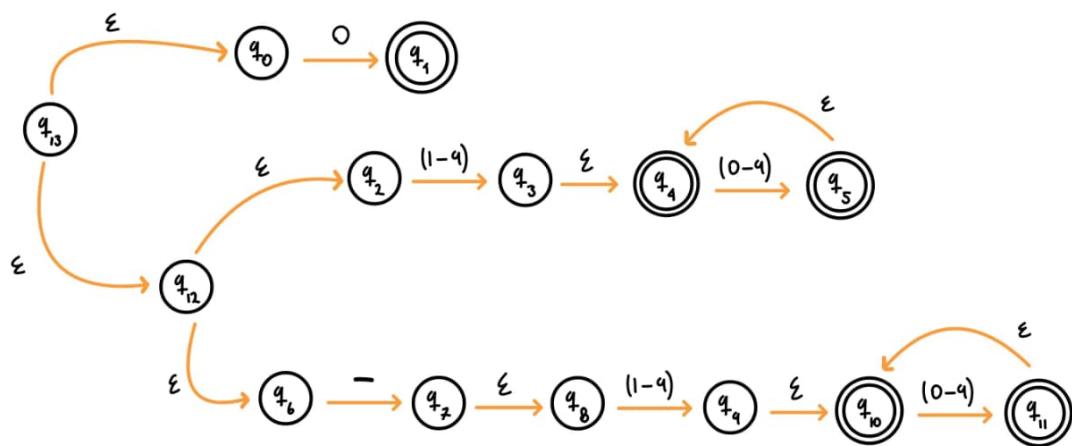
Para poder probar nuestra implementación, hicimos unos tests rápidos en el módulo `TestAFNEp.hs`. Dicho modulo, nos imprime los AFN ϵ de las siguientes ER : $0(0+1)^*$, $(ab+b)^*ab^*$ y $0 + [1-9][0-9]^* + -[1-9][0-9]^*$.

Para seguir con el ejemplo de los enteros, el AFN ϵ 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]),("q5",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 '+',[q11]),("q10",Just '-',[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",Just '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) $\text{AFN}_\varepsilon \rightarrow \text{AFN}$

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

Para esta conversión, implementamos el algoritmo basado en las ε -*cerradura* de un estado q (denotado $ECLOSURE(q)$ ó $EC(q)$), que es el conjunto de estados de un AFN_ε 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 AFN_ε .
2. Definir una nueva función de transición. Supongamos que $q \in Q$ y $a \in \Sigma$:

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

3. Determinar el nuevo conjunto de estados finales:

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

Si aplicamos esta estrategia al AFN_ε de los enteros, que obtuvimos en la sección anterior, obtendremos lo siguiente:

①

- $EC(q_0) = \{q_0, q_1, q_2, q_4, q_8\}$
- $EC(q_1) = \{q_1, q_2, q_4\}$
- $EC(q_2) = \{q_2\}$
- $EC(q_3) = \{q_3\}$
- $EC(q_4) = \{q_4\}$
- $EC(q_5) = \{q_5, q_6\}$
- $EC(q_6) = \{q_6\}$
- $EC(q_7) = \{q_6, q_7\}$
- $EC(q_8) = \{q_8\}$
- $EC(q_9) = \{q_9, q_{10}\}$
- $EC(q_{10}) = \{q_{10}\}$
- $EC(q_{11}) = \{q_{11}, q_{12}\}$
- $EC(q_{12}) = \{q_{12}\}$
- $EC(q_{13}) = \{q_{12}, q_{13}\}$

② $|q_0|$

$$\begin{aligned} \cdot \delta'(q_0, 0) &= EC(\delta(EC(q_0), 0)) = EC(\delta(\{q_0, q_1, q_2, q_4, q_8\}, 0)) \\ &= EC(q_3) = \underline{\underline{\{q_3\}}} \end{aligned}$$

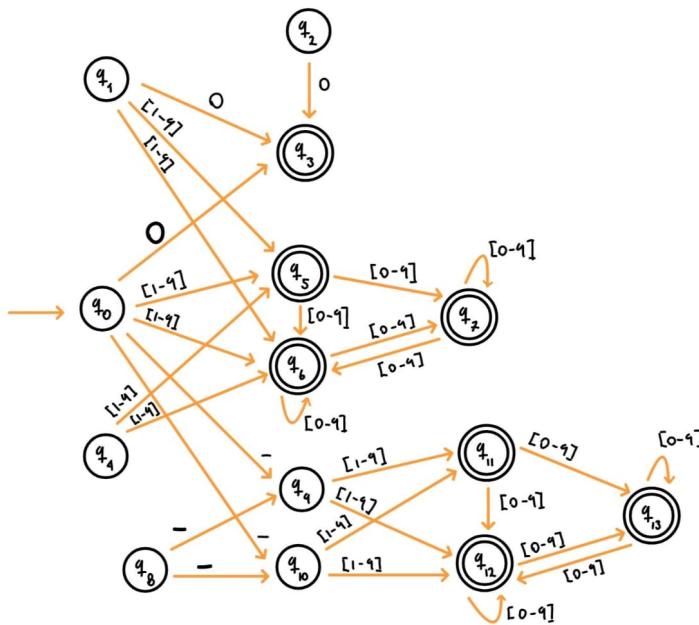
$$\begin{aligned} \cdot \delta'(q_0, [1-4]) &= EC(\delta(EC(q_0), [1-4])) = \\ EC(\delta(\{q_0, q_1, q_2, q_4, q_8\}, [1-4])) &= EC(q_5) = \underline{\underline{\{q_5, q_6\}}} \end{aligned}$$

$$\begin{aligned} \cdot \delta'(q_0, [0-4]) &= EC(\delta(EC(q_0), [0-4])) = \\ EC(\delta(\{q_0, q_1, q_2, q_4, q_8\}, [0-4])) &= EC(\emptyset) = \underline{\underline{\emptyset}} \end{aligned}$$

$$\begin{aligned} \cdot \delta'(q_0, -) &= EC(\delta(EC(q_0), -)) = EC(\delta(\{q_0, q_1, q_2, q_4, q_8\}, -)) \\ &= EC(q_9) = \underline{\underline{\{q_9, q_{10}\}}} \end{aligned}$$

Los nuevos estados finales (dado $F' = \{q \in Q : EC(q) \cap F \neq \emptyset\}$)

es $F' = \{q_3, q_5, q_6, q_7, q_{11}, q_{12}, q_{13}\}$



Antes de continuar, es importante aclarar que no hemos enseñado todos los pasos para llegar al autómata anterior. Específicamente, calculando la nueva fun-

ción de transición para todos los estados con todo el alfabeto. El procedimiento completo para pasar de $ER \rightarrow AFN\epsilon \rightarrow AFN \rightarrow AFD \rightarrow AFDmin \rightarrow MDD$ (hablando de los enteros), se encuentra en el archivo `Enteros.pdf` que esta en este mismo directorio. Por simplicidad, mostraremos solo pasos claves de estos procedimientos para que el reporte no nos quede tan grande.

Una vez aclarado esto, podemos pasar con nuestra implementación en haskell.

Lo primero que tenemos que hacer es re calcular su principal cambio respecto al módulo anterior: La función de transición. Para esto, definimos un nuevo tipo de dato AFN que es casi idéntico al AFNEp, pero con la nueva diferencia clave:

```
-- Nueva función de transición, que si o si, necesita un símbolo

-- Estado origen -> Símbolo -> Lista de estados destino
type Trans_afn = (Estado, Char, [Estado])

-- Definición de AFN
data AFN = AFN {
    estados2 :: [Estado],
    alfabeto2 :: [Char],
    transiciones2 :: [Trans_afn],
    inicial2 :: Estado,
    finales2 :: [Estado]
} deriving (Eq)
```

Además, decimos utlizar `Data.Set` para representar los conjuntos de estados de las EC . Esto lo hacemos con `import qualified Data.Set as Conj`, y lo hicimos con el fin de optimizar las operaciones de la unión, intersección y búsqueda. Y evitar un poco rompernos la cabeza.

Ahora, para implementar los tres puntos de nuestra estrategia, tenemos lo siguiente.

- (1) **Calcular EC(q):** En nuestro código, implementamos esto en la función `eclosure` mediante una búsqueda DFS:

```
eclosure :: AFNEp -> Estado -> Conj.Set Estado
eclosure m q =
    -- Iniciamos dfs con el estado q tanto en los estados por revisar,
    -- como en los estados visitados.
    --dfs [q] (Conj.singleton q)
```

```

where
  -- EstadosPorRevisar -> EstadosVisitados -> EstadosAlcanzables
  dfs :: [Estado] -> Conj.Set Estado -> Conj.Set Estado
  dfs [] visitados = visitados
  dfs (qi : estados) visitados =
    -- Vemos todos los vecinos alcanzables desde qi con epsilon.
    let vecinosAlcanzables = Conj.fromList [ qDestino |
      (qOrigen, Nothing, qDestinos) <- transiciones m, qOrigen == qi,
      qDestino <- qDestinos ]
    -- Filtramos los vecinos que no hayamos visitado.
    nuevosVecinos = Conj.difference vecinosAlcanzables visitados
    ...
  in dfs nuevosPorRevisar nuevosVisitados

```

También definimos `closure_Union`, que es una función auxiliar que aplica `closure` a cada estado de un conjunto y une todos los resultados.

- (2) Cálculo de la nueva función de transición (δ'): La fórmula $\delta'(q, a) = EC(\delta(EC(q), a))$ la traducimos casi directamente en nuestra función `transicionesAFN`.

```

transicionesAFN :: AFNEp -> [Trans_afn]
transicionesAFN m =
  [ (q, a, Conj.toList deltaFinal) -- Creamos la transición (q, a, destinos),
  | q <- estados m,           -- para cada estado q
    a <- alfabeto m,          -- y para cada símbolo 'a'.
    -- Calculamos EC(q).
    let ecQ = closure m q
    -- Calculamos d (EC(q), a).
    delta_ecQ = delta m ecQ a
    -- Calcular EC( d( EC(q), a ) ).
    deltaFinal = closure_Union m delta_ecQ
    -- Solo creamos la transición si el conjunto de destino no es vacío.
    , not (Conj.null deltaFinal) ]

```

También definimos la función `delta`, que recibe una conjunto de estados y un símbolo, para devolvernos el conjunto de estados destino con ese símbolo.

- **(3) Cálculo de los nuevos estados finales:** Implementamos esta lógica en la función `nuevosFinales`, donde verificamos si la intersección (`Conj.disjoint`) entre la `EC(q)` y los finales antiguos (`antiguosFinales`) no está vacía.

```
nuevosFinales :: AFNEp -> [Estado]
nuevosFinales m =
    let antiguosFinales = Conj.fromList (finales m)
    in [ q | q <- estados m,
           let ec_q = closure m q,
           -- A y B tienen intersección
           not (Conj.disjoint ec_q antiguosFinales) ]
```

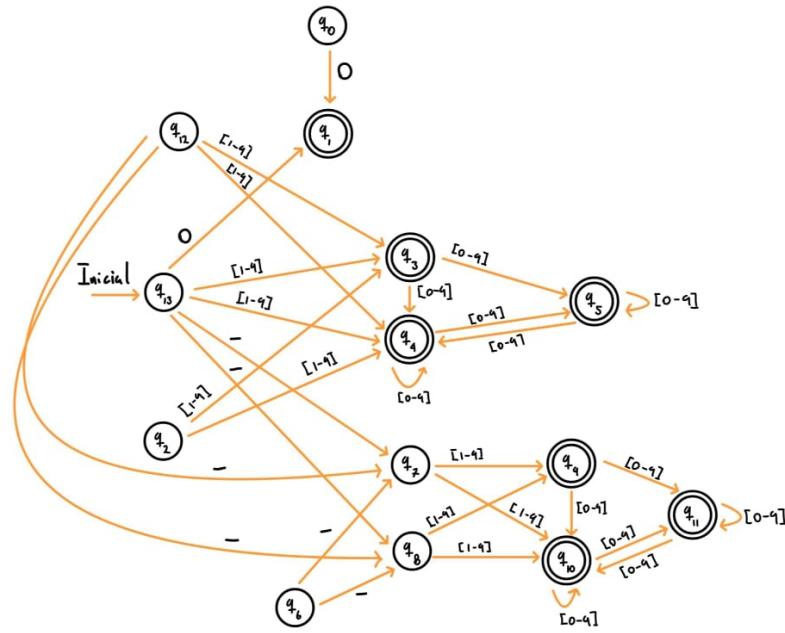
Finalmente, la función principal de este módulo es `aFNEp_to_AFN`, y construye el nuevo AFN resultante del AFN_ε . Lo que hacemos es conservar los estados, el alfabeto y el estado inicial, pero sustituir las transiciones y los estados finales por los que acabamos de calcular.

```
-- <<< Función principal >>>
aFNEp_to_AFN :: AFNEp -> AFN
aFNEp_to_AFN m = AFN {
    estados2 = estados m,
    alfabeto2 = alfabeto m,
    inicial2 = inicial m,
    transiciones2 = transicionesAFN m,
    finales2 = nuevosFinales m
}
```

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

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

Si lo dibujamos, obtenemos el siguiente AFN:



Dado el AFN ε obtenido por el programa en la sección anterior, podemos concluir que este AFN es su equivalente.

(4) AFN → AFD

El AFN que generamos en el módulo anterior no es adecuado para un analizador léxico, pues dado que desde un estado y un solo símbolo podríamos llegar a varios estados, tendríamos bastante ambigüedad y conflicto con nuestra analizador. Entonces, necesitamos un AFD que obtendremos mediante la construcción por subconjuntos. Toda esta lógica esta implementada en el módulo `AFD.hs`

El algoritmo de las notas del curso las podemos reescribir en los siguientes pasos:

1. Construcción por Subconjuntos: La idea es que cada estado en el nuevo AFD no corresponde a un solo estado del AFN, sino a un conjunto de estados del AFN. Empezamos definiendo el estado inicial, que es el conjunto que contiene únicamente al estado inicial del AFN. Es decir, $\{q_0\}$.

Partiendo de este conjunto, comenzamos calculando las nuevas transiciones entre subconjuntos. Para un conjunto de estados C del AFD y un símbolo del alfabeto s , la transición $\delta(C, s)$ se define como el conjunto de todos los estados alcanzables desde cualquier estado en S usando el símbolo a en el AFN.

Nuestra implementación no calcula todos los $2^{|Q|}$ subconjuntos del conjunto potencia. Preferimos realizar una búsqueda partiendo del estado q_0 para garantizar que solo los estados alcanzables desde aquí sean creados. Como consecuencia, eliminamos todos los estados inalcanzables en este mismo movimiento.

Por último, todos los estados finales serán los subconjuntos de estados que contengan al menos un estado final del AFN que recibimos como entrada.

2. Completar el AFD resultante: En el paso anterior, podría pasarnos que en el AFD existan estados que no tengan una transición para cada símbolo del alfabeto. Para solucionar este problema, creamos un nuevo estado sumidero para que todas las transiciones faltantes apunten a este nuevo estado. Finalmente, este estado transiciona a sí mismo con todos los símbolos del alfabeto.
3. Renombramos los estados: En el AFD resultante, podríamos tener nombres del tipo " q_n, q_m, \dots ", pues trabajamos con subconjuntos de estados. Lo que hacemos en esta fase es renombrar los estados para que solo tengan un valor, como " q_n ".

Si aplicamos esta estrategia al AFN de los enteros, que obtuvimos en la sección anterior, tendremos el siguiente procedimiento:

① Construcción por subconjuntos

Iniciamos con $\{q_0\}$

$\{q_0\}$

$$\delta(\{q_0\}, 0) = \underline{\{q_3\}}$$

$$\delta(\{q_0\}, [1-9]) = \underline{\{q_5, q_6\}}$$

$$\delta(\{q_0\}, -) = \underline{\{q_9, q_{10}\}}$$

$\{q_2, q_{13}\}$

$$\delta(\{q_{12}, q_{13}\}, [0-9]) = \underline{\{q_{12}, q_{13}\}}$$

$$\{q_3\} \rightarrow \emptyset$$

$\{q_5, q_6\}$

$$\delta(\{q_5, q_6\}, [0-9]) = \underline{\{q_6, q_7\}}$$

$\{q_9, q_{10}\}$

$$\delta(\{q_9, q_{10}\}, [1-9]) = \underline{\{q_{11}, q_{12}\}}$$

$\{q_6, q_7\}$

$$\delta(\{q_6, q_7\}, [0-9]) =$$

$\{q_6, q_7\}$

$\{q_{11}, q_{12}\}$

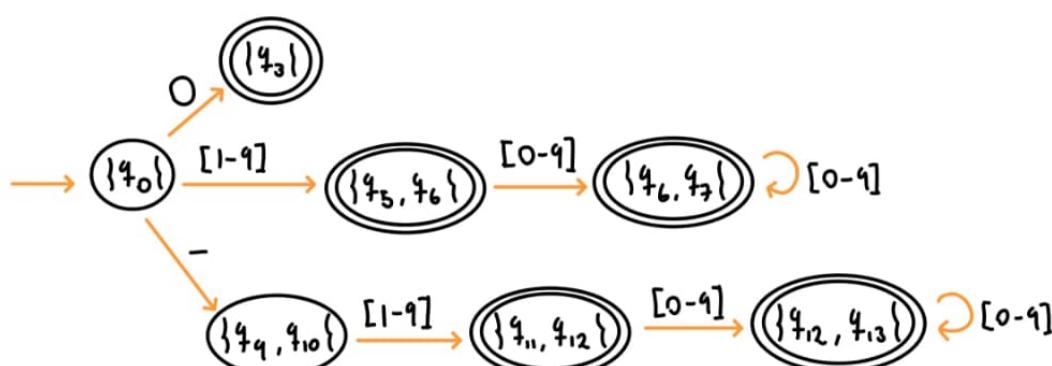
$$\delta(\{q_{11}, q_{12}\}, [0-9]) =$$

$\{q_{12}, q_{13}\}$

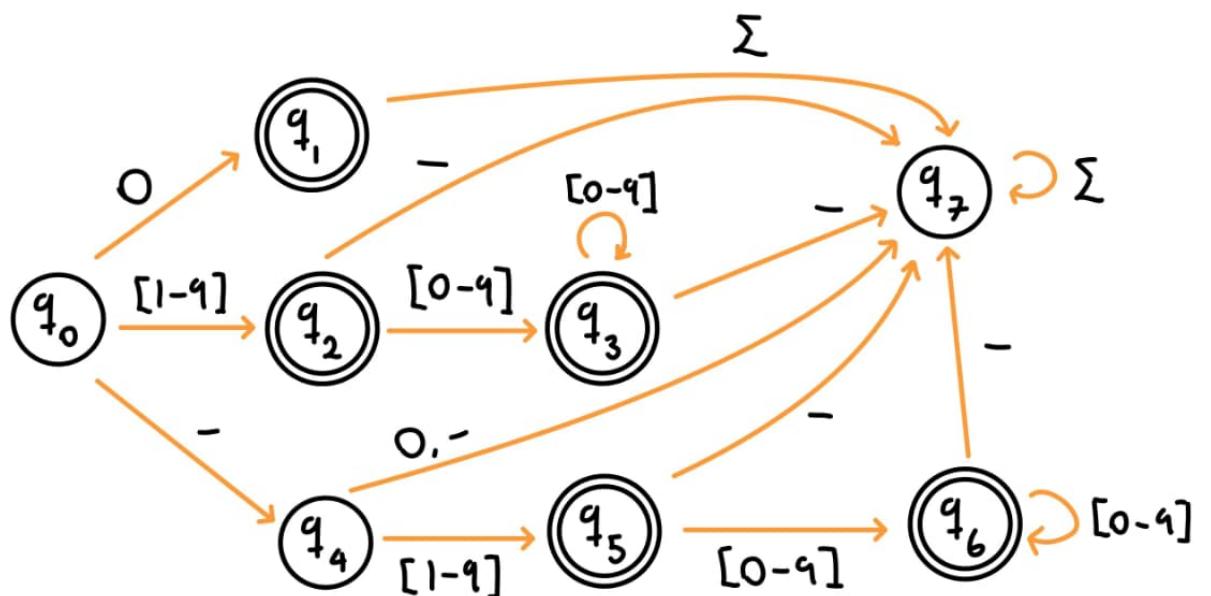
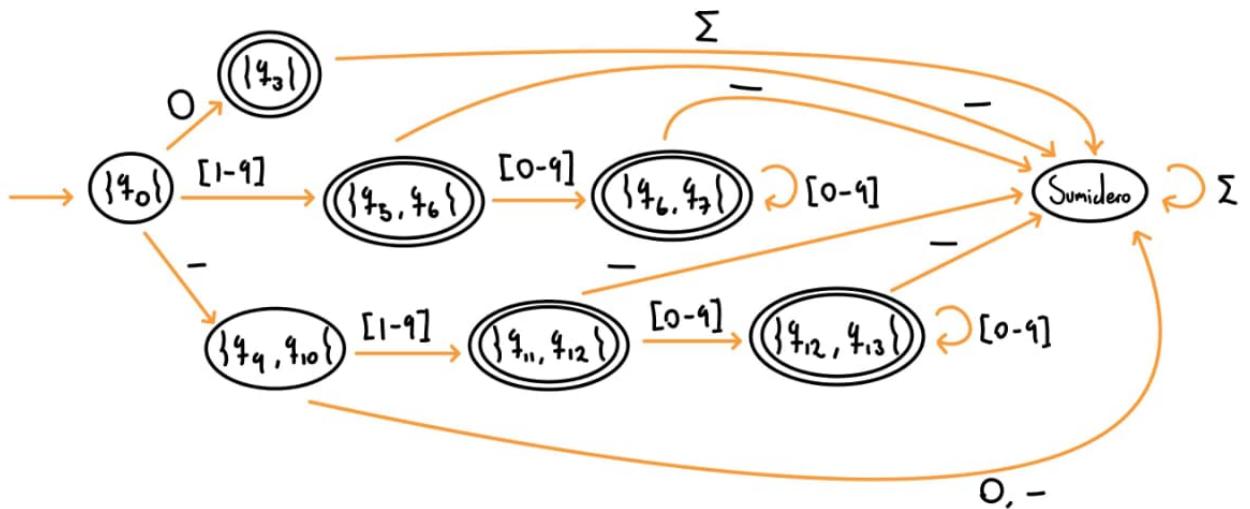
Nuevos estados: $\{q_0\}, \{q_3\}, \{q_5, q_6\}, \{q_9, q_{10}\}, \{q_6, q_7\}, \{q_{11}, q_{12}\}, \{q_{12}, q_{13}\}$

Estados finales: $\{q_3\}, \{q_5, q_6\}, \{q_6, q_7\}, \{q_{11}, q_{12}\}, \{q_{12}, q_{13}\}$

Autómata resultante



② Completar Autómata



Pasando con nuestra implementación en haskell, lo primero que tenemos que hacer es definir sus transiciones, que solamente van de un estado a otro con un símbolo.

```
-- Estado origen -> Símbolo -> Estado destino.
type Trans_afd = (Estado, Char, Estado)
```

Nuestro algoritmo se basa en la idea de que un estado en el nuevo AFD es un conjunto de estados del AFN original. Para manejar esto, creamos la función **renombra** que toma una lista de estados, la ordena y la une en un único Estado:

```
renombra :: [Estado] -> Estado
renombra = intercalate "," . sort
```

La siguiente función es **alcanzables**, e implementa la función de transición del nuevo AFD. Dado una lista de estados C (simulando nuestro conjunto) y un símbolo a , busca en todas las transiciones del AFN que inician en el estado q_i ($q_i \in C$) con a , recolecta todos los estados destinos y los devuelve como una nueva lista de estados.

```
alcanzables :: AFN -> [Estado] -> Char -> [Estado]
alcanzables afn conj a =
    -- Recolecta todos los estados destino.
    nub [ destino
        -- Iteramos sobre cada transición del AFN.
        | (q, c', ds) <- transiciones2 afn,
          -- La transición debe originarse en un estado de ''conj''.
          q `elem` conj ,
          -- La transición debe usar el símbolo ''a''
          c' == a ,
          -- Finalmente, iteramos sobre la lista de destinos 'ds'
          destino <- ds ]
```

La siguiente función es **descubrirEstados** realiza una búsqueda de estados partiendo del estado inicial. Lo que hace es mantener una lista de estados pendientes y de estados visitados. Solo los estados que son alcanzables desde el inicial se añaden a la lista de pendientes y estos eventualmente terminaran en la lista de visitados. Con esto garantizamos que cualquier estado inalcanzable nunca sea descubierto.

```

descubrirEstados :: AFN -> [([Estado], Estado)]
descubrirEstados afn =
    -- El estado inicial del AFD es el conjunto {q0} del AFN
    let conjuntoInicial = [inicial2 afn]
        nombre = renombra conjuntoInicial
    -- Llamamos a la función 'procesa', que es una función recursiva,
    -- de la búsqueda
    in procesa afn [(conjuntoInicial, nombre)] []

procesa :: AFN -> [([Estado], Estado)] -> [([Estado], Estado)]
                                         -> [([Estado], Estado)]
-- Caso base -> No hay más pendientes. Devolvemos la lista de visitados.
procesa _ [] visitados = visitados
procesa afn ((conj, nombre):pendientes) visitados
    -- Si ya visitamos este estado, lo saltamos.
    | nombre `elem` map snd visitados = procesa afn pendientes visitados
    | otherwise =
        -- Calculamos todas sus transiciones salientes
        let nuevos = [ (alcanzables afn conj c, renombra (alcanzables afn conj c))
                      | c <- alfabeto2 afn , not (null (alcanzables afn conj c))]
        -- recursión.
        in procesa afn (pendientes ++ nuevos) (visitados ++ [(conj, nombre)])

```

La siguiente función es `calcularFinales`, que itera sobre los estados descubiertos y lo define como estado final si cualquiera de los estados que componen al conjunto era un estado final en el AFN original.

```

calcularFinales :: AFN -> [([Estado], Estado)] -> [Estado]
calcularFinales afn todosEstados =
    [ estadoF
    | (conjunto, estadoF) <- todosEstados ,
      any ('elem' finales2 afn) conjunto ]

```

La siguiente función es `completarAFD`. Inicia buscando todas las transiciones faltantes para guardarlas en una lista. Si esta lista está vacía entonces el AFD está completo, pero si no, se crea el estado "Sumidero" y se añaden las transiciones faltantes apuntando a él. Finalmente, de este estado hacemos un ciclo de transiciones con todos los símbolos del alfabeto.

```

completarAFD :: AFD -> AFD
completarAFD afd =
    let estados = estadosD afd
        alfabeto = alfabetoD afd
        transiciones = transicionesD afd
        estadoSumidero = "Sumidero"

        -- Encontrar todas las transiciones faltantes.
        transicionesFaltantes = [ (q, c, estadoSumidero)
            | q <- estados, c <- alfabeto
            , not (existeTrans transiciones q c) ]

        -- Ver si necesitamos añadir el estado sumidero.
        (estadosNuevos, transicionesNuevas)
            -- Si no faltan transiciones, pues no hacemos nada.
            | null transicionesFaltantes = (estados, transiciones)
            | otherwise =
                -- Si faltan, añadimos las transiciones faltantes.
                let transError = [ (estadoSumidero, c, estadoSumidero)
                    | c <- alfabeto ]
                    in (estados ++ [estadoSumidero],
                        transiciones ++ transicionesFaltantes ++ transError)

        -- Devolvemos el AFD completo.
        in afd { estadosD = estadosNuevos, transicionesD = transicionesNuevas }
    
```

La última función lógica, es `renombrarAFD` para limpiar los nombres. Lo que hacemos es crear una lista de pares (`NombreAntiguo`, `NombreNuevo`), y para empezar, asegurarnos de que el estado inicial se nombre como q_0 . Los demás estados los nombramos con el formato q_i . Finalmente, reconstruimos los estados y sus transiciones con los nuevos nombres.

La función principal del módulo es `aFN_to_AFD` simplemente conecta toda nuestra lógica. Primero construimos los subconjuntos y sus transiciones, luego completamos el AFD y finalmente renombramos los estados.

```

aFN_to_AFD :: AFN -> AFD
aFN_to_AFD afn =
    -- Construcción por subconjuntos.
    let todosEstados = descubrirEstados afn

```

```
-- Todos los calculos...
listaEstados = map snd todosEstados

afdIntermedio = AFD { estadosD = listaEstados, ... }

-- Completar el AFD.
let afdCompleto = completarAFD afdIntermedio

-- Renombrar estados.
let afdFinal = reanombrarAFD afdCompleto

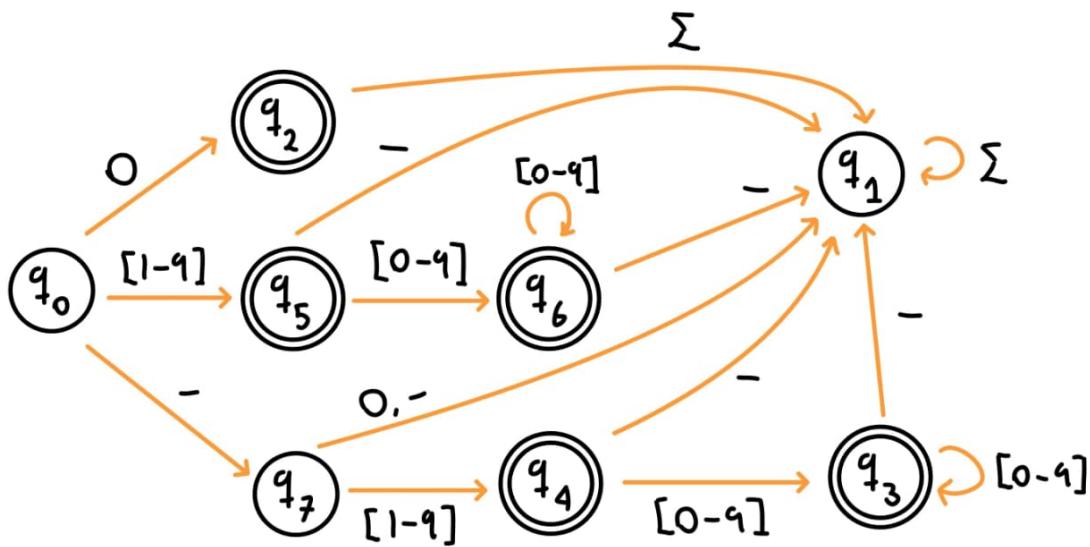
-- Devolvemos el resultado final.
in afdFinal
```

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

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

```
--> Prueba 3 : ' 0 + [1-9][0-9]^* + -[1-9][0-9]^* ' <-----
***** Imprimiendo AFD *****
>> estados      = ["q0","q1","q2","q3","q4","q5","q6","q7"]
>> alfabeto     = "0123456789-"
>> transiciones = [{"q0": "0", "q2"}, {"q0": "1", "q5"}, {"q0": "2", "q5"}, {"q0": "3", "q5"}, {"q0": "4", "q5"}, {"q0": "5", "q5"}, {"q0": "6", "q5"}, {"q0": "7", "q5"}, {"q0": "-", "q7"}, {"q1": "0", "q0"}, {"q1": "2", "q6"}, {"q1": "3", "q6"}, {"q1": "4", "q6"}, {"q1": "5", "q6"}, {"q1": "6", "q6"}, {"q1": "7", "q6"}, {"q1": "-", "q7"}, {"q2": "0", "q0"}, {"q2": "1", "q5"}, {"q2": "2", "q6"}, {"q2": "3", "q6"}, {"q2": "4", "q6"}, {"q2": "5", "q6"}, {"q2": "6", "q6"}, {"q2": "7", "q6"}, {"q2": "-", "q7"}, {"q3": "0", "q0"}, {"q3": "1", "q5"}, {"q3": "2", "q6"}, {"q3": "3", "q6"}, {"q3": "4", "q6"}, {"q3": "5", "q6"}, {"q3": "6", "q6"}, {"q3": "7", "q6"}, {"q3": "-", "q7"}, {"q4": "0", "q0"}, {"q4": "1", "q5"}, {"q4": "2", "q6"}, {"q4": "3", "q6"}, {"q4": "4", "q6"}, {"q4": "5", "q6"}, {"q4": "6", "q6"}, {"q4": "7", "q6"}, {"q4": "-", "q7"}, {"q5": "0", "q0"}, {"q5": "1", "q5"}, {"q5": "2", "q6"}, {"q5": "3", "q6"}, {"q5": "4", "q6"}, {"q5": "5", "q6"}, {"q5": "6", "q6"}, {"q5": "7", "q6"}, {"q5": "-", "q7"}, {"q6": "0", "q0"}, {"q6": "1", "q5"}, {"q6": "2", "q6"}, {"q6": "3", "q6"}, {"q6": "4", "q6"}, {"q6": "5", "q6"}, {"q6": "6", "q6"}, {"q6": "7", "q6"}, {"q6": "-", "q7"}, {"q7": "0", "q0"}, {"q7": "1", "q5"}, {"q7": "2", "q6"}, {"q7": "3", "q6"}, {"q7": "4", "q6"}, {"q7": "5", "q6"}, {"q7": "6", "q6"}, {"q7": "7", "q6"}, {"q7": "-", "q7"}]
>> inicial      = "q0"
>> finales     = ["q2","q3","q4","q5","q6"]
Listo :D
```

Si dibujamos este AFD, obtenemos:



Tanto el AFD obtenido por el programa y que obtuvimos con nuestro procedimiento a mano son equivalentes. Podemos concluir que el procedimiento es correcto.

(5) AFD → AFDmin

En la fase anterior obtuvimos un AFD completo y funcional, aunque no necesariamente óptimo, pues es probable que contenga estados redundantes o equivalentes. Para poder optimizar nuestro AFD necesitamos colapsar todos estos estados equivalentes para producir un AFD equivalente, pero que sea mínimo. Esta lógica esta implementada en el módulo `AFDmin.hs`.

La estrategia que implementamos se basa en el algoritmo de llenado de tabla que esta en las notas del curso:

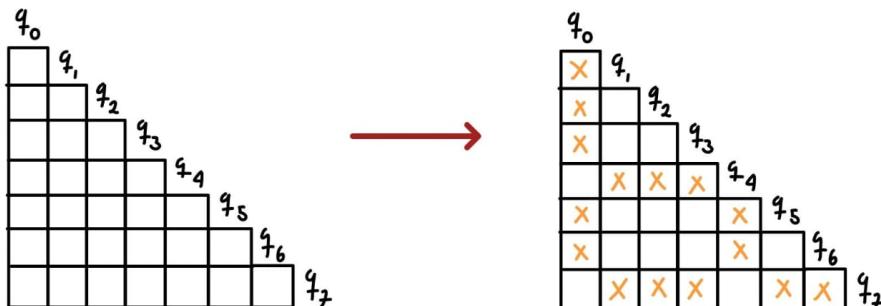
1. Creamos una tabla con todos los pares de estados $\{p, q\}$. Luego, marcamos un par como "distinguible" si uno de ellos es un estado final y el otro no.
2. Repetimos el siguiente paso hasta que no haya más cambios en la tabla: Marcamos un par $\{p, q\}$ (que aún no esté marcado) si existe algún símbolo a del alfabeto tal que al transicionar con a , el par resultante $\{p', q'\}$ ya está marcado como distingüible en la tabla.

Para aclararlo, dos estados p y q son distingüibles si existe al menos una cadena c tal que al procesarla, uno de ellos termina en un estado final y el otro no.

3. Una vez que ya no hay cambios en la tabla, cualquier par $\{p, q\}$ que no esté marcado se considera equivalente. Entonces, nuestro paso final es tomar todos los pares equivalentes, agruparlos en conjuntos y construir un nuevo AFD donde cada uno de estos conjuntos se convierte en un único estado.

Si aplicamos esta estrategia al AFD de los enteros, que obtuvimos en la sección anterior, tendremos el siguiente procedimiento:

① Tabla de pares de estados



② Repetición de paso 2.

 $\{q_0, q_4\}$

$$\begin{array}{l|l|l|l} \delta(q_0, 0) = q_1 & \delta(q_0, [1-q]) = q_2 & \delta(q_0, -) = q_4 & \{q_0, q_4\} \\ \delta(q_4, 0) = q_7 & \delta(q_4, [1-q]) = q_5 & \delta(q_4, -) = q_7 & \{q_4, q_7\} \\ \uparrow & & & \\ \text{Si está marcado} & & & \end{array}$$

 $\therefore \text{Marcamos } \{q_0, q_4\}$

Nos brincamos pasos hasta la tabla final...

La tabla final:

q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
X							
X	X						
X	X						
X	X	X					
X	X	X	X				
X	X	X	X	X			
X	X	X	X	X	X		
X	X	X	X	X	X	X	

③ Fusionar estados equivalentes

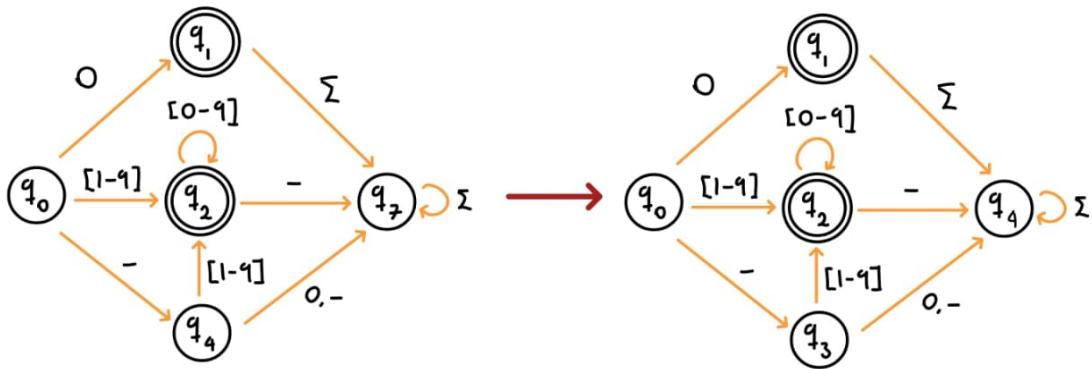
Tenemos lo siguiente

$$q_2 \approx q_3, q_2 \approx q_5, q_2 \approx q_6, q_3 \approx q_5, q_3 \approx q_6$$

$$\text{y } q_5 \approx q_6$$

$$\Rightarrow q_2 \approx q_3 \approx q_5 \approx q_6$$

Colapsamos estos estados y sus transiciones:



Para poder hacer nuestra implementación en haskell de la forma más fiel posible a esta estrategia, vamos a apoyarnos en las estructuras de datos **Map** y **Set**.

La razón de esto viene de una anedocta rápida. En uno de nuestros cursos de modelado y programación (donde aprendimos haskell por primera vez), jugamos justamente con la estructura **Map** para "mapear" un tesoro. Básicamente hicimos un **Map** con coordenadas e íbamos marcando los lugares por lo que queríamos pasar y resultaba bastante rápido en complejidad. Se nos ocurrió aplicar esta misma lógica aquí.

Lo primero que hicimos fue definir un nuevo dato **Par**, que es la representación de dos estados (p, q) :

```
-- Par de estados (p, q) donde p < q.
type Par = (Estado, Estado)
```

Luego, creamos otros dos tipos de datos: **TablaPares** y **MapaDelta**. **TablaPares** es la implementación de nuestra tabla de pares (p, q) donde $p < q$, y la forma en la que marcaremos las "casillas" de nuestra "tabla" será con valores booleanos. Todas las casillas están marcadas con **False** desde el inicio, y los iremos marcando con **True** si ese par de estados son distinguibles. **MapaDelta** es una especie de memoria cache para nuestras transiciones y no tener que buscar en la lista de transiciones cada vez, y así poder realizar búsquedas aun más rápidas.

```
-- Tabla de pares entre estados.
type TablaPares = Mapa.Map Par Bool

-- Memoria cache de transiciones.
type MapaDelta = Mapa.Map (Estado, Char) Estado
```

La función principal de este módulo es `minimizaAFD`, y empezamos definiendo la tabla de pares de estados:

```
-- Creamos la lista de todos los pares (p, q).
todosPares = [ (q1, q2) | q1 <- estados, q2 <- estados, q1 < q2 ]

-- Creamos la tabla inicial donde marcamos como true a todos
-- los pares que sean distinguibles.
tablaInicial = Mapa.fromList [ (p, esMarcadoInicial p) | p <- todosPares ]
  where
    esFinal q = q `Conj.member` finales
    -- Lo marcamos si un estado es final y el otro no.
    esMarcadoInicial (p, q) = esFinal p /= esFinal q
```

El siguiente paso será ir marcando la tabla hasta que ya no cambie. Para esto, tenemos la función `encontrarPunto` que recalcula la tabla entera basándose en la tabla anterior y se detiene solo cuando `tablaNueva == tablaActual` (es decir, no hubo cambios).

```
debeMarcarse :: TablaPares -> MapaDelta -> Par -> Char -> Bool
debeMarcarse tablaActual mapaDelta (p, q) a =
  -- Checar a dónde van 'p' y 'q' con el símbolo 'a'
  let deltaP = mapaDelta Mapa.! (p, a)
      deltaQ = mapaDelta Mapa.! (q, a)
  -- Ahora vemos si el par resultante ya está marcado
  in esParMarcado tablaActual (deltaP, deltaQ)
```

```
logicaDeMarcado :: TablaPares -> MapaDelta -> [Char] -> Par -> Bool -> Bool
logicaDeMarcado tablaActual mapaDelta alfabeto (p, q) esMarcado
  -- Si ya está marcado como 'True', pues así se queda.
  | esMarcado = True
  -- Si no está marcado, comprobamos si debe marcarse.
  | otherwise =
    -- Devolvemos True si cualquier símbolo del alfabeto
    -- fuerza que este par se marque.
    any (debeMarcarse tablaActual mapaDelta (p, q)) alfabeto
```

Una vez que tenemos la tabla final, tenemos que filtrar todos los pares que quedaron en `False`, que son nuestros pares equivalentes. La función que hace esto

es `encontrarParticiones`, y trata a los estados equivalentes como una grafica (construyendo un mapa de adyacencia) y usamos DFS para encontrar todos los componentes conectados". Cada componente conectado es exactamente una clase de equivalencia, es decir, una partición.

```

encontrarParticiones :: [Estado] -> [Par] -> [Conj.Set Estado]
encontrarParticiones estados paresEquivalentes =
    let
        -- Cramos el mapa de adyacencia.
        mapaA = Map.fromListWith (++) $
            concat [ [(p, [q]), (q, [p])] | (p, q) <- paresEquivalentes ]
        -- Iniciamos la búsqueda DFS para encontrar componentes conectados.
        in buscar mapaA estados Conj.empty
    
```

La parte final es construir el nuevo autómata a partir de las particiones. Para poder probar nuestra implementación, hicimos unos tests rápidos en el módulo `TestAFDmin.hs`. Dicho modulo, nos imprime los AFDmin resultantes de los AFD de las ER : $0(0+1)^*$, $(ab+b)^*ab^*$ y $0 + [1-9][0-9]^* + -[1-9][0-9]^*$.

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

```

----> Prueba 3 : ' 0 + [1-9][0-9]^* + -[1-9][0-9]^* ' <-----
-----  

***** Imprimiendo AFD *****  

>> estados      = ["q0","q1","q2","q3","q4"]  

>> alfabeto     = "0123456789-*"  

>> transiciones = [{"q0": "0", "q2"}, {"q0": "1", "q3"}, {"q0": "2", "q3"}, {"q0": "3", "q3"}, {"q0": "4", "q3"}, {"q0": "5", "q3"}, {"q0": "6", "q3"}, {"q0": "7", "q3"}, {"q0": "8", "q3"}, {"q0": "9", "q3"}, {"q0": "-", "q4"}, {"q1": "0", "q1"}, {"q1": "1", "q1"}, {"q1": "2", "q1"}, {"q1": "3", "q1"}, {"q1": "4", "q1"}, {"q1": "5", "q1"}, {"q1": "6", "q1"}, {"q1": "7", "q1"}, {"q1": "8", "q1"}, {"q1": "9", "q1"}, {"q1": "*", "q1"}, {"q2": "0", "q1"}, {"q2": "1", "q1"}, {"q2": "2", "q1"}, {"q2": "3", "q1"}, {"q2": "4", "q1"}, {"q2": "5", "q1"}, {"q2": "6", "q1"}, {"q2": "7", "q1"}, {"q2": "8", "q1"}, {"q2": "9", "q1"}, {"q2": "-", "q1"}, {"q3": "0", "q3"}, {"q3": "1", "q3"}, {"q3": "2", "q3"}, {"q3": "3", "q3"}, {"q3": "4", "q3"}, {"q3": "5", "q3"}, {"q3": "6", "q3"}, {"q3": "7", "q3"}, {"q3": "8", "q3"}, {"q3": "9", "q3"}, {"q3": "*", "q3"}, {"q4": "0", "q4"}, {"q4": "1", "q4"}, {"q4": "2", "q4"}, {"q4": "3", "q4"}, {"q4": "4", "q4"}, {"q4": "5", "q4"}, {"q4": "6", "q4"}, {"q4": "7", "q4"}, {"q4": "8", "q4"}, {"q4": "9", "q4"}, {"q4": "-", "q1"}]  

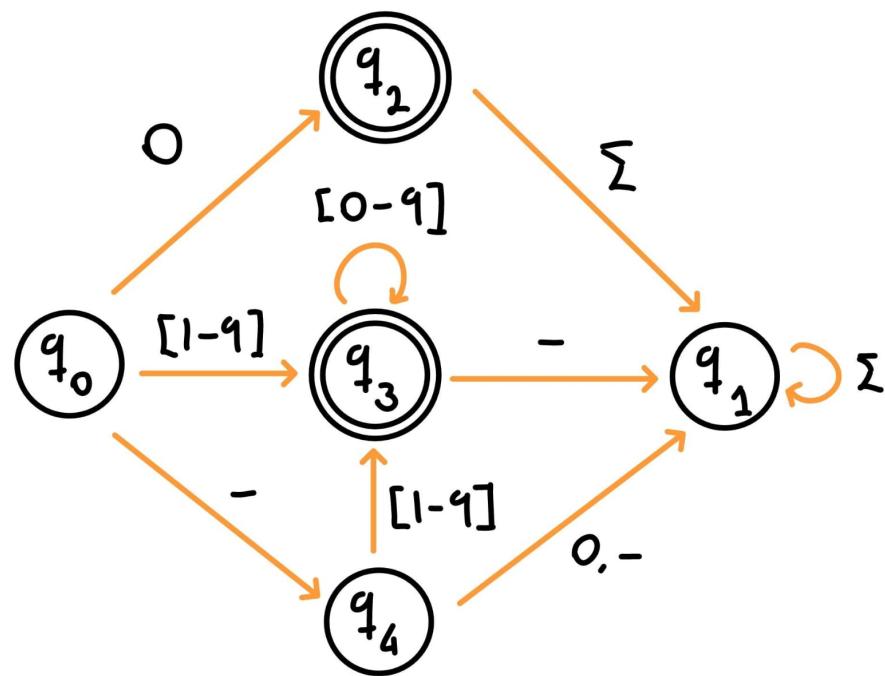
>> inicial       = "q0"  

>> finales      = ["q2", "q3"]  

Listo :D  

-----
```

Si lo dibujamos, obtenemos el siguiente AFD min:



(6) AFDmin → MDD

Un analizador léxico necesita responder a la siguiente pregunta: ¿A qué categoría de token pertenece esta cadena?. Aquí es donde entra nuestra Máquina Discriminadora Determinista (MDD).

Podemos decir que un MDD es un AFD enriquecido con una función que asigna categorías léxicas a los estados de aceptación. Su definición formal es $M = \{Q, \Sigma, \delta, q_0, F, \mu, \perp\}$. Los primero 5 elementos son los correspondientes al AFD, y el componente crucial es la función μ que mapea cada estado final a una categoría léxica.

Entonces, en lugar de que el autómata acepte una cadena en un estado final, ahora las clasifica basándose en la etiqueta de ese estado final.

En teoría (y como marcan la mayoría de recursos del curso) tendríamos que hacer una MDD grande juntando todas las AFD en una sola. En un principio lo diseñamos así, pero a la hora de ejecutar el proyecto final nunca terminaba de analizar. Para optimizar esta estrategia, buscamos en varios recursos estrategias alternativas hasta que la encontramos en el libro *Engineering a Compiler* de Keith D. Cooper y Linda Torczon. Específicamente, en el capítulo 2 *Scanners* se define (aunque no de manera explícita) una estrategia para construir muchas MDD pequeñas y especializadas para cada categoría de token de nuestra gramática.

Por ejemplo, el proceso para poder ejecutar la idea anterior con los enteros es tomar su ER y traducirla a un AFDmin. Luego, necesitamos enriquecer (con la función μ) este AFD de los enteros para crear la MDD de los enteros, donde todos los estados finales del AFD se mapean como ".^{Enteros}". Esto lo tenemos que hacer para cada categoría (o ER) de nuestro lenguaje IMP, para así tener una lista de MDDs especializadas en cada categoría.

El trabajo del lexer, que explicamos en la siguiente sección, será ejecutar todas estas MDD y ver cuál encuentra la mejor coincidencia.

Para implementar toda esta lógica en haskell (en el módulo `MDD.hs`), primero tenemos que definir las estructuras de datos y la principal lógica de nuestro analizador. Lo primero fue definir la salida de nuestro lexer. Un Token puede ser un token con categoría y lexema, una directiva para Omitir o un Error:

```
type Categoría = String
type Lexema = String
```

```
-- Definimos los tipos de salida del lexer.
data Token
    = Token Categoria Lexema
    | Omitir
    | Error Char
deriving (Show, Eq)
```

Luego, definimos la MDD siguiendo la definición formal, que es un AFD junto con la función μ , que implementamos como un **Mapa** para asociar estados finales a su categoría.

```
-- Definición de MDD.
data MDD = MDD {
    afd :: AFD,
    mu :: Mapa.Map Estado Categoria
} deriving (Show)
```

Para gestionar nuestra estrategia de muchas MDD, creamos un **MDDInfo**. Esto agrupa cada MDD con su categoría correspondiente y una **Prioridad**. Esta prioridad es crucial para resolver las ambigüedades en nuestro analizador, pues si el analizador lee la cadena "*if*" puede ser asociada con los identificadores o las palabras reservadas.

```
type Prioridad = Int

-- Agrupa una MDD cn su prioridad y categoría.
data MDDInfo = MDDInfo {
    prioridad :: Prioridad,
    categoria :: Categoria,
    mdd :: MDD
} deriving (Show)
```

Con estas estructuras definidas, la implementación la dividimos en dos funciones principales: **simular** y **lexer**.

La función **simular** toma una MDD y una cadena de entrada, para encontrar la coincidencia más larga que esa MDD puede reconocer desde el inicio de la cadena.

```

simular :: MDD -> String -> Maybe (Lexema, Estado)
simular mdd input =
    let
        deltaMap = Mapa.fromList [ ((q, c), dest) |
                                    (q, c, dest) <- transicionesD (afd mdd) ]
        q0 = inicialD (afd mdd)
        muMap = mu mdd

        -- Función recursiva.
        ciclo :: Estado -> String -> String -> Maybe (Lexema, Estado)
                                         -> Maybe (Lexema, Estado)
        ciclo q consumido restante mejorMatchHastaAhora =
            let
                esFinalActual = Mapa.member q muMap
                -- Si estamos en un estado final, actualizamos el mejor match.
                nuevoMejorMatch = if esFinalActual && length consumido >=
                    maybe 0 (length . fst) mejorMatchHastaAhora
                    then Just (consumido, q)
                    else mejorMatchHastaAhora
            in
                case restante of
                    [] -> nuevoMejorMatch -- Ya se acabó la entrada.
                    (c:cs) ->
                        case Mapa.lookup (q, c) deltaMap of
                            Just q_siguiente -> ciclo q_siguiente
                                (consumido ++ [c]) cs nuevoMejorMatch
                            Nothing -> nuevoMejorMatch
    in ciclo q0 "" input Nothing

```

La función `lexer` maneja todas las MDD, que recibe la lista de todas nuestras `MDDInfo` y la cadena de entrada e intenta simular todas las MDD en la entrada actual. Si ninguna tiene éxito, consume el carácter actual como un *Error* y continúa. Si hay éxitos, primero filtra por la longitud máxima y de los que empataron en longitud, escoge el que tenga la prioridad más alta. Finalmente, emitimos el Token (o lo omitimos si la categoría es "omitar") y repetimos este proceso con el resto de la cadena.

Para ver que nuestra implementación es correcta, podemos ejecutar los tests del archivo `TestMDD.hs`.

(7) Lexer y Main

Nuestro módulo `Lexer.hs` es "la mamá de los pollitos", pues conecta todos los módulos que definimos anteriormente y tiene como función principal definir las reglas léxicas con sus prioridades y construir la lista de MDDs especializadas.

Iniciamos definiendo un tipo de dato `GramaticaLexica` como una lista de tuplas (Categoria, Prioridad, Expr). Es importante aclarar que mientras menor sea el valor del número de la prioridad, es mayor la prioridad que se le da (esperamos explicarnos bien). Con este dato, ya podemos definir nuestro lenguaje IMP:

```
type GramaticaLexica = [(Categoria, Prioridad, Expr)]
type Prioridad = Int

gramaticaIMP :: GramaticaLexica
gramaticaIMP =
  [ ("PALABRA_RESERVADA", 1, palabrasReservadas)
  , ("IDENTIFICADOR", 2, identificador)
  , ("ENTERO", 3, enteros)
  , ("OPERADOR", 4, operadores)
  , ("DELIMITADOR", 5, delimitadores)
  , ("omitir", 10, espacios)
  , ("omitir", 10, comentarios)
  ]
```

Decidimos dejar a las palabras reservadas con una prioridad mayor a los identificadores, para evitar ambigüedades.

La función `construirMDDIndividual` encapsula toda la lógica desde una ER hasta su MDD:

```
construirMDDIndividual :: Categoria -> Expr -> MDD
construirMDDIndividual categoria expr =
  let afnEps  = compilarAFNEp expr          -- ER -> AFNEp
      afn    = aFNEp_to_AFN afnEps           -- AFNEp -> AFN
      afd    = aFN_to_AFD afn                -- AFN -> AFD
      afdMin = minimizaAFD afd              -- AFD -> AFD min
      -- Construcción de la función mu.
      muSimple = Map.fromList [ (estadoFinal, categoria) |
                                estadoFinal <- finalesD afdMin ]
  in MDD afdMin muSimple
```

Por último, la función `construirListaMDDs` aplica este proceso a cada elemento de nuestra `gramaticaIMP`, generando la lista de `MDDInfo` lista para ser consumida por el lexer.

En nuestro siguiente (y último) módulo `Main.hs`, tiene la responsabilidad de integrar la configuración del lexer para procesar un archivo real.

Al iniciarse el programa, se llama a `construirListaMDDs` usando la `gramaticaIMP`. Esto compila todas las ER en sus respectivos AFDmin antes de leer cualquier archivo. Luego, el programa busca e intenta leer el archivo `implementacion.imp`, que es el que contiene el "código" por analizar. Si el archivo no existiera, el programa se carga con un código de ejemplo predeterminado. Dependiendo de cual fue el caso, se invoca a la función `lexer` pasando la lista de autómatas y el código fuente. Cuendo termina, el lexer devuelve una lista de tokens encontrados.

Por ejemplo, si el contenido del archivo `implementacion.imp` es

```
x := 10;  
y := 5;
```

Entonces el resultado es el siguiente:

```
=====
|| Construyendo Analizador Léxico para IMP ||
=====

>>> Construyendo Lista de MDDs en base a la gramática...
>>> ¡Listo! Leyendo la implementación a probar ...
>>> Archivo leido: main/implementacion.imp

>>> Contenido leído:

x := 10;
y := 5;

>>> Analizando entrada...

>>>> Tokens reconocidos <<<
Token "IDENTIFICADOR" "x"
Token "OPERADOR" ":="
Token "ENTERO" "10"
Token "DELIMITADOR" ";"
Token "IDENTIFICADOR" "y"
Token "OPERADOR" ":="
Token "ENTERO" "5"
Token "DELIMITADOR" ";"

>>> ¡Análisis léxico completado!

=====
|| Adios ||
=====
```

El contenido del archivo `implementacion.imp` se puede modificar al antojo del usuario, aunque se recomienda que uno se apegue a las reglas del lenguaje IMP para no tener mayores dificultades.

Para probar que este último paso lo hemos hecho bien, solo queda ejecutar las pruebas del módulo `TestLexer.hs`.

En el archivo `README.md` esta todo especificado, pero aquí va un resumen rápido de ejecución de las pruebas y del programa:

Para probar el programa con tu implementación en el archivo `implementacion.imp`, solo nos posicionamos en la carpeta "raiz" del proyecto y ejecutamos `stack run`. Para correr las pruebas, nos ponemos en la misma posición y solo ejecutamos `stack test`.

Conclusiones

Con el desarrollo de este proyecto hemos podido comprender y realizar un puente entre la parte teórica y práctica entre los lenguajes formales y la construcción de compiladores. Podemos decir que hemos logrado implementar un analizador léxico funcional para el lenguaje IMP.

Consideramos que nuestra implementación desde una ER hasta su MDD es una estrategia eficaz, que ademas facilitó la depuración de cada etapa con las pruebas creadas para poder saber si estabamos tomando un buen camino y garantizar que en cada etapa se seguía aceptando el mismo lenguaje.

La construcción de la lista de MDD en vez de hacer una sola (y gigante) MDD resultó ser bastante crucial para nuestro proyecto, pues optimizó el rendimiento del analizador y nos simplificó la lógica de resolución de conflictos. Ademas, nos resultó bastante útil declarar una prioridad numérica entre categorías léxicas, pues así evitamos ambigüedades entre los identificadores y las palabras reservadas.

Pero al final, nos sentimos satisfechos (a excepción de la fecha de entrega :() del analizador léxico resultante. Nos gusta que es capaz de procesar archivos de "código real". Pidemos decir que este proyecto sienta una buena base para las siguientes fases del compilador, como el análisis sintáctico (que nos imaginamos que debe ser un verdadero dolor de cabeza a la hora de implementar...).

Bibliografia

- Notas del curso. Tanto las proporsionadas por el profesor, tanto las hechas por nosotros.
- Cooper y Torczon, *Engineering a Compiler*, 2nd ed., 2011.
- Winskel, *The Formal Semantics of Programming Languages: An Introduction*, 1993.