Universidad Nacional Autónoma de México

Facultad de Ciencias

Compiladores

Proyecto 01: Construcción de un Analizador Léxico

Equipo:

Cabrera Sánchez Ana Dhamar

López Prado Emiliano

Peña Mata Juan Luis

Rodríguez Miranda Alexia

Rosas Franco Diego Angel

Profesor:

Manuel Soto Romero

Ayudantes:

Diego Méndez Medina

José Alejandro Pérez Márquez

Ayudantes Laboratorio:

Jose Manuel Evangelista Tiburcio

Fausto David Hernández Jasso

Fecha de entrega:

19 de Octubre 2025

1. Introducción

A lo largo de la historia de la computación, el desarrollo de los compiladores ha estado estrechamente ligado al avance de las teorías formales del lenguaje y la lógica. El punto de partida puede rastrearse en los trabajos de Noam Chomsky, quien formalizó la teoría de gramáticas y estableció las bases para clasificar los lenguajes según su complejidad sintáctica. Paralelamente, los aportes de Alan Turing con sus máquinas de Turing y de Alonzo Church con el cálculo lambda sentaron los fundamentos de la computabilidad y la semántica formal, conceptos que hoy permiten a los compiladores realizar transformaciones seguras y correctas sobre programas y expresiones.

Antes de la computación moderna, los matemáticos ya buscaban automatizar procesos lógicos y demostraciones, lo que eventualmente condujo al surgimiento de los primeros lenguajes de programación. Entre ellos, FORTRAN destacó por incorporar uno de los primeros mecanismos de optimización automática en compilación, marcando un punto clave en la evolución de los compiladores hacia las herramientas que conocemos hoy.

En este contexto, el presente proyecto tiene como objetivo comprender y construir la primera etapa de un compilador: el análisis léxico. Para ello, se implementará desde cero esta fase utilizando el lenguaje Haskell, capaz de expresar formalismos de manera declarativa y segura. El analizador léxico desarrollado podrá adaptarse a diferentes especificaciones de lenguaje; en nuestro caso, trabajaremos con una variante simplificada del lenguaje IMP.

2. Marco Teórico

Para comprender adecuadamente este proyecto, se presentarán a continuación algunas definiciones formales que servirán de base teórica para el desarrollo posterior.

2.1. Lenguaje

[1] Formalmente, un ***lenguaje*** sobre un alfabeto Σ es un subconjunto de Σ^*, y se denota comúnmente con una letra mayúscula, $A, B, C, \ldots, Y, Z$. Todo lenguaje L satisface $\emptyset \subseteq L \subseteq \Sigma^*$.

Dado que los lenguajes son conjuntos, las operaciones usuales entre conjuntos son operaciones válidas entre lenguajes. Por lo cual, si R, S son lenguajes definidos sobre un mismo alfabeto Σ, es decir, $R, S \subseteq \Sigma^*$, entonces los siguientes conjuntos también son lenguajes sobre Σ:

$R \cup S$	Unión
$R \cap S$	Intersección
$R - S$	Diferencia
$\overline{R} = \Sigma^* - R$	Complemento

Nombramos las operaciones anteriores como ***operaciones de conjuntos*** u ***operaciones booleanas*** para distinguirlas de las ***operaciones lingüísticas***.

Las ***operaciones lingüísticas*** son extensiones de las operaciones entre cadenas en un lenguaje. Se nombran y definen de la siguiente manera:

Dados dos lenguajes R, S definidos sobre Σ:

- **Concatenación.** La concatenación, denotada $R \cdot S$, o simplemente RS, se define como:

$$RS = \{\, rs \mid r \in R, s \in S \,\}$$

- **Potencia.** Denotada como R^n, donde n es un número natural, se define recursivamente como:

$$R^0 = \{\varepsilon\},$$
$$R^n = R^{n-1} \cdot R = \{\, r_1 r_2 \cdots r_n \mid r_i \in R,\ 1 \leq i \leq n \,\}$$

  Es decir, R^n es el conjunto de todas las concatenaciones de n cadenas de R en todas las formas posibles.

- **Cerradura de Kleene.** Se define como la unión de todas las potencias de R y se denota por R^*

$$R^* = \bigcup_{i \geq 0} R^i = R^0 \cup R^1 \cup \cdots \cup R^n$$

- **Cerradura positiva.** De manera similar, se define como el conjunto de las potencias positivas de un lenguaje, es decir:

$$R^+ = \bigcup_{i \geq 1} R^i = R^1 \cup R^2 \cup \cdots \cup R^n$$

Los lenguajes se clasifican en distintas categorías. Para la construcción de un analizador léxico, nos enfocamos principalmente en los ***lenguajes regulares*** que son todos los lenguajes que pueden construirse a partir de los lenguajes básicos $\emptyset, \{\varepsilon\}, \{a\},\ (a \in \Sigma)$ usando las ***operaciones regulares:*** unión, concatenación y cerradura de Kleene.

2.2. Expresiones Regulares

[2] Una ***expresión regular*** se define recursivamente sobre un alfabeto Σ de la siguiente manera:

- $\emptyset$ es una expresión regular y denota el conjunto vacío.
- ε es una expresión regular y denota el conjunto que contiene a la cadena vacía, $\{\varepsilon\}$.
- Para cada símbolo $a \in \Sigma$, a es una expresión regular y denota el conjunto que contiene la cadena unitaria $\{a\}$.
- Si r y s son expresiones regulares que denotan los lenguajes R y S respectivamente, entonces:
  - $(r + s)$ es una expresión regular que denota el conjunto $R \cup S$.
  - (rs) es una expresión regular que denota el conjunto RS.
  - (r^*) es una expresión regular que denota el conjunto R^*.

En esta definición, los paréntesis se utilizan como símbolos de agrupación y la asociatividad de los operadores en las expresiones regulares se establece bajo la siguiente jerarquía de prioridad:

1. Cerradura de Kleene *: asocia a la izquierda.
2. Concatenación: asocia a la derecha.
3. Unión $+$: asocia a la derecha.

Las expresiones regulares se utilizan para describir de manera compacta y formal los lenguajes regulares. Para una expresión regular R, el lenguaje que reconoce la expresión es:

$$L(R) = \{\ w \mid w \in \Sigma^*,\ \text{y, } w \text{ es reconocido por la expresión regular } R\ \}$$

Por lo cual, la definición de expresiones regulares se puede simplificar a la información en la siguiente tabla:

Expresión Regular	**Lenguaje**
$\emptyset$	$\emptyset$
ε	$\{\varepsilon\}$
a	$\{a\}$
(r^*)	R^*
(rs)	RS
$(r+s)$	$R \cup S$

2.3. Autómata

[1] Un ***autómata*** es una máquina abstracta con capacidad de computar una cadena de entrada y determinar si dicha cadena pertenece a un lenguaje dado. Como salida, el autómata indica si la cadena es aceptada o rechazada.

Formalmente, se define como una $5-tupla\ M = (Q, \Sigma, \delta, q_0, F)$, donde:

- Q es un conjunto finito de estados
- Σ es un conjunto de símbolos que representan el alfabeto del autómata. Todas las cadenas que procesa M pertenecen a Σ^*.
- δ es la función de transición que se utiliza para especificar el estado al que se mueve el autómata luego de leer un símbolo de entrada.
- q_0 es el estado inicial, $q_0 \in Q$.
- F es el conjunto, no vacío, de estados finales que determinan la aceptación de una cadena procesada por el autómata, $F \subseteq Q$.

Los autómatas poseen distintas propiedades que permiten clasificarlos en varias categorías. En la construcción de un analizador léxico, destacan las siguientes:

- **AFN$-\varepsilon$**

  Un ***autómata finito no determinista con ε-transiciones (AFN$-\varepsilon$)***, es un autómata definido como $M = (Q, \Sigma, \delta, q_0, F)$, donde la función de transición δ toma la definición:

  $$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$$

  La transición distinguida $\delta(q, \varepsilon) = \{q_{i1}, \ldots, q_{in}\}$, se denomina ***transición*** ε y significa que al encontrarse en el estado q, el autómata puede moverse a cualquier estado del conjunto $T_\varepsilon = \{q_{i1}, \ldots, q_{in}\}$ sin leer un símbolo de entrada.

- **AFN**

  Un ***autómata finito no determinista (AFN)*** es un autómata $M = (Q, \Sigma, \delta, q_0, F)$, en el que la función de transición δ no permite transiciones ε, es decir:

  $$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$

  Un AFN puede ser visto como un $AFN - \varepsilon$ en el que hay *cero* transiciones ε, o como un autómata que necesita leer un símbolo de entrada para poder moverse de un estado a otro.

- **AFD**

  Un ***autómata finito determinista (AFD)*** es un autómata $M = (Q, \Sigma, \delta, q_0, F)$, en el que, para cada estado $q \in Q$ y cada símbolo $a \in \Sigma$, existe un único estado $q' \in Q$ al cual moverse luego de leer el símbolo de entrada a, es decir:

  $$\delta : Q \times \Sigma \to Q$$

  Notemos que el procesamiento de cualquier cadena en un AFD sigue un único camino de estados desde el estado inicial q_0 hasta el estado en el que se acepta o rechaza.

2.4. Equivalencia computacional de Autómatas Finitos

[2] En la *Teoría de Autómatas y Lenguajes Formales* se muestra la equivalencia computacional entre los modelos de autómatas finitos:

$$AFN - \varepsilon \equiv AFN \equiv AFD$$

Esto significa que, para cada autómata definido en cualquiera de los tres modelos, es posible construir autómatas equivalentes en los otros dos modelos. Como resultado de lo anterior tenemos que los modelos $AFN - \varepsilon, AFN, AFD$ aceptan exactamente los mismos lenguajes. El Teorema de Kleene, indica que estos lenguajes son precisamente los lenguajes regulares.

Teorema de Kleene. *Un lenguaje es regular sí y sólo sí es aceptado por un autómata finito.*

Aunque las demostraciones formales de la equivalencia computacional de los autómatas finitos y el Teorema de Kleene quedan fuera del alcance de este artículo, se presenta un procedimiento general para transformar una expresión regular en un autómata finito determinista.

Dicho procedimiento se divide en 3 etapas principales:

1. Convertir una expresión regular en un $AFN - \varepsilon$.
2. Transformar un $AFN - \varepsilon$ en un AFN.
3. Transformar un AFN en un AFD.

Etapa 1: Conversión de una expresión regular en un $AFN - \varepsilon$.

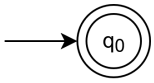
En esta etapa utilizaremos el ***Algoritmo de Thompson***, un proceso recursivo que define los casos necesarios para interpretar cada expresión regular como un autómata finito no determinista con transiciones ε.

Los autómatas que corresponden a los casos base de expresiones regulares se construyen directamente, mientras que los autómatas para casos recursivos (unión, concatenación y cerradura de Kleene) se obtienen tras construir los autómatas de sus subexpresiones.

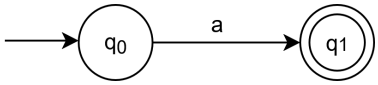
A continuación, se describen los distintos casos considerados por el algoritmo.

Caso 1. Autómata que acepta el lenguaje $L = \emptyset$.
El autómata acepta el lenguaje representado por la expresión regular $R = \emptyset$.

Caso 2. Autómata que acepta el lenguaje $L = \{\varepsilon\}$.
El autómata acepta el lenguaje representado por la expresión regular $R = \varepsilon$.

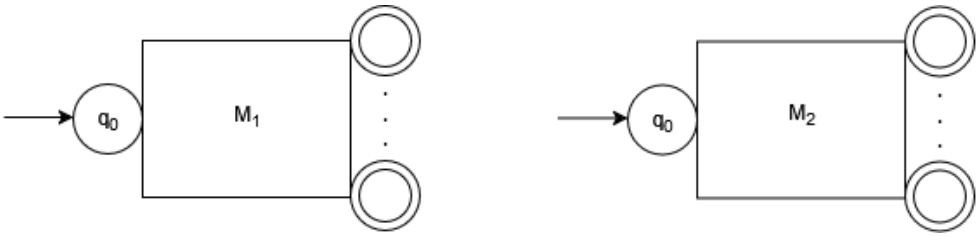


Caso 3. Autómata que acepta el lenguaje $L = \{a\},\ a \in \Sigma$.
El autómata acepta el lenguaje representado por la expresión regular $R = a$.



En los siguientes casos, suponemos que para las expresiones regulares R y S, existen M_1 y M_2, dos $AFN - \varepsilon$, tales que $L(M_1) = R$ y $L(M_2) = S$.

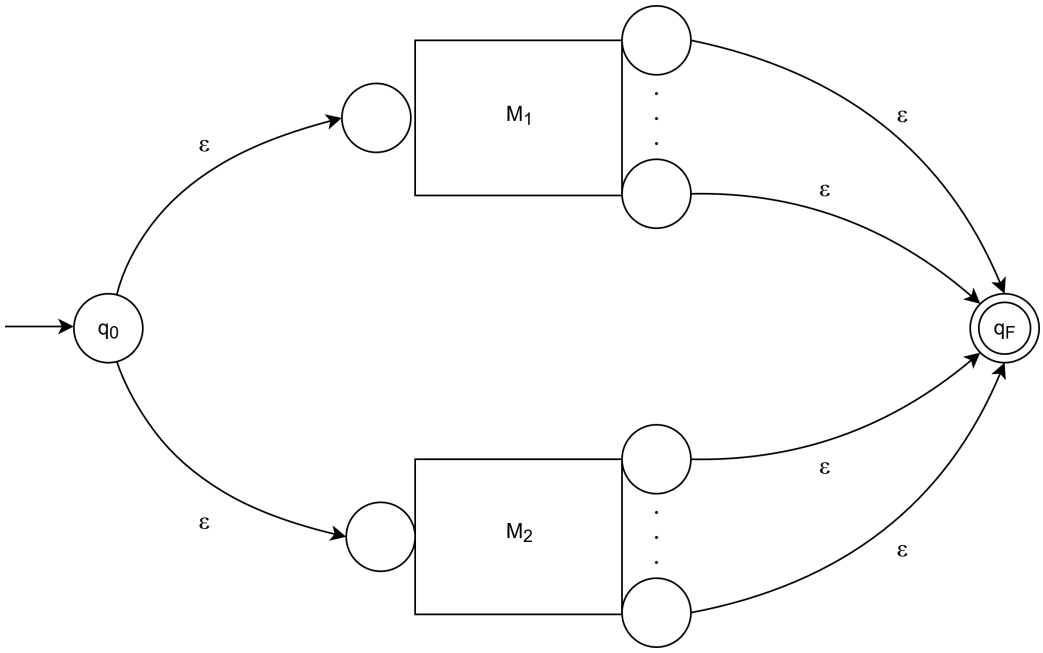
La representación de dichos autómatas se muestra en el siguiente diagrama:



Observación: El estado inicial de cada autómata se define localmente; por ello, el estado q_0 de M_1 no necesariamente es el mismo que el estado q_0 de M_2.

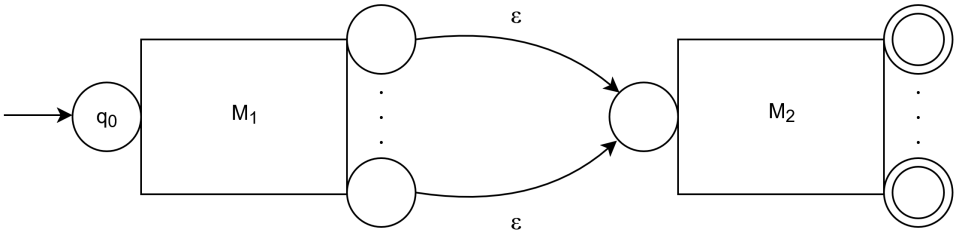
Teniendo en cuenta estas especificaciones, presentamos los casos posibles para transformar una expresión regular en un $AFN - \varepsilon$.

Caso 4. Autómata que acepta el Lenguaje $L = R \cup S$.
Se agrega un nuevo estado inicial que se conecta mediante transiciones ε a los estados iniciales de M_1 y M_2. De la misma forma, el conjunto de estados finales de cada autómata se conecta con transiciones ε a un nuevo estado final, q_F.

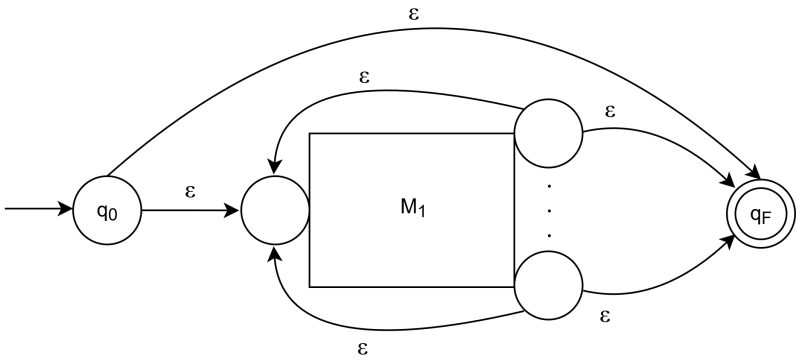


Caso 5. Autómata que acepta el Lenguaje $L = RS$.
Los estados finales del autómata M_1 se conectan mediante transiciones ε al estado inicial de M_2, por lo cual, dejan de ser estados finales y el estado inicial de M_2 deja de ser inicial.

El estado inicial del autómata se define como el estado inicial de M_1, y el conjunto de estados finales es definido como el conjunto de estados finales de M_2.



Caso 6. Autómata que acepta el Lenguaje $L = R^*$.
Los estados finales del autómata M_1 se conectan mediante transiciones ε a un nuevo estado final, q_F, y, al estado inicial de M_1. Además, se define un nuevo estado inicial que se conecta una vez más mediante transiciones ε al estado inicial de M_1 y al nuevo estado final q_F.



Etapa 2: Transformación de un $AFN - \varepsilon$ en un AFN.

Para esta etapa construiremos un AFN a partir de un $AFN - \epsilon$ mediante el uso de $\epsilon - closure$. Donde calculamos los estados a los que podemos llegar a partir de 0 o más transiciones ϵ desde de un estado q, simulando así las trayectorias obtenidas con ϵ transiciones, hay que notar que $q \in \epsilon - closure$ de inicio.

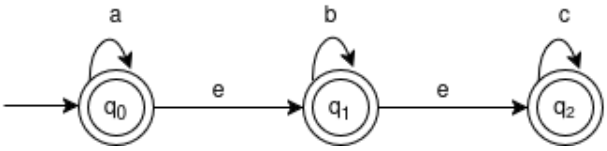
Así $\epsilon - closure$ se define como:

$$ECLOSURE(\{q1, ..., qk\}) = ECLOSURE(q1) \cup ... \cup ECLOSURE(qk).$$

Ahora definiremos δ' la función de transición para nuestro AFN.

$$\delta' : Q \times \Sigma \to P(Q)$$
$$\delta'(q, a) = ECLOSURE(\delta(ECLOSURE(q), a))$$

Esto nos dice que dado un estado $q \in Q$ y un símbolo $a \in \Sigma$, determinamos los estados alcanzables desde q primero encontrando todos los alcanzables con ϵ, $ECLOSURE(q)$, una vez encontrados estos vemos a donde llegan con el símbolo a, $\delta(ECLOSURE(q), a)$, y finalmente completamos con los estados alcanzables desde estos últimos ($ECLOSURE(\delta(ECLOSURE(q), a))$), así determinamos la trayectoria desde q con a.
Veamos el siguiente autómata:



Su respectivo $ECLOSURE$ es:

- $ECLOSURE(q_0) = \{q_0, q_1, q_2\}$
- $ECLOSURE(q_1) = \{q_1, q_2\}$
- $ECLOSURE(q_2) = \{q_2\}$

Notemos que aunque no existiera un lazo, el estado mismo q_i pertenece a $ECLOSURE(q_i)$, pues llegamos con 0 transiciones.

Además observemos que tenemos que aplicar δ' a cada estado y cada elemento del alfabeto, por lo que su complejidad es de $n \times k$, siendo n la el número de estados y k la cantidad de elementos de alfabeto, así δ' es la siguiente:

$$\begin{aligned}
\delta'(q_0, a) &= ECLOSURE(\delta(ECLOSURE(q_0), a)) \\
&= ECLOSURE(\delta(\{q_0, q_1, q_2\}, a)) \\
&= ECLOSURE(\{q_0\}) = \{q_0, q_1, q_2\} \\
\delta'(q_0, b) &= ECLOSURE(\delta(ECLOSURE(q_0), b)) \\
&= ECLOSURE(\delta(\{q_0, q_1, q_2\}, b)) \\
&= ECLOSURE(\{q_1\}) = \{q_1, q_2\} \\
\delta'(q_0, c) &= ECLOSURE(\delta(ECLOSURE(q_0), c)) \\
&= ECLOSURE(\delta(\{q_0, q_1, q_2\}, c)) \\
&= ECLOSURE(\{q_2\}) = \{q_2\} \\
\delta'(q_1, a) &= ECLOSURE(\delta(ECLOSURE(q_1), a)) \\
&= ECLOSURE(\delta(\{\}, a)) = ECLOSURE(\{\}) = \emptyset
\end{aligned}$$

$$\begin{aligned}\delta'(q_1, b) &= ECLOSURE(\delta(ECLOSURE(q_1), b)) \\ &= ECLOSURE(\delta(\{q_1, q_2\}, b)) \\ &= ECLOSURE(\{q_1\}) = \{q_1, q_2\}\end{aligned}$$

$$\begin{aligned}\delta'(q_1, c) &= ECLOSURE(\delta(ECLOSURE(q_1), c)) \\ &= ECLOSURE(\delta(\{q_1, q_2\}, c)) \\ &= ECLOSURE(\{q_2\}) = \{q_2\}\end{aligned}$$

$$\begin{aligned}\delta'(q_2, a) &= ECLOSURE(\delta(ECLOSURE(q_2), a)) \\ &= ECLOSURE(\delta(\{q_2\}, a)) = ECLOSURE(\{\}) = \emptyset\end{aligned}$$

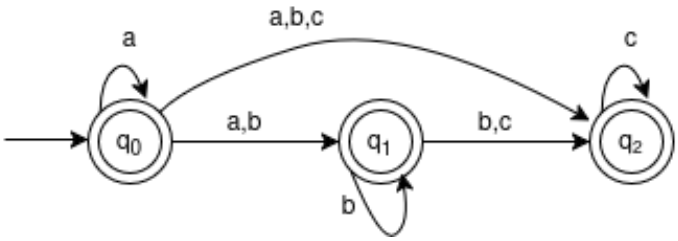
$$\begin{aligned}\delta'(q_2, b) &= ECLOSURE(\delta(ECLOSURE(q_2), b)) \\ &= ECLOSURE(\delta(\{q_2\}, b)) = ECLOSURE(\{\}) = \emptyset\end{aligned}$$

$$\begin{aligned}\delta'(q_2, c) &= ECLOSURE(\delta(ECLOSURE(q_2), c)) \\ &= ECLOSURE(\delta(\{q_2\}, c)) \\ &= ECLOSURE(\{q_2\}) = \{q_2\}\end{aligned}$$

Finalmente, los estados finales corresponden a todos aquellos que contengan un estado final de la original AFNEp, que en este caso serían todos, pues desde el inicio cada estado del AFNEp es final. Formalmente:

$$F' = \{q \in Q : ECLOSURE(q) \text{ contiene al menos un estado de aceptación}\}$$

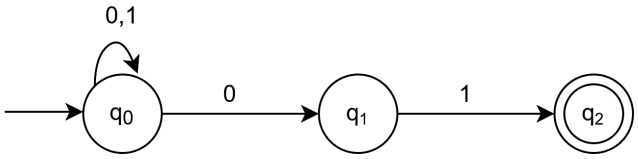
Así nuestro AFN se ve de la siguiente forma:



Etapa 3: Transformación de un AFN en un AFD.

Durante esta etapa, utilizamos un algoritmo de construcción de subconjuntos para lograr convertir nuestro autómata finito no determinista a uno determinista. Si bien, lo más común es que se utilice el algoritmo basado en construir una tabla usando todos los posibles subconjuntos del conjunto potencia, la realidad es que existe un algoritmo más eficiente que nos facilita mucho más el trabajo, se describe a continuación.

Dado un autómata finito no determinista,



empezaremos a construir una tabla donde, por un lado, agregaremos un conjunto de estados del autómata y por el otro, un conjunto de estados a los que podemos llegar utilizando todos los posibles símbolos de transición del alfabeto afín al autómata.

Aquí es donde difiere un poco nuestro algoritmo, pues para esta variante lo que haremos será agregar a la tabla **únicamente** el estado inicial, a partir de ahí, seguiremos construyendo la tabla.

	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$

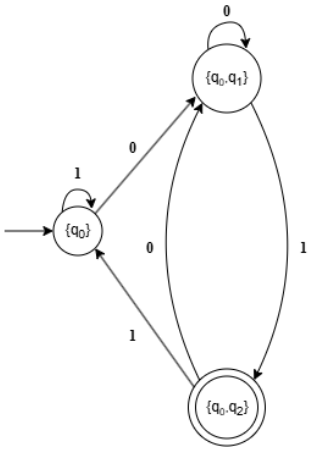
Ahora bien, el siguiente paso será agregar los conjuntos descubiertos (alcanzables) utilizando el estado inicial q_0 a la misma tabla como estados de entrada. Cada que encontremos un nuevo conjunto de estados alcanzables, los agregaremos a la tabla. Repitiendo este proceso hasta que ya no podamos agregar más estados nuevos a la tabla.

		0	1
$\rightarrow$	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
F	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
	$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

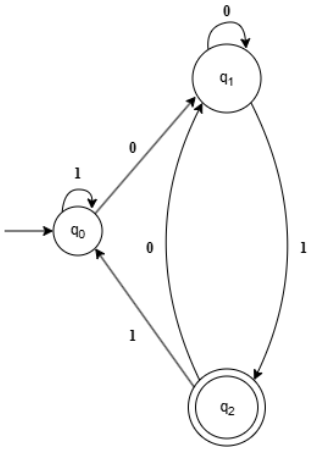
Además de llenar la tabla, tenemos que recordar que todos aquellos conjuntos de estados que contengan por lo menos un estado final del autómata serán considerados estados finales en el autómata finito determinista.

A diferencia del algoritmo original que contempla todos los posibles estados del conjunto potencia, y después descarta todos aquellos inalcanzables, aquí solo tenemos que llenar la tabla recursivamente iniciando sobre el estado inicial y terminamos cuando no haya más estados nuevos que procesar, y nos evitamos procesar estados que ni siquiera van a llegar al autómata final.

El siguiente paso es construir el autómata finito determinista a partir de la tabla, pues ahora consideraremos a cada conjunto de estados como un solo estado, como se muestra en la siguiente ilustración.



Finalmente, y por pura simpleza, podemos cambiar el nombre de los conjuntos a algo más simple, como volverlos a numerar de q_0 hasta q_n.



Para este punto, ya hemos terminado de convertir nuestro autómata finito no determinista en uno determinista.

2.5. Minimización AFD

[2] Tras construir un AFD, es posible que queden estados redundantes o equivalentes que no aportan información adicional sobre el lenguaje reconocido. La ***minimización*** consiste en identificar y colapsar estos estados equivalentes para obtener un AFD más pequeño y eficiente. Esto resulta especialmente importante en un compilador, donde el analizador léxico procesa cada símbolo de la entrada, y cualquier reducción en el número de estados mejora el rendimiento y reduce la memoria utilizada.

Formalmente, se define una ***relación de equivalencia entre estados***:

Dado un AFD $\langle Q, \Sigma, \delta, q_0, F \rangle$, dos estados $p, q \in Q$ son equivalentes, denotado $p \approx q$, si para toda cadena $x \in \Sigma^*$ se cumple:

$$\delta^*(p, x) \in F \iff \delta^*(q, x) \in F$$

Intuitivamente, dos estados son equivalentes si, sin importar qué cadena se lea, ambos llevan a aceptación o rechazo de manera idéntica.

El ***algoritmo de minimización*** se realiza mediante los siguientes pasos:

1. Construir una tabla de pares de estados p, q.
2. Marcar los pares donde uno de los estados sea final y el otro no.
3. Iterativamente, marcar un par p, q si para algún símbolo $a \in \Sigma$, la transición $\delta(p, a)$ y $\delta(q, a)$ lleva a un par ya marcado.
4. Los pares que queden sin marcar corresponden a estados equivalentes que se pueden colapsar en un solo estado.

Como resultado, se obtiene un AFD mínimo, que reconoce el mismo lenguaje pero con menos estados y transiciones, lo que se traduce en eficiencia y simplificación del analizador léxico.

2.6. Componente léxico

[3] Un ***componente léxico***, también denominado ***token***, es la unidad más pequeña con significado que un analizador léxico puede reconocer dentro del código fuente de un programa. Ejemplos de componentes léxicos en un lenguaje de programación incluyen palabras reservadas, identificadores, operadores y símbolos relacionales utilizados por el lenguaje.

2.7. Categoría Léxica

[3] En el análisis léxico, una ***categoría léxica*** representa el tipo de componente léxico que un lexer reconoce dentro del código fuente. Cada token reconocido se clasifica dentro de una categoría que refleja su función en el lenguaje de programación.

Ejemplos de categorías léxicas incluyen:

- **Palabras reservadas:** por ejemplo, `if`, `while`, `return`.
- **Identificadores:** nombres de variables, funciones o tipos definidos por el usuario.
- **Constantes literales:** números, cadenas de texto o caracteres.
- **Operadores y símbolos:** símbolos aritméticos, lógicos o de puntuación como $+$, $-$, $=$, `;`.

La asignación de cadenas a categorías léxicas permite que el analizador sintáctico pueda procesar la estructura del programa de manera consistente.

2.8. MDD. Máquina Discriminadora Determinista

[4] Una ***Máquina Discriminadora Determinista (MDD)*** extiende la noción de AFD al análisis léxico para clasificar cadenas en diferentes categorías. A diferencia de un AFD que solo acepta o rechaza, una MDD devuelve la categoría léxica de cada lexema reconocido.

Formalmente, una MDD se define como:

$$M = (Q, \Sigma, \delta, q_0, F, \mu, \perp)$$

donde:

- Q es el conjunto finito de estados.
- Σ es el alfabeto de entrada.
- $\delta : Q \times \Sigma \to Q$ es la función de transición determinista.
- $q_0 \in Q$ es el estado inicial.
- $F \subseteq Q$ es el conjunto de estados finales.
- $\mu : F \to C$ asigna una ***categoría léxica*** de un conjunto $C = c_1, \ldots, c_k$ a cada estado final.
- $\perp$ representa un error léxico, cuando la cadena no es reconocida.

La MDD funciona evaluando la cadena de entrada y asociando cada lexema al estado final alcanzado mediante μ. El prefijo máximo de la cadena que lleva a un estado final define el token reconocido, garantizando que cada lexema se clasifique correctamente.

2.9. Analizador Léxico

[3] El ***analizador léxico***, también llamado ***lexer***, es el componente del compilador encargado de leer el código fuente y transformarlo en una secuencia de tokens con categorías léxicas asociadas.

Su funcionamiento teórico se puede resumir en los siguientes pasos:

1. Leer la cadena de entrada carácter por carácter.
2. Identificar el prefijo máximo que forma un lexema reconocido por la MDD.
3. Asignar la categoría léxica correspondiente a dicho lexema mediante la función μ de la MDD.
4. Avanzar en la cadena y repetir el proceso hasta procesar toda la entrada.

Gracias a esta organización, el lexer permite que las etapas posteriores del compilador, como el análisis sintáctico y semántico, trabajen sobre una secuencia de tokens bien definidos, abstraídos de los detalles de los caracteres individuales.

3. Metodología e Implementación

En esta sección se describe el método utilizado para la construcción de un analizador léxico a partir de las definiciones formales previamente establecidas. Se detalla la secuencia de transformaciones necesarias para convertir la especificación léxica de un lenguaje de programación (más adelante, ejemplificado con el lenguaje IMP) en un autómata determinista mínimo capaz de reconocer los componentes léxicos correspondientes.

El desarrollo parte de la representación de los patrones léxicos mediante expresiones regulares, las cuales se transforman sucesivamente en autómatas equivalentes: primero en un $AFN-\varepsilon$, luego en un AFN, y posteriormente en un AFD. Este autómata determinista se somete a un proceso de minimización para eliminar redundancias, obteniendo así un AFD_{min} o autómata determinista mínimo. Finalmente, dicho autómata se convierte en una estructura manipulable por el analizador léxico (MDD), que permite identificar los lexemas de entrada y clasificarlos en las categorías léxicas definidas por el lenguaje.

Cada subsección describe con detalle los pasos de este flujo de transformación y los aspectos clave de la implementación en Haskell que permiten materializar dichas construcciones formales.

3.1. Conversión de una expresión regular en un $AFN-\varepsilon$

Durante esta primera etapa, nuestro objetivo es convertir una expresión regular en un autómata finito no determinista con transiciones epsilon, ya que a partir de las expresiones regulares podemos definir el lenguaje que buscamos aceptar, y con su equivalente en autómata, siendo el $AFN-\varepsilon$ el autómata más sencillo (directo) de construir, podemos procesar las cadenas de entrada para una primera etapa de aceptación o rechazo de la misma.

1. Conversión de String a Regex.

Inicialmente recibimos una especificación de una expresión regular en formato String, por lo que nuestro primer paso es convertir dicha string en algo de tipo Regex, siendo el tipo Regex el siguiente:

```haskell
data Regex = Symbol Char
           | Concat Regex Regex
           | Union Regex Regex
           | Star Regex
           deriving (Show, Read, Eq)
```

Los cuales vemos se corresponden con los operadores concat (sin operador explícito), union (+), star (*), además de los símbolos, por lo que en primera instancia nos conviene establecer estos operadores como palabras reservadas, al igual que los delimitadores necesarios, como '(', ')', resultando en:

```haskell
reserved :: [Char]
reserved = ['+', '*', '(', ')','[', ']']
```

Los corchetes en este caso, es necesario reservarlos para los casos de listas (o rangos), como "[a-A],[1...9]"que también son válidos en expresiones regulares.

A continuación la función principal para obtener la Regex correspondiente:

```
getRegex :: String -> Regex
getRegex s = case getRegexAux Nothing (filter (not . isSpace) s) of
               Just expr -> expr
               Nothing -> error "Expresión vacía no permitida"
```

Podemos ver que el uso de Nothing y Just es una consideración de diseño para representar epsilón mediante el uso de Nothing, en otro caso Just representa a la expresión. Luego continuamos con la llamada a la función auxiliar, donde como parámetros de entrada tenemos un acumulador para la expresión y la cadena estandarizada para que no contenga espacios en blanco.
A su vez está función auxiliar utiliza varias funciones auxiliares, debido a la cantidad de consideraciones necesarias para procesar la cadena, como los operadores posfijos (que a su vez usa la función *pase_complete_expresion*, el uso de paréntesis balanceados, así como el uso de corchetes para listas, entre otras consideraciones.
Por lo que terminamos con un código complejo que trataremos brevemente de retratar. Algunas de las funciones auxiliares que nos ayudan a mostrar dichas consideraciones del procesamiento de la cadena son las siguientes:

```
applyPostOperators :: Regex -> String -> (Regex, String)
```

Tuvimos que hacer una diferencia entre los operadores infijos y postfijos, debido a que la generación de la cadena resultaba complicada tratando a todos como infijos, debido a que el operador de la estrella de Kleen (*) necesita de la expresión previa para construirse correctamente, mientras que operadores como la concatenación y la unión se definen a partir de una parte izquierda y una derecha, e incluso de esta manera se necesita hacer una distinción en el caso de la unión, debido a que expresiones como .ª+b"se procesan diferente a "(ab)+c", así existe su definición como operador infijo y una diferente como operador postfijo.

```
extractBracketed :: String -> (String, String)
extractBracketed s = extract 1 '[' ']' [] s

extractParenthesized :: String -> (String, String)
extractParenthesized s = extract 1 '(' ')' [] s
```

El de arriba es un ejemplo de las funciones necesarias para manejar el caso de expresiones con paréntesis y corchetes, que aunque pareciera sencillo, la extracción de la parte interna y su respectivo manejo, ya sea considerando que existe un restante de la cadena por manejar o no, resulto en varias consideraciones extra.
También se tuvieron consideraciones como el uso de escape (\) para símbolos restringidos, para que pudiéramos incorporarlos de ser necesario.
Finalmente dejando estas consideraciones aparte el parseo de la string para convertirla en Regex se puede definir de la siguiente manera:

- El caso base donde la cadena es vacía, simplemente retornamos el acumulador.

```
        getRegexAux acc [] = acc

```

- Caso base, la cadena contiene un único símbolo. Donde también tenemos que considerar el caso de los paréntesis, de concatenación (si teníamos una expresión previa) y si no había expresión previa entonces es simplemente el símbolo. Recordemos que este es el caso donde no se detecto una operación como unión o star, ya que de ser el caso se procede de manera diferente, con operaciones auxiliares como parseCompleteExpression (en el caso de la estrella de Kleen).

```
        getRegexAux acc [x]
        | x == ')' = error "Parentesis de cierre no esperado"
        | x `elem` reserved = error $ "Símbolo reservado '" ++ [x] ++ "'
  necesita de una regex valida para utilizarse"
        | otherwise = case acc of
                      Nothing -> Just (Symbol x)
                      Just a -> Just (Concat a (Symbol x))

```

- Luego procedemos a los casos recursivos:
  - ...

    . Lo primero que verificaremos es el caso de tener conjuntos (rangos/listas) identificados a partir de corchetes ([a...A]), en este caso basta con expandir la lista usando el operador Union y verificar el uso de opereadores.

```
    getRegexAux acc ('[':xs)
    let (inside, restAfterBracket) = extractBracketed xs
        lista =  if null inside then error "Conjunto vacío no
   soportado" else '[' : inside ++ "]"
        expanded = expandList lista
        regexList = foldr1 Union (map Symbol expanded)
        (exprWithPostOps, remainingAfterPostOps) = applyPostOperators
    regexList restAfterBracket
        actual_acc = case acc of
          Nothing -> Just exprWithPostOps
          Just a  -> Just (Concat a exprWithPostOps)
    in getRegexAux actual_acc remainingAfterPostOps

```

  - (...). Similar ocurre con los paréntesis. En ambos casos se hace la distinción sobre el contenido del paréntesis y el resto de la cadena, pues inicialmente parseamos la expresión dentro de los paréntesis y luego procedemos a aplicar los postoperadores.

```
    getRegexAux acc (x:xs)
    | x == ')' = error "Parentesis de cierre no esperado"
    | x == '(' =
        let (insideStr, restAfterParen) = extractParenthesized xs
        in if null insideStr
           then error "Expresión vacía no permitida"
           else
             case parseCompleteExpression insideStr of
               Just inside ->
                 let (exprWithPostOps, remainingAfterPostOps) =
                       applyPostOperators inside restAfterParen
                 in
                   case acc of
                     Nothing -> getRegexAux (Just exprWithPostOps)
   remainingAfterPostOps
                     Just a ->
                       case remainingAfterPostOps of
                         ('+':restAfterPlus) ->
                           case getRegexAux Nothing restAfterPlus
   of
```

```
                                    Nothing -> error "Expresión esperada
    después de '+'"
                                    Just right -> Just (Union (Concat a
    exprWithPostOps) right)
                                     _ -> getRegexAux (Just (Concat a
    exprWithPostOps)) remainingAfterPostOps
```

- Unión (+). Para el caso de la unión basta con verificar que en efecto tenemos una expresión izquierda construida, hacer la llamada para la construcción derecha y englobarlo dentro de Union, pues consideramos que estamos tratando con el caso más básico de la Union, tratándolo como un operador infijo.

```
            x == '+' =
        case acc of
          Nothing -> error "Expresión esperada antes de '+'"
          Just left ->
            case getRegexAux Nothing xs of
              Nothing -> error "Expresión esperada después de '+'"
              Just right -> Just (Union left right)
```

- Star (*). En este caso directamente podemos generar Star(Symbol a), que es el caso más básico, debido a que para aplicar la estrella de Kleen a expresiones más complejas es necesario el uso de paréntesis que se plantea anteriormente, los únicos casos para analizar es si hay una expresión antes y/o después de la estrella.

```
             not (null xs) && head xs == '*' =
            let remaining = tail xs
                starred = Star (Symbol x)
            in case getRegexAux (Just starred) remaining of
              Nothing -> case acc of
                Nothing -> Just starred
                Just a -> Just (Concat a starred)
              Just next -> case acc of
                Nothing -> Just next
                Just a -> Just (Concat a next)
```

Por último cabe notar que eliminamos la posibilidad de que el último símbolo sea un operador, pero para generalizar la idea de que los operadores sean tratados como tal y no como símbolos (ayudando a la adecuada construcción de regex), establecemos la siguiente línea:

```
    | x `elem` reserved =
        error $ "Símbolo reservado '" ++ [x] ++ "' en posición inválida"
```

2. Regex a AFNEp

Considerando los elementos de un autómata definiremos el tipo $AFN - \varepsilon$ de la siguiente manera:

```
data AFNEp = AFNEp {
  estados :: [String],
  alfabeto :: [Char],
  transiciones :: [Trans_eps],
  inicial :: String,
  final :: String
} deriving (Show, Eq)
```

Mientras que las transiciones las podemos construir como se ve a continuación:

```
type Trans_eps = (String, Maybe Char, [String])
```

Donde el primer String corresponde al estado actual, Maybe Char es el carácter a procesar y [String] son los estados a lo que podemos llegar.
La función principal para esta etapa la llamamos **expr_to_AFNEp**, donde recibe una regex y devuelve su correspondiente AFNEp, esta a su vez llama a la función auxiliar con los siguientes parámetros:

```
expr_to_AFNEp_aux regex 0
```

De esta manera iniciamos la construcción del AFNEp con la cadena regex y un total de 0 estados generados. A continuación analizaremos recursivamente las operaciones presentes en regex y construiremos sus respectivos autómatas, teniendo como posibles casos:

- Symbol a. El caso base para la construcción del autómata más básico es que la regex conste de un único símbolo, en cuyo caso basta con generar dos estados, el inicial y el final, indicando que llegamos al final procesando a. Agregamos el símbolo 'a' a nuestro alfabeto y contabilizamos 2 estados más, el número de estados es lo que nos ayuda a nombrar de forma única a los estados que generemos, pues inicialmente los nombramos q_0, q_1 para después renombrarlos en función de la cantidad de estados que tengamos.

```
expr_to_AFNEp_aux (Symbol a) n =
  (AFNEp {
    estados = [q0, q1],
    alfabeto = [a],
    transiciones = [(q0, Just a, [q1])],
    inicial = q0,
    final = q1
  }, n + 2)
  where
  q0 = "q" ++ show n
  q1 = "q" ++ show (n + 1)}


```

- Union a b. Si nuestra regex resulta ser una unión entonces consideramos el caso recursivo para la expresión a, que genera su respectivo autómata y del mismo modo para la expresión, generamos un nuevo estado inicial, que conectaremos mediante transiciones epsilón a los iniciales de a y b, y finalmente establecemos un nuevo estado final conectando los finales de a y b (que dejan de ser finales).

```
expr_to_AFNEp_aux (Union a b) n =
  (AFNEp {
    estados = estados m1 ++ estados m2 ++ [q0, q1],
    alfabeto = rmDup $ alfabeto m1 ++ alfabeto m2,
    transiciones = transiciones m1 ++ transiciones m2
                   ++ [(q0, Nothing, [inicial m1, inicial m2]),
                       (final m1, Nothing, [q1]),
                       (final m2, Nothing, [q1])],
    inicial = q0,
    final = q1
  }, n')
  where
    (m1, na) = expr_to_AFNEp_aux a n
    (m2, nb) = expr_to_AFNEp_aux b na
    q0 = "q" ++ show nb
    q1 = "q" ++ show (nb + 1)
    n' = nb + 2

```

- Concat a b. Este caso es de los más sencillos pues no tenemos que generar estados nuevos, ya que el estado inicial de a, se establece como el inicial del autómata, mientras que el final de b, es el final del autómata, para finalmente enlazar el final del autómata de a, con el inicial de b, a través de una transición epsilón. La recursión se aplica en la generación de los respectivos autómatas de a y b.

```
expr_to_AFNEp_aux (Concat a b) n =
  (AFNEp {
    estados = estados m1 ++ estados m2,
    alfabeto = rmDup $ alfabeto m1 ++ alfabeto m2,
    transiciones = transiciones m1 ++ transiciones m2
                   ++ [(final m1, Nothing, [inicial m2])],
    inicial = inicial m1,
    final = final m2
  }, n')
  where
    (m1, na) = expr_to_AFNEp_aux a n
    (m2, n') = expr_to_AFNEp_aux b na

```

- Star a*. Al igual que en los anteriores tenemos que generar el autómata de a, luego simular el comportamiento de star tomando en cuenta dos consideraciones, la primera es dos estados nuevos para el inicial y el final, tenemos que conectar q0 (el nuevo inicial) con el inicial de a y a nuestro nuevo final q1, a través de transiciones epsilón, que como vemos se condensa en una linea, simplemente indicando que estados se conectan con epsilón, de esta forma podemos aceptar epsilón o empezar a procesar la cadena; como segunda consideración conectamos el final de a, a nuestro nuevo final q1, para simplemente aceptar la cadena, y lo conectamos al estado inicial de a, para volver a procesar la cadena, todo con transiciones epsilón.

```
expr_to_AFNEp_aux (Star a) n =
  (AFNEp {

     estados      = estados m1 ++ [q0, q1],
     alfabeto     = alfabeto m1,
```

```
        transiciones = transiciones m1
                       ++ [(q0, Nothing, [inicial m1, q1]),
                           (final m1, Nothing, [inicial m1, q1])],
        inicial      = q0,
        final        = q1
    }, n')
    where
      inicial n
      (m1, na) = expr_to_AFNEp_aux a n
      q0 = "q" ++ show na
      q1 = "q" ++ show (na + 1)
      n' = na + 2

```

3.2. Transformación de $AFN - \varepsilon$ en AFN

Para esta sección utilizaremos todo los previamente descrito, es decir que aunque asumimos un AFNEp, lo que queremos es recibir una cadena con especificación Regex la cual convertiremos a un AFNEp, así nuestra función principal getAFN es :

```
getAFN s = afnEp_to_AFN (getAFNEp s)
```

Mientras que nuestra estructura para describir un AFN es la siguiente:

```
afnEp_to_AFN m =  AFN {
  estadosN = estados m,
  alfabetoN =  alfabeto m,
  transicionesN = filterEmptyTransitions (trans_eps_to_afn m),
  inicialN = inicial m,
  finalN = final m
}
```

Como podemos ver en la parte de arriba, la estructura del AFNEp se conserva en su mayoría, excepto por las transiciones, a las cuales les aplicamos la función *trans_eps_to_afn*, correspondiente a la función closure vista en el marco teórico, veamosla a continuación:

```
trans_eps_to_afn m = concatMap (trans_eps_to_afn_aux m (transiciones m) (
   alfabeto m)) (estados m)
```

En este caso aplicamos concatMap para aplicar la función auxiliar a cada estado y obtener sus respectivas transiciones sin epsilón. Esta función auxiliar esta compuesta de la siguiente manera:

```
trans_eps_to_afn_aux _ _ [] _ = []
trans_eps_to_afn_aux m l (c:cs) q = (q, c, qn) : (trans_eps_to_afn_aux m l cs
   q)
  where qn = eclosure2 m $ do_trans_nep2 l c (eclosure l m q)
```

Como podemos ver utilizamos tres funciones auxiliares para calcular los estados alcanzables, estos es de inicio con eclosure el cuál ve los estados alcanzables con transiciones epsilon, luego do_trans_nep2 calcula a donde podemos llegar de los estados alcanzados previamente con cada símbolo del alfabeto y finalmente desde ahí un segundo eclosure para verificar los estados alcanzables por última vez con epsilón transiciones. Esto define δ' completando el proceso de construcción del AFN.

3.3. Transformación de *AFN* en *AFD*

La finalidad de esta etapa es tomar un autómata finito no determinista y convertirlo en un autómata finito determinista a partir de generar tablas de transiciones. Para este fin, se utiliza una nueva estructura para representar los estados: los **superEstados**

```
type SuperState = (String, [String])
```

Esta estructura es específicamente útil para representar conjuntos de estados, que se estarán utilizando para representar los nuevos estados del autómata finito, pues permite guardar una referencia a todos los estados que lo conforman a la vez que mantiene un nombre de estado simplificado (como q_0)

La conversión requiere de diversas funciones auxiliares, probablemente una de las más importantes sea:

Procesado de Símbolos
El algoritmo requiere una manera de poder conocer a qué estados puede transitar un estado o conjunto de estado utilizando el **alfabeto** (Σ) del autómata. Dado un símbolo $\alpha \in \Sigma$, pueden surgir 3 casos diferentes:

- El *superestado* no tiene transiciones consumiendo α
- El *superestado* tiene transiciones consumiendo α y es un *superestado* que ya hemos encontrado anteriormente
- El *superestado* tiene transiciones consumiendo α y es un *superestado* nuevo nunca antes visto durante el algoritmo.

```
processSymbol :: AFN -> [SuperState] -> SuperState -> ([Trans_afd], [SuperState], Int) -> Char -> ([Trans_afd], [SuperState], Int)
processSymbol n allKnownStates (currentName, currentAFNStates) (accTrans, accWorklist, accId) symbol =
    if isTargetEmpty
    then (accTrans, accWorklist, accId)
    else
        case maybeTargetSuperState of
            Just (targetName, _) -> ((currentName, symbol, targetName) : accTrans, accWorklist, accId)

            Nothing -> (newTrans : accTrans, newSuperState : accWorklist, accId + 1)
    where
        targetAFNStates = move n currentAFNStates symbol
        isTargetEmpty = null targetAFNStates

        maybeTargetSuperState = findSuperStateByContent targetAFNStates allKnownStates

        newName = "q" ++ show accId
        newSuperState = (newName, targetAFNStates)
        newTrans = (currentName, symbol, newName)
```

Construcción de AFD

Este es el núcleo principal del algoritmo, el punto principal del algoritmo es ir procesando nuevos superestados a partir del superestado inicial {q0}, cada que encontremos un superestado nuevo utilizando `processSymbol`, lo iremos agregando a la lista de superestados principal, además de agregarlo a una lista de superestados a procesar, vamos a terminar el algoritmo en el momento que nuestra lista de superestados a procesar esté completamente vacía, pues implica que ya hemos encontrado todos los posibles superestados **alcanzables** a partir del estado inicial, esto nos garantiza que no tendremos estados inalcanzables y por ende inútiles.

```
buildAFD :: AFN -> [SuperState] -> [SuperState] -> [Trans_afd] -> Int -> ([SuperState], [Trans_afd])
buildAFD _ [] processed trans _ = (processed, trans)
buildAFD n (currentSuperState:restWorklist) processed trans nextStateId =
    buildAFD n (restWorklist ++ newWorklistItems) (currentSuperState:processed) (trans ++ newTransitions) finalNextStateId
    where
        (currentName, currentAFNStates) = currentSuperState
        alfabeto = alfabetoN n

        (newTransitions, newWorklistItems, finalNextStateId) =
            foldl (processSymbol n allKnownStates currentSuperState) ([], [], nextStateId) alfabeto

        allKnownStates = processed ++ (currentSuperState:restWorklist)
```

De esta manera tenemos una lista de todos los superestados y sus transiciones en el nuevo autómata finito determinista, por la naturaleza de la estructura **superestado**, simplemente se cambia el conjunto de estados que representa a un nuevo estado por su nombre coloquial, es decir, el superestado {q0, q1, q2} pasa a a simplemente ser q2, de esta forma el AFD queda más compacto y entendible.

Seleccion de estados finales

Finalmente, el estado inicial en el AFN se mantiene como inicial en el AFD, y se tiene que determinar que estados serán finales. Un estado final en el AFD será aquel que en su superestado contenga como mínimo a un estado final del AFN.

```
findFinalStates :: String -> [SuperState] -> [String]
findFinalStates finalStateAFN allSuperStates =
    map fst $ filter (\(_, afnStates) -> finalStateAFN `elem` afnStates) allSuperStates
```

3.4. Minimización de un *AFD*

La etapa de minimización tiene como objetivo obtener un autómata determinista equivalente con el menor número posible de estados, sin alterar el lenguaje que reconoce.
En esta implementación, se desarrolló un módulo especializado que encapsula el proceso completo de minimización, manteniendo el tipo de datos `AFD` y aplicando una arquitectura modular con uso eficiente de estructuras como `Set` y `Map`.
El procedimiento se estructura en tres fases principales:

1. Eliminación de estados inalcanzables.

El proceso comienza filtrando los estados que no pueden alcanzarse desde el estado inicial. Para ello, la función `removeUnreachable` aplica una búsqueda iterativa mediante la función auxiliar `reachableStates`, que utiliza conjuntos (`Set`) para determinar de manera eficiente todos los estados accesibles por alguna secuencia de transiciones válidas. El AFD resultante mantiene solo los estados alcanzables, junto con sus transiciones y estados finales asociados.

```
removeUnreachable :: AFD -> AFD
removeUnreachable afd =
  let reachableSet = reachableStates afd (Set.singleton (inicialD afd))
      reachables   = Set.toList reachableSet
  in afd { estadosD = reachables
         , transicionesD = filter (\(e,_,_) -> Set.member e reachableSet)
                                  (transicionesD afd)
         , finalD = filter (`Set.member` reachableSet) (finalD afd)
         }
```

2. Refinamiento y agrupación de estados equivalentes.

El siguiente paso consiste en identificar los estados que son indistinguibles con respecto al lenguaje aceptado.
Inicialmente, se realiza una partición del conjunto de estados separando los finales de los no finales. A partir de esta base, la función `refinePartition` subdivide recursivamente los grupos, comparando el comportamiento de cada estado frente a los símbolos del alfabeto (`alfabetoD`). Este refinamiento continúa hasta alcanzar una partición estable, es decir, cuando ya no se producen cambios en los grupos.

Para acelerar las operaciones de búsqueda, se utiliza un mapa (`Map`) que asocia cada estado con el grupo al que pertenece. La función `groupEquivalentsDesde` finalmente asigna nombres nuevos a los grupos generados, a partir de un índice configurable (por ejemplo, `q0`, `q1`, ...).

```
groupEquivalentsDesde :: Int -> AFD -> [Grupo]
groupEquivalentsDesde start afd =
  let finals    = finalD afd
      nonFinals = filter (`Set.notMember` (Set.fromList finals)) (estadosD afd
   )
      initPart  = [finals, nonFinals]
      refined   = refinePartition afd initPart
  in zipWith (\i g -> ("q" ++ show (start + i), g)) [0..] refined
```

3. Reconstrucción del autómata mínimo.

Con la partición final estable, se construye un nuevo AFD en el que cada grupo de equivalencia se convierte en un único estado.
La función `buildTransitions` genera las transiciones del autómata resultante utilizando un mapa de correspondencias entre estados y grupos. Cada transición se define tomando como representante el primer elemento de cada grupo, ya que todos sus miembros son equivalentes respecto a la función de transición δ.
Además, se eliminan explícitamente las transiciones que conducen al estado trampa "Trash" mediante `filterTrashTransitions`, con el fin de limpiar el autómata final.

El nuevo estado inicial se determina con `findGroupName`, que identifica el grupo correspondiente al estado inicial original, mientras que los estados finales se definen como aquellos grupos que contienen al menos un estado final previo.

La función `minimizaAFDDesde` integra todos estos pasos y permite especificar desde qué número comenzar el nombrado de los nuevos estados.

La función `getAFDmin` actúa como punto de entrada principal, tomando una cadena que convierte a AFD y de este devuelve su versión minimizada.

```
getAFDmin :: String -> AFD
getAFDmin s = minimizaAFD $ getAFD s

minimizaAFDDesde :: Int -> AFD -> AFD
minimizaAFDDesde start afd =
  let afdReachable = removeUnreachable afd
      gruposEquiv  = groupEquivalentsDesde start afdReachable
      transMinRaw  = buildTransitions afdReachable gruposEquiv
      transMin     = filterTrashTransitions transMinRaw
      iniMin       = findGroupName (inicialD afdReachable) gruposEquiv
      finMin       = [ g | (g, qs) <- gruposEquiv
                         , any (`elem` finalD afdReachable) qs ]
  in AFD { estadosD = map fst gruposEquiv
         , alfabetoD = alfabetoD afdReachable
         , transicionesD = transMin
         , inicialD = iniMin
         , finalD = finMin }
```

3.5. Conversión de *AFD* en *MDD*

Esta etapa consiste en tomar un autómata finito determinista y convertirlo en una máquina discriminadora determinista, para este proceso es necesario saber los tokens que serán parte de la función de transición de la MDD, estas transiciones las definimos como Trans_mdd

```
type Trans_mdd = (String, String)
```

Cambien definimos ErrorMDD para indicar un error dentro de la MDD

```
type ErrorMDD = String
```

El resto de información necesaria es heredada del AFD con el que se formara la MDD, sin embargo, se tiene que realizar las siguientes operaciones para formar la MDD:

Obtener la función de transición.

Antes de poder hacer las asignaciones, tuvimos que modelar una forma de encontrar cada estado final de una ER en específico. Dentro del AFD, cada ER tiene concatenado al final "#token#"para poder recuperar qué estados finales deberían emitir cada token. Para la función de transición, obtendremos las asignaciones de cada token.

```
generarAsignaciones :: AFD -> Tokens -> [Trans_mdd]
generarAsignaciones afd [] = []
generarAsignaciones afd (x:tokens) = asignacionesFijas ++ asignacionesNuevas
```

```
        where
            asignacionesFijas = obtenerAsignacion afd x
            asignacionesNuevas = obtenerNuevos asignacionesFijas (
   generarAsignaciones afd tokens)
```

Para cada token se juntan todos los estados que se encuentran al recorrer al AFD en reversa con el token que se quiere asignar, como el token esta delimitado por '#'

```
obtenerAsignacion :: AFD -> String -> [Trans_mdd]
obtenerAsignacion afd token = let lista = (buscarFinales afd (finalD afd) (
   reverse ("#" ++ token ++ "#"))) in [(estadoF, token) | estadoF <- lista]
```

Obtener los estados finales.

Como ya se obtuvieron las asignaciones y estas asocian un estado final a un token, únicamente recuperamos estos estados.

```
obtenerFinales :: [Trans_mdd] -> [String]
obtenerFinales [] = []
obtenerFinales ((estado, _):asignaciones) = (estado):(obtenerFinales
   asignaciones)
```

Limpieza de estados y transiciones.

Con la función de asignación y los nuevos estados finales, podemos tomar las tranciociones de los estados finales que consumen el caracter '#' como transiciones inútiles,

```
obtenerTransicionesInutilesG :: [String] -> AFD -> [Trans_afd]
obtenerTransicionesInutilesG [] afd = []
obtenerTransicionesInutilesG (x:estados) afd = encontrarDesde aux afd  ++
                                obtenerTransicionesInutilesG estados afd
                                    where
                                      aux = buscarEstados '#' x trancisiones
                                      rancisiones = transicionesD afd
```

Una vez eliminadas esas transiciones también dejamos estados no accesibles, para eliminarlos basta con ver todos los estados a los nos mandan las transiciones inútiles.

```
estadosResultantes :: [Trans_afd] -> Set.Set String
estadosResultantes estados = (Set.fromList([estadoI | (_, _, estadoI) <-
   estados]))
```

3.6. Implementación del Analizador Léxico

Siguiendo los pasos del funcionamiento del Analizador Léxico tomamos una función para avanzar en la cadena a procesar y otra para el regreso si no se lego a un estado final.

Si se terminó de procesar una cadena se tienen los siguientes casos:

- El estado al que se llego sea el inicial. En este caso se devuelve una lista vacía puesto que ya no hay nada que procesar.

- El estado al que se llego sea un estado final. En este caso se puede emitir un token asociado a la cadena que se proceso.
- En caso contrario se emite el error léxico.

```
procesarAux :: String -> String -> [(Char, String)] -> MDD -> [(String, String
    )]
procesarAux [] estadoA visitados mdd =
            if estadoA == inicial && (length visitados == 0) then []
            else
              if estadoA `elem` finales then
                [(obtenerToken estadoA mdd, obtenerContenido visitados) ]
              else
               [(errorMDD mdd,"")]
            where
              inicial = inicialMDD mdd
              finales = finalMDD mdd
```

Por otro lado, para procesar un carácter de la cadena se toman los casos

- Existe una transición a la que avanzar. En este caso se avanza de estado y se guarda el carácter y el estado usado.
- No existe una transición a la que avanzar se pasa a la fase de regreso.

```
procesarAux (x:xs) estadoA visitados mdd =
            if (hayTrancision x estadoA transiciones)then
              procesarAux xs (funcionTransicionCaracter x estadoA transiciones
    ) ((x,estadoA):visitados) mdd
            else
              regresar (x:xs) estadoA visitados mdd
            where
              transiciones = transicionesMDD mdd
```

Para el regreso se emplean los caracteres y estados que ya se consumieron, esto usando una lista de visitados, si esta lista es vacia significa que durante el regreso no se encontro un estado final, por lo que se emite un error léxico.

Si la lista de visitados es no vacia, se toman los siguientes caos:

- Si se regresa a un estado final, se emite el token asociado a ese estado final con la cadena que se proceso hasta ese estado, además, se procesa lo que queda de la cadena desde el estado inicial.
- Si el estado al que se regresa no es final, se restaura el carácter en la cadena y se regresa al estado anterior, quitando ese par de los visitados.

```
regresar :: String -> String -> [(Char, String)] -> MDD -> [(String, String)]
regresar cadena estadoA [] mdd = [(errorMDD mdd,"")]
regresar cadena estadoA ((c, estadoR):xs) mdd =
            if estadoA `elem` finales then
                [(obtenerToken estadoA mdd, obtenerContenido ((c, estadoR):xs)
    )] ++ procesarAux cadena inicial [] mdd
            else
                regresar ([c] ++ cadena) estadoR xs mdd
            where
                finales = finalMDD mdd
                inicial = inicialMDD mdd
```

4. Resultados

El analizador léxico desarrollado en Haskell logró implementar correctamente todo el flujo de construcción teórico planteado, desde la especificación de expresiones regulares hasta la obtención de una máquina determinista discriminadora funcional para cualquier lenguaje.
A través de la combinación modular de los componentes:

- Regex.hs
- AFNEp.hs
- AFN.hs
- AFNmin.hs
- MDD.hs
- LectorArchivo.hs

se consiguió un sistema capaz de procesar especificaciones léxicas de manera declarativa, generando autómatas que reconocen los patrones definidos sin intervención manual.

Uso de pruebas unitarias
Para validar el correcto funcionamiento de cada módulo, se realizaron pruebas unitarias sobre expresiones regulares simples y compuestas.

Estas pruebas se encuentran organizadas en el directorio `samples/`, donde se incluye el archivo `samples/imp/Test.hs`, el cual contiene los casos de prueba correspondientes a todas las etapas del analizador, desde la conversión de expresiones regulares hasta la obtención del autómata determinista mínimo. Asó mismo, en este archivo se integran las pruebas específicas para el lenguaje IMP, con el fin de evaluar el comportamiento completo del lexer en un contexto de lenguaje real.

Integración con la especificación del lenguaje IMP
El analizador se aplicó posteriormente a una especificación léxica basada en el lenguaje IMP, que contempla categorías como identificadores, números enteros, operadores aritméticos, operadores booleanos, delimitadores y palabras reservadas.

El sistema generó una *Máquina Discriminadora Determinista (MDD)* y un **lexer** capaces de reconocer, discriminar y clasificar correctamente todas las categorías léxicas definidas conforme a las expresiones regulares del lenguaje, y no solo del este lenguaje, si no de cualquier lenguaje regular que se especifique junto con sus etiquetas y expresiones regulares que lo definen en un archivo de texto.

Diseño de la especificación léxica (`specs/IMP.md`)
Como parte del diseño del sistema, se optó por separar la definición de las expresiones regulares del código fuente, almacenándolas en un archivo externo llamado `specs/IMP.md`. Este enfoque modular facilita la modificación y extensión de la especificación léxica sin necesidad de alterar el código del analizador. El archivo contiene la descripción de cada categoría léxica junto con su correspondiente expresión regular, siguiendo el formato mostrado a continuación:

```
entero
0 + ['1'..'9']['0'..'9']* + (-['1'..'9']['0'..'9'])*
palabraReservada
true + false + skip + if + then + else + while + do
identificador
[['a'..'z'] ++ ['A'..'Z']]([['a'..'z'] ++ ['A'..'Z']]*)['0'..'9']*
operador
(\+)+(-)+(\*)
opbool
= + <= + not + and
asignacion
:=
delimitador
;
```

5. Conclusión

Durante el desarrollo de este proyecto enfrentamos diversos retos técnicos y conceptuales. Implementar un analizador léxico desde cero implicó comprender a fondo los fundamentos teóricos de los autómatas y las expresiones regulares, además de traducirlos a una implementación funcional en Haskell, un lenguaje en el que no sentíamos tanta confianza al programar por la poca experiencia que teníamos con él. Uno de los mayores desafíos fue manejar la complejidad inherente al proceso de conversión de un AFN a un AFD, cuyo costo puede alcanzar el orden de 2^n en el peor de los casos. Esto nos llevó a buscar soluciones eficientes y cuidadosas en la representación de estados y transiciones.

A pesar de la dificultad, el aprendizaje fue muy significativo. Comprendimos por qué el análisis léxico constituye una etapa esencial del compilador, al sentar las bases sobre las cuales se construyen las fases sintáctica y semántica. A lo largo del proyecto se desarrolló un flujo completo que inicia con la definición de expresiones regulares y concluye con la implementación de una Máquina Discriminadora Determinista (MDD) capaz de reconocer y clasificar componentes léxicos dentro de un lenguaje de programación, reforzando así la conexión entre los modelos teóricos y su aplicación computacional.

Finalmente, la creación del analizador nos permitió reconocer la importancia de la optimización y la simplificación en los compiladores: la minimización de estados no solo incrementa la eficiencia del análisis, sino que también facilita la comprensión y el mantenimiento del sistema. El resultado final es un analizador léxico modular, escalable y formalmente correcto, capaz de adaptarse a distintas especificaciones léxicas, en particular al lenguaje IMP utilizado como base experimental.

Bibliografía

[1] R. De Castro Korgi, *Teoría de la Computación. Lenguajes, autómatas, gramáticas*. Bogotá D.C., Colombia: Universidad Nacional de Colombia, Facultad de Ciencias, UNIBIBLOS, primera ed., 2004.

[2] M. Soto Romero, "De las expresiones regulares a los autómatas finitos deterministas." `https://lambdasspace.github.io/CMP/notas/cmp_n08.pdf`, 2025. Nota de clase.

[3] M. Soto Romero, "Sintaxis léxica." `https://lambdasspace.github.io/CMP/notas/cmp_n06.pdf`, 2025. Nota de clase.

[4] M. Soto Romero, "Máquinas discriminadoras deterministas." `https://lambdasspace.github.io/CMP/notas/cmp_n09.pdf`, 2025. Nota de clase.