Proyecto 01

CONSTRUCCIÓN DE UN ANALIZADOR LÉXICO

Compiladores

by λexers

Universidad Nacional Autónoma de México
Facultad de Ciencias

Compiladores

Proyecto 1

Cruz Pineda Fernando
Espinosa Roque Rebeca
Guzmán Bucio Luis Antonio
Morales Martínez Edgar Jesús
Vázquez Dávila José Adolfo

October 22, 2025

Contents

0.1 Introducción

El análisis léxico constituye la primera fase del proceso de compilación, encargada de transformar una secuencia de caracteres en una secuencia de tokens que serán procesados por las fases subsecuentes. El presente trabajo desarrolla un analizador léxico completo para el lenguaje imperativo simple IMP, aplicando sistemáticamente la teoría de lenguajes formales y autómatas finitos.

0.1.1 Objetivos

El presente trabajo tiene los siguientes objetivos:

- Construir un analizador léxico para el lenguaje IMP mediante la aplicación sistemática de la teoría de lenguajes formales y autómatas.
- Implementar de manera rigurosa la cadena de transformaciones

$$\text{ER} \to \text{AFN} - \varepsilon \to \text{AFN} \to \text{AFD} \to \text{AFD}_{\text{min}} \to \text{MDD} \to \texttt{lexer}$$

  documentando cada etapa constructiva.
- Validar el analizador mediante casos de prueba representativos del lenguaje IMP, verificando corrección y completitud del reconocimiento léxico.

0.2 Preliminares Formales

0.2.1 Lenguajes Formales

Alfabetos, Cadenas y Lenguajes

Definición 1 ***Alfabeto y Cadenas***

Sea Σ un conjunto finito no vacío.

1. Σ se denomina **alfabeto**.
2. Una **cadena** (o palabra) sobre Σ es una secuencia finita

$$w = a_1 a_2 \cdots a_n$$

   donde $a_i \in \Sigma$ para $1 \leq i \leq n$. La cadena vacía se denota ε.
3. La **longitud** de una cadena w, denotada $|w|$, es el número de símbolos en w. Se tiene $|\varepsilon| = 0$.

Definición 2 ***Cerradura de Kleene***

Sea Σ un alfabeto. La **cerradura de Kleene** de Σ, denotada Σ^*, se define como

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$$

donde $\Sigma^0 = \{\varepsilon\}$ y $\Sigma^{n+1} = \{wa \mid w \in \Sigma^n, a \in \Sigma\}$ para $n \geq 0$.
La **cerradura positiva** Σ^+ se define como $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Definición 3 ***Lenguaje Formal***

Un **lenguaje formal** sobre un alfabeto Σ es un subconjunto $L \subseteq \Sigma^*$.

Expresiones Regulares

Las expresiones regulares constituyen un formalismo algebraico para especificar lenguajes regulares mediante la aplicación recursiva de operadores sobre un alfabeto base.

Definición 4 **Expresión Regular**

Sea Σ un alfabeto. El conjunto de **expresiones regulares** sobre Σ, denotado $\mathcal{RE}(\Sigma)$, y la función semántica $L : \mathcal{RE}(\Sigma) \to \mathcal{P}(\Sigma^*)$ se definen inductivamente:
Casos base:

1. $\emptyset \in \mathcal{RE}(\Sigma)$ y $L(\emptyset) = \emptyset$
2. $\varepsilon \in \mathcal{RE}(\Sigma)$ y $L(\varepsilon) = \{\varepsilon\}$
3. $\forall a \in \Sigma : a \in \mathcal{RE}(\Sigma)$ y $L(a) = \{a\}$

Paso inductivo: Si $r, s \in \mathcal{RE}(\Sigma)$, entonces:

1. $(r + s) \in \mathcal{RE}(\Sigma)$ y $L(r + s) = L(r) \cup L(s)$ (unión)
2. $(r \cdot s) \in \mathcal{RE}(\Sigma)$ y $L(r \cdot s) = \{uv \mid u \in L(r) \wedge v \in L(s)\}$ (concatenación)
3. $(r^*) \in \mathcal{RE}(\Sigma)$ y $L(r^*) = \bigcup_{i=0}^{\infty} L(r)^i$ (cerradura de Kleene)

Ningún otro objeto es una expresión regular sobre Σ.

Teorema 1 **Kleene**

Sea Σ un alfabeto y $L \subseteq \Sigma^*$. Entonces L es regular si y solo si existe $r \in \mathcal{RE}(\Sigma)$ tal que $L = L(r)$.

Proof. ($\to$) Sea L regular. Entonces existe un AFD $M = (Q, \Sigma, \delta, q_0, F)$ tal que $L = L(M)$. Aplicando el algoritmo de eliminación de estados, se construye inductivamente una expresión regular r tal que $L(r) = L(M)$.

($\leftarrow$) Sea $r \in \mathcal{RE}(\Sigma)$. Proceder por inducción estructural sobre r:

Casos base: Para $\emptyset$, ε, y $a \in \Sigma$, se construyen trivialmente AFN-ε con estados apropiados.

Paso inductivo: Supóngase que existen AFN-ε N_1 y N_2 reconociendo $L(r)$ y $L(s)$ respectivamente. Entonces:

- Para $r + s$: se aplica la construcción de Thompson para unión.
- Para $r \cdot s$: se aplica la construcción de Thompson para concatenación.
- Para r^*: se aplica la construcción de Thompson para cerradura.

En cada caso, el autómata resultante es AFN-ε y reconoce el lenguaje correspondiente. Por conversión AFN-$\varepsilon \to$ AFD mediante la construcción de subconjuntos, $L(r)$ es regular. □

0.2.2 Análisis Léxico

Componentes Léxicos

Definición 5 ***Patrón, Lexema y Token***

En el contexto del análisis léxico, se distinguen tres conceptos fundamentales:

1. Un **patrón** es una descripción formal, típicamente mediante expresiones regulares, de la estructura que pueden adoptar los lexemas de un token.
2. Un **lexema** es una secuencia concreta de caracteres en el texto fuente que coincide con el patrón de algún token.
3. Un **token** es un par $\langle t, v \rangle$ donde t es el nombre o tipo del token (identificador, operador, palabra reservada, etc.) y v es el valor del atributo, usualmente el lexema correspondiente.

Ejemplo 1 ***Tokens y Lexemas en IMP***

Considérese el fragmento de código IMP:

$$\texttt{x := 42}$$

La secuencia de tokens generada es:

$$\langle \texttt{ID}, \text{“x”} \rangle, \quad \langle \texttt{ASSIGN}, \text{“:=”} \rangle, \quad \langle \texttt{NUM}, \text{“42”} \rangle$$

donde `x`, `:=`, y `42` son los lexemas correspondientes.

0.3 El Lenguaje IMP

0.3.1 Gramática Abstracta

El lenguaje IMP (Imperative Language) constituye un fragmento mínimo de lenguajes imperativos como C o Java, diseñado para el estudio formal de semánticas operacionales y denotacionales. Su sintaxis abstracta se especifica mediante las siguientes producciones en forma de Backus-Naur:

```
1  Aexp ::= n | x | Aexp + Aexp | Aexp - Aexp | Aexp * Aexp
2  Bexp ::= true | false | Aexp = Aexp | Aexp ≤ Aexp | not Bexp | Bexp and Bexp
3  Com  ::= skip | x := Aexp | Com; Com | if Bexp then Com else Com | while Bexp do Com
```

donde `Aexp` denota expresiones aritméticas, `Bexp` expresiones booleanas, y `Com` comandos.

0.3.2 Especificación Léxica

Alfabeto de IMP

Sea Σ_{IMP} el alfabeto sobre el cual se define la sintaxis léxica de IMP:

$$\begin{aligned}
\Sigma_{\text{digit}} &= \{0, 1, 2, \ldots, 9\} \\
\Sigma_{\text{lower}} &= \{a, b, \ldots, z\} \\
\Sigma_{\text{upper}} &= \{A, B, \ldots, Z\} \\
\Sigma_{\text{op}} &= \{+, -, *, =, \leq, ;, :, (,)\} \\
\Sigma_{\text{ws}} &= \{\sqcup, \backslash\texttt{t}, \backslash\texttt{n}\} \\
\Sigma_{\text{IMP}} &= \Sigma_{\text{digit}} \cup \Sigma_{\text{lower}} \cup \Sigma_{\text{upper}} \cup \Sigma_{\text{op}} \cup \Sigma_{\text{ws}} \cup \{/\}
\end{aligned}$$

Tokens mediante Expresiones Regulares

A continuación especificamos los tokens de IMP mediante expresiones regulares. Utilizamos la notación extendida donde r^+ denota $r \cdot r^*$ y $[c_1 c_2 \cdots c_n]$ denota $c_1 + c_2 + \cdots + c_n$.

Palabras reservadas:

Las palabras reservadas del lenguaje se reconocen exactamente como cadenas literales:

$$\begin{aligned}
\texttt{NOT} &\to \texttt{not} \\
\texttt{AND} &\to \texttt{and} \\
\texttt{IF} &\to \texttt{if} \\
\texttt{THEN} &\to \texttt{then} \\
\texttt{ELSE} &\to \texttt{else} \\
\texttt{SKIP} &\to \texttt{skip} \\
\texttt{WHILE} &\to \texttt{while} \\
\texttt{DO} &\to \texttt{do} \\
\texttt{TRUE} &\to \texttt{true} \\
\texttt{FALSE} &\to \texttt{false}
\end{aligned}$$

Literales enteros:

Siguiendo la convención de que los enteros positivos no llevan signo explícito y que el cero se representa únicamente como `0`:

$$\begin{aligned}
\text{nonzero} &\to [1\text{-}9] \\
\text{digit} &\to [0\text{-}9] \\
\text{nat} &\to 0 + \text{nonzero} \cdot \text{digit}^* \\
\text{NUM} &\to \text{nat} + (\underline{-} \cdot \text{nonzero} \cdot \text{digit}^*)
\end{aligned}$$

donde $\underline{-}$ denota el símbolo literal de resta (no el operador de alternancia).

Restricciones léxicas: Esta especificación excluye explícitamente:

- Representaciones como -0 (cero con signo)
- Enteros con ceros a la izquierda no significativos (e.g., 007, -042)
- La expresión regular garantiza que:
  - El único entero que comienza con `0` es `0` mismo
  - Los enteros negativos deben ser $-[1\text{-}9][0\text{-}9]^*$ (sin -0 ni ceros iniciales)

Operadores:

Para distinguir los operadores del metalenguaje de expresiones regulares de aquellos pertenecientes a IMP, subrayamos estos últimos:

$$\begin{aligned}
\texttt{PLUS} &\to \underline{+} \\
\texttt{MINUS} &\to \underline{-} \\
\texttt{TIMES} &\to \underline{*} \\
\texttt{EQ} &\to \underline{=} \\
\texttt{LEQ} &\to \underline{\leq} \\
\texttt{SEMICOLON} &\to \underline{;} \\
\texttt{ASSIGN} &\to \underline{:=}
\end{aligned}$$

Delimitadores:

$$\begin{aligned}
\texttt{LPAREN} &\to \underline{(} \\
\texttt{RPAREN} &\to \underline{)}
\end{aligned}$$

Identificadores:
Los identificadores deben comenzar con una letra y pueden contener letras y dígitos:

$$\texttt{letter} \to [A\text{-}Z] + [a\text{-}z]$$
$$\texttt{ID} \to \texttt{letter} \cdot (\texttt{letter} + \texttt{digit})^*$$

Espacios en blanco:
Los caracteres de espaciado son reconocidos pero descartados:

$$\texttt{ws\_char} \to \sqcup + \texttt{\textbackslash t} + \texttt{\textbackslash n}$$
$$\texttt{WHITESPACE} \to \texttt{ws\_char}^+$$

Comentarios de línea:
IMP admite comentarios de una sola línea iniciados por `//` y terminados por el carácter de nueva línea:

$$\texttt{any\_char} \to [\Sigma_{\text{IMP}} \setminus \{\texttt{\textbackslash n}\}]$$
$$\texttt{COMMENT} \to \texttt{//} \cdot \texttt{any\_char}^* \cdot \texttt{\textbackslash n}$$

Los tokens `WHITESPACE` y `COMMENT` se descartan durante el análisis léxico y no aparecen en la secuencia de tokens resultante.

0.3.3 Autómatas Finitos

Los autómatas finitos constituyen modelos matemáticos abstractos de máquinas de cómputo con memoria finita, capaces de reconocer exactamente la clase de lenguajes regulares. Presentamos las definiciones en orden creciente de restricción estructural.

Autómata Finito No Determinista con ε-transiciones

Definición 6 AFN-ε

Un **autómata finito no determinista con ε-transiciones** (AFN-ε) es una quíntupla

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$$

donde:

- Q es un conjunto finito de **estados**
- Σ es un **alfabeto** de entrada
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ es la **función de transición**
- $q_0 \in Q$ es el **estado inicial**
- $F \subseteq Q$ es el conjunto de **estados finales** o de aceptación

La función de transición δ permite dos formas de no determinismo:

1. Para un par (q, a) con $a \in \Sigma$, el conjunto $\delta(q, a)$ puede contener múltiples estados, indicando opciones de transición.

2. Las ε-transiciones $\delta(q, \varepsilon)$ permiten cambios de estado sin consumir símbolos de entrada.

Definición 7 **ε-cerradura**

Sea $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ un AFN-ε y $S \subseteq Q$. La **ε-cerradura** de S, denotada $\varepsilon\text{-cl}(S)$, es el menor conjunto que satisface:

1. $S \subseteq \varepsilon\text{-cl}(S)$
2. Si $q \in \varepsilon\text{-cl}(S)$ y $p \in \delta(q, \varepsilon)$, entonces $p \in \varepsilon\text{-cl}(S)$

Equivalentemente, $\varepsilon\text{-cl}(S)$ es el conjunto de estados alcanzables desde S mediante cero o más ε-transiciones.

Definición 8 **Función de transición extendida para AFN-ε**

La función de transición extendida $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ se define inductivamente:

- $\hat{\delta}(q, \varepsilon) = \varepsilon\text{-cl}(\{q\})$
- $\hat{\delta}(q, wa) = \varepsilon\text{-cl}\left(\bigcup_{p \in \hat{\delta}(q,w)} \delta(p, a)\right)$ para $w \in \Sigma^*, a \in \Sigma$

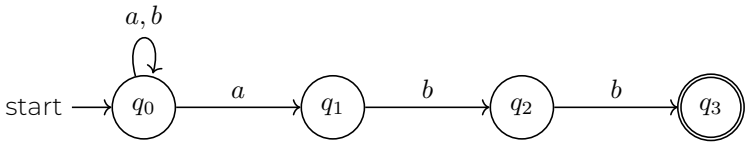
Definición 9 **Lenguaje aceptado por AFN-ε**

Sea $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ un AFN-ε. El **lenguaje aceptado** por $\mathcal{A}$ es

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Ejemplo 2 **AFN-ε para $(a+b)^*abb$**

Considérese el AFN-ε que reconoce cadenas sobre $\{a, b\}$ terminadas en abb:



El no determinismo se manifiesta en q_0 al leer el símbolo a: el autómata puede permanecer en q_0 o transitar a q_1.

Autómata Finito No Determinista

Definición 10 **AFN**

Un **autómata finito no determinista** (AFN) es un AFN-ε sin ε-transiciones. Formalmente, es una quíntupla $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ con

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

La semántica de aceptación para AFN se obtiene como caso particular de la definición para AFN-ε, donde todas las ε-cerraduras se reducen a la identidad.

Autómata Finito Determinista

Definición 11 **AFD**

Un **autómata finito determinista** (AFD) es una quíntupla

$$\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$$

donde:

- Q, Σ, q_0, F son como en AFN
- $\delta : Q \times \Sigma \to Q$ es una función total

La función de transición es determinista: para cada par $(q, a) \in Q \times \Sigma$, existe exactamente un estado $\delta(q, a)$.

Definición 12 **Función de transición extendida para AFD**

La función extendida $\hat{\delta} : Q \times \Sigma^* \to Q$ se define:

- $\hat{\delta}(q, \varepsilon) = q$
- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a) \quad$ para $w \in \Sigma^*, a \in \Sigma$

Definición 13 **Lenguaje aceptado por AFD**

Sea $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ un AFD. El lenguaje aceptado es

$$L(\mathcal{M}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

Teorema 2 **Equivalencia de AFN-ε, AFN y AFD**

Para todo AFN-ε $\mathcal{A}$, existe un AFD $\mathcal{M}$ tal que $L(\mathcal{A}) = L(\mathcal{M})$. En particular:

1. Todo AFNε puede convertirse en un AFN equivalente mediante eliminación de ε-transiciones.
2. Todo AFN puede convertirse en un AFD equivalente mediante la construcción de subconjuntos.

Esquema de Demostración: (1) AFN-$\varepsilon \to$ AFN: Dado $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, se construye $\mathcal{A}' = (Q, \Sigma, \delta', q_0, F')$ donde:

$$\delta'(q, a) = \varepsilon\text{-cl}(\delta(q, a)) \text{ para cada } q \in Q, a \in \Sigma$$
$$F' = \{q \in Q \mid \varepsilon\text{-cl}(\{q\}) \cap F \neq \emptyset\}$$

Es decir, un estado q es final en $\mathcal{A}'$ si y solo si desde q se puede alcanzar un estado final de $\mathcal{A}$ mediante cero o más transiciones ε.

(2) AFN $\to$ AFD: Dado $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, se construye $\mathcal{M} = (Q', \Sigma, \delta', q_0', F')$ mediante la construcción de subconjuntos:

$$
\begin{aligned}
Q' &= \mathcal{P}(Q) \\
q_0' &= \{q_0\} \\
\delta'(S,a) &= \bigcup_{q \in S} \delta(q,a) \text{ para } S \in Q', a \in \Sigma \\
F' &= \{S \in Q' \mid S \cap F \neq \emptyset\}
\end{aligned}
$$

□

Minimización de Autómatas Finitos Deterministas

Definición 14 ***Estados distinguibles***

Sea $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ un AFD. Dos estados $p, q \in Q$ son **distinguibles** si existe $w \in \Sigma^*$ tal que

$$\hat{\delta}(p,w) \in F \iff \hat{\delta}(q,w) \notin F$$

Se dice que p y q son **equivalentes** (denotado $p \sim q$) si no son distinguibles.

Teorema 3 ***Equivalencia es una relación de equivalencia***

La relación $\sim$ sobre Q es una relación de equivalencia, es decir, es reflexiva, simétrica y transitiva.

Definición 15 ***AFD mínimo***

Un AFD $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ es **mínimo** si:

1. Todos los estados en Q son alcanzables desde q_0, es decir,
$$\forall q \in Q, \exists w \in \Sigma^* : \hat{\delta}(q_0, w) = q$$
2. No existe AFD $\mathcal{M}'$ con menos estados tal que $L(\mathcal{M}) = L(\mathcal{M}')$.

Observación: La condición (1) es esencial, ya que un autómata puede tener estados inalcanzables que no afectan al lenguaje aceptado. La minimización estándar asume que todos los estados son alcanzables; si no lo son, deben eliminarse primero.

Teorema 4 ***Unicidad del AFD mínimo***

Para todo AFD $\mathcal{M}$, existe un único AFD mínimo $\mathcal{M}_{\text{min}}$ (salvo isomorfismo) tal que $L(\mathcal{M}) = L(\mathcal{M}_{\text{min}})$.

Construcción. Sea $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$. El AFD mínimo $\mathcal{M}_{\text{min}} = (Q/\sim, \Sigma, \delta', [q_0], F')$ se obtiene mediante:

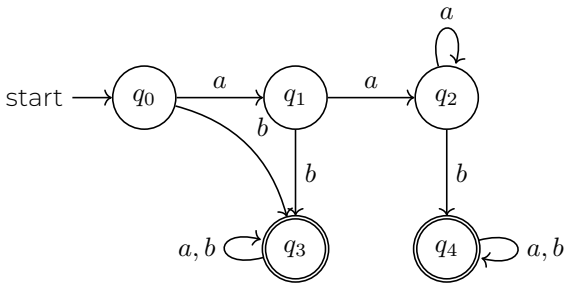
$$
\begin{aligned}
Q/\sim &= \{[q] \mid q \in Q\} \quad \text{(clases de equivalencia bajo } \sim) \\
\delta'([q],a) &= [\delta(q,a)] \\
F' &= \{[q] \mid q \in F\}
\end{aligned}
$$

La construcción está bien definida pues si $p \sim q$, entonces $\delta(p,a) \sim \delta(q,a)$ para todo $a \in \Sigma$. □

El algoritmo de minimización de Hopcroft calcula la partición $Q/\sim$ en tiempo $O(|Q| \log |Q| \cdot |\Sigma|)$ mediante refinamiento iterativo, partiendo de la partición inicial $\{F, Q \setminus F\}$.

Ejemplo 3 *Minimización de AFD*

Considérese el AFD con estados redundantes:



Aplicando el algoritmo de minimización, se identifica que $q_3 \sim q_4$ (ambos aceptan todas las cadenas), resultando en el AFD mínimo con 4 estados en lugar de 5.

Máquina Discriminadora Determinista

Definición 16 *MDD*

Una **Máquina Discriminadora Determinista** (MDD) es un AFD aumentado con información léxica. Formalmente, es una séxtupla

$$\mathcal{D} = (Q, \Sigma, \delta, q_0, F, \lambda)$$

donde $(Q, \Sigma, \delta, q_0, F)$ es un AFD y $\lambda : F \to \mathcal{T}$ es una función de **etiquetado** que asigna a cada estado final una categoría léxica (tipo de token) del conjunto $\mathcal{T}$ de tipos de tokens.

El conjunto de tipos de tokens $\mathcal{T}$ para IMP incluye:

$$\mathcal{T} = \{\texttt{NUM}, \texttt{ID}, \texttt{PLUS}, \texttt{MINUS}, \texttt{IF}, \texttt{WHILE}, \ldots\}$$

Definición 17 *Función de reconocimiento léxico*

Sea $\mathcal{D} = (Q, \Sigma, \delta, q_0, F, \lambda)$ una MDD. La función de reconocimiento $\rho : \Sigma^* \to \mathcal{T} \cup \{\bot\}$ se define como:

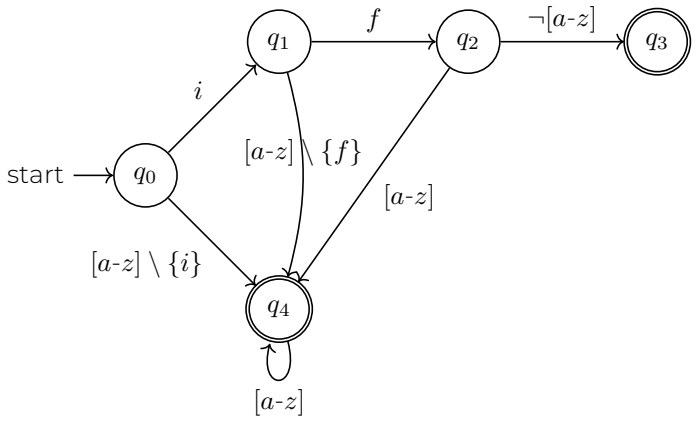
$$\rho(w) = \begin{cases} \lambda(\hat{\delta}(q_0, w)) & \text{si } \hat{\delta}(q_0, w) \in F \\ \bot & \text{si } \hat{\delta}(q_0, w) \notin F \end{cases}$$

donde $\bot$ denota rechazo léxico.

La MDD constituye la representación operacional final del analizador léxico, obtenida tras el proceso de minimización del AFD. Cada estado final discrimina una categoría léxica específica, permitiendo la clasificación inmediata del lexema reconocido.

Ejemplo 4 *MDD para identificadores y palabras reservadas*

Considérese una MDD que distingue entre identificadores y la palabra reservada `if`:



donde $\lambda(q_3) = \texttt{IF}$ y $\lambda(q_4) = \texttt{ID}$.

0.3.4 Analizador Léxico

Función del Analizador Léxico

Definición 18 *Analizador Léxico*

Un **analizador léxico** (o *lexer*) es una función

$$\texttt{lexer} : \Sigma^* \to (\mathcal{T} \times \Sigma^*)^* \cup \{\texttt{Error}\}$$

que transforma una cadena de entrada en una secuencia de tokens, donde cada token es un par $\langle t, v \rangle$ con $t \in \mathcal{T}$ (tipo de token) y $v \in \Sigma^*$ (lexema).

El analizador léxico debe resolver dos problemas fundamentales:

1. **Segmentación**: Particionar la entrada en unidades léxicas válidas.
2. **Clasificación**: Asignar a cada unidad su categoría léxica correspondiente.

Políticas de Reconocimiento

Definición 19 *Maximal Munch*

La política de **maximal munch** (o coincidencia máxima) establece que el analizador léxico debe consumir la cadena más larga posible que coincida con algún patrón léxico. Formalmente, si existen cadenas $w_1, w_2 \in \Sigma^*$ tales que w_1 es prefijo propio de w_2 y ambas son reconocidas por la MDD, entonces el lexer selecciona w_2.

Definición 20 *Retroceso al último estado final*

En caso de que la MDD $\mathcal{D}$ alcance un estado no final $q \notin F$ y no exista transición para el siguiente símbolo, el analizador debe **retroceder** al último estado final visitado, aceptando el lexema correspondiente y reanudando el análisis desde el símbolo siguiente al último consumido.

Ejemplo 5 *Maximal Munch y retroceso*

Considérese la entrada `ifx` con la MDD del ejemplo anterior. El proceso de análisis procede como sigue:

1. Desde q_0, se lee `i` y se transita a q_1.
2. Desde q_1, se lee `f` y se transita a q_2.
3. Desde q_2, se lee `x` y se transita a q_4 (estado final con $\lambda(q_4) = \texttt{ID}$).
4. No hay más símbolos, se acepta el token $\langle \texttt{ID}, \text{“ifx”} \rangle$.

La política de maximal munch garantiza que `ifx` sea reconocido como un único identificador, no como la palabra reservada `if` seguida de `x`.

Construcción del Analizador Léxico

El analizador léxico para IMP se construye mediante la siguiente cadena de transformaciones:

1. **Especificación**: Se definen las expresiones regulares $r_1, \ldots, r_n$ para cada categoría léxica $t_1, \ldots, t_n \in \mathcal{T}$.
2. **Construcción de AFN-ε**: Para cada r_i, se aplica la construcción de Thompson obteniendo AFN-ε $\mathcal{A}_i = (Q_i, \Sigma, \delta_i, q_{0,i}, F_i)$ tal que $L(\mathcal{A}_i) = L(r_i)$.
3. **Unión de autómatas**: Se construye el AFN-ε combinado

$$\mathcal{A} = \bigcup_{i=1}^{n} \mathcal{A}_i$$

   mediante un nuevo estado inicial con ε-transiciones hacia cada $q_{0,i}$.
4. **Eliminación de ε-transiciones**: Se obtiene AFN $\mathcal{A}'$ equivalente sin ε-transiciones.
5. **Determinización**: Se aplica la construcción de subconjuntos para obtener AFD $\mathcal{M}$.
6. **Minimización**: Se aplica el algoritmo de Hopcroft obteniendo AFD_{min} $\mathcal{M}_{\text{min}}$.
7. **Etiquetado**: Se construye la MDD $\mathcal{D} = (\mathcal{M}_{\text{min}}, \lambda)$ asignando etiquetas léxicas a estados finales.
8. **Implementación**: Se implementa la función `lexer` que ejecuta $\mathcal{D}$ con las políticas de maximal munch y retroceso.

Definición 21 *Corrección del analizador léxico*

Sea $\mathcal{L} = \{L(r_1), \ldots, L(r_n)\}$ la familia de lenguajes especificados para cada categoría léxica. El analizador léxico es **correcto** si:

1. **Completitud**: Para toda entrada válida w tal que w puede segmentarse en $w = w_1 \cdots w_k$ con $w_i \in L(r_{j_i})$, el lexer produce la secuencia de tokens correspondiente.
2. **Soundness**: Si el lexer produce una secuencia de tokens $\langle t_1, v_1 \rangle, \ldots, \langle t_k, v_k \rangle$, entonces cada $v_i \in L(r_{j_i})$ donde $t_i = t_{j_i}$.

La corrección del analizador léxico se garantiza mediante la corrección de cada transformación en el pipeline de construcción, en particular:

- La construcción de Thompson preserva el lenguaje reconocido (Teorema de Kleene).

- La eliminación de ε-transiciones preserva el lenguaje.

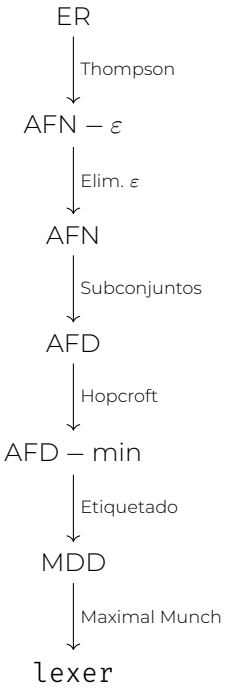
- La construcción de subconjuntos preserva el lenguaje.

- La minimización preserva el lenguaje.

0.4 Metodología

0.4.1 Pipeline de Construcción

La construcción del analizador léxico sigue la cadena de transformaciones fundamentada en el Teorema de Kleene:



Cada transformación preserva el lenguaje reconocido mientras optimiza la estructura para la implementación eficiente.

0.4.2 Especificación Léxica de IMP

La especificación léxica completa de IMP se resume en la tabla siguiente, donde cada categoría léxica $t \in \mathcal{T}$ se asocia con su expresión regular correspondiente:

Categoría	Token	Expresión Regular
Palabras reservadas		
Negación	`NOT`	`not`
Conjunción	`AND`	`and`
Condicional	`IF`	`if`
Rama verdadera	`THEN`	`then`
Rama falsa	`ELSE`	`else`
Comando vacío	`SKIP`	`skip`
Ciclo	`WHILE`	`while`
Cuerpo de ciclo	`DO`	`do`
Booleano verdadero	`TRUE`	`true`
Booleano falso	`FALSE`	`false`
Literales e identificadores		
Enteros	`NUM`	$\texttt{nat} + (- \cdot \texttt{nonzero} \cdot \texttt{digit}^*)$
Identificadores	`ID`	$\texttt{letter} \cdot (\texttt{letter} + \texttt{digit})^*$
Operadores		
Suma	`PLUS`	$+$
Resta	`MINUS`	$-$
Multiplicación	`TIMES`	$*$
Igualdad	`EQ`	$=$
Menor o igual	`LEQ`	$\leq$
Asignación	`ASSIGN`	$:=$
Secuenciación	`SEMICOLON`	;
Delimitadores		
Paréntesis izquierdo	`LPAREN`	(
Paréntesis derecho	`RPAREN`	)
Ignorados		
Espacios	`WHITESPACE`	$\texttt{ws\_char}^+$
Comentarios	`COMMENT`	$\texttt{//} \cdot \texttt{any\_char}^* \cdot \texttt{\n}$

Table 1: Especificación léxica completa de IMP

donde las definiciones auxiliares son:

$$\begin{aligned} \texttt{nonzero} &= [1\text{-}9] \\ \texttt{digit} &= [0\text{-}9] \\ \texttt{nat} &= 0 + \texttt{nonzero} \cdot \texttt{digit}^* \\ \texttt{letter} &= [A\text{-}Z] + [a\text{-}z] \\ \texttt{ws\_char} &= \texttt{␣} + \texttt{\t} + \texttt{\n} \\ \texttt{any\_char} &= [\Sigma_{\text{IMP}} \setminus \{\texttt{\n}\}] \end{aligned}$$

Esta especificación constituye la entrada del sistema de construcción del analizador léxico.

0.4.3 Transformación ER → AFN-ε: Construcción de Thompson

La construcción de Thompson traduce composicionalmente cada expresión regular en un AFN-ε equivalente. El algoritmo es inductivo sobre la estructura de la expresión regular, garantizando que cada autómata resultante tenga exactamente un estado inicial y un estado final.

Teorema 5 ***Construcción de Thompson***

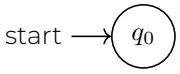
Para toda expresión regular $r \in \mathcal{RE}(\Sigma)$, existe un AFN-$\varepsilon$ $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ tal que $L(\mathcal{A}) = L(r)$ con las siguientes propiedades estructurales:

1. $|F| = 1$ (exactamente un estado final)
2. El estado inicial q_0 no tiene transiciones entrantes
3. El estado final no tiene transiciones salientes
4. El número de estados es lineal en el tamaño de r

Construcción inductiva. Procedemos por inducción estructural sobre r.

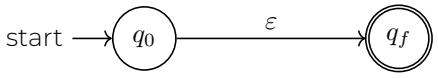
Casos base:

1. Expresión vacía ($\emptyset$): Se construye el autómata



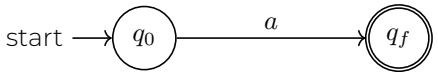
con $Q = \{q_0\}$, $F = \emptyset$, y $\delta(q_0, a) = \emptyset$ para todo $a \in \Sigma \cup \{\varepsilon\}$. Se tiene $L(\mathcal{A}) = \emptyset$.

2. Cadena vacía (ε): Se construye el autómata



con $Q = \{q_0, q_f\}$, $F = \{q_f\}$, y $\delta(q_0, \varepsilon) = \{q_f\}$. Se tiene $L(\mathcal{A}) = \{\varepsilon\}$.

3. Símbolo ($a \in \Sigma$): Se construye el autómata



con $Q = \{q_0, q_f\}$, $F = \{q_f\}$, y $\delta(q_0, a) = \{q_f\}$. Se tiene $L(\mathcal{A}) = \{a\}$.

Casos inductivos:
Sean $r, s \in \mathcal{RE}(\Sigma)$ y supóngase inductivamente que existen AFN-ε

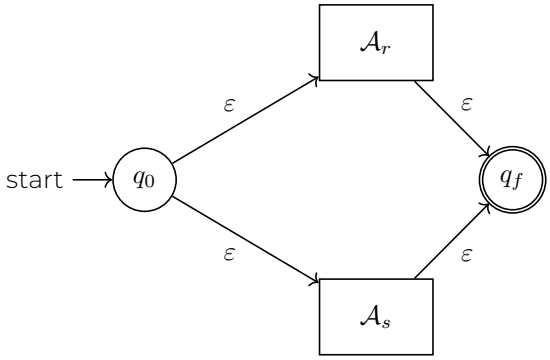
$$\mathcal{A}_r = (Q_r, \Sigma, \delta_r, q_{0,r}, \{q_{f,r}\}) \quad \text{y} \quad \mathcal{A}_s = (Q_s, \Sigma, \delta_s, q_{0,s}, \{q_{f,s}\})$$

tales que $L(\mathcal{A}_r) = L(r)$ y $L(\mathcal{A}_s) = L(s)$, con $Q_r \cap Q_s = \emptyset$.

1. Unión ($r + s$): Se construye $\mathcal{A}_{r+s} = (Q, \Sigma, \delta, q_0, \{q_f\})$ donde:

$$Q = Q_r \cup Q_s \cup \{q_0, q_f\}$$

$$\delta(q, a) = \begin{cases} \{q_{0,r}, q_{0,s}\} & \text{si } q = q_0 \text{ y } a = \varepsilon \\ \delta_r(q, a) & \text{si } q \in Q_r \setminus \{q_{f,r}\} \\ \delta_s(q, a) & \text{si } q \in Q_s \setminus \{q_{f,s}\} \\ \{q_f\} & \text{si } q \in \{q_{f,r}, q_{f,s}\} \text{ y } a = \varepsilon \\ \emptyset & \text{en otro caso} \end{cases}$$



Se verifica que $L(\mathcal{A}_{r+s}) = L(\mathcal{A}_r) \cup L(\mathcal{A}_s) = L(r) \cup L(s) = L(r+s)$.

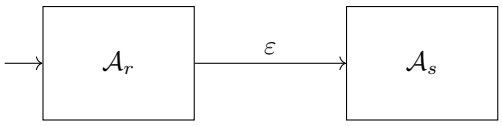
2. Concatenación $(r \cdot s)$: Se construye $\mathcal{A}_{r\cdot s} = (Q, \Sigma, \delta, q_{0,r}, \{q_{f,s}\})$ donde:

$$Q = Q_r \cup Q_s$$

$$\delta(q,a) = \begin{cases} \delta_r(q,a) & \text{si } q \in Q_r \setminus \{q_{f,r}\} \\ \{q_{0,s}\} & \text{si } q = q_{f,r} \text{ y } a = \varepsilon \\ \delta_s(q,a) & \text{si } q \in Q_s \\ \emptyset & \text{en otro caso} \end{cases}$$

Observación crítica sobre estados finales:

- El conjunto de estados finales es $F_{r\cdot s} = F_s$ (únicamente los estados finales de $\mathcal{A}_s$)
- El estado $q_{f,r}$ deja de ser final en la concatenación
- Esto es esencial: una cadena debe atravesar *ambos* autómatas para ser aceptada

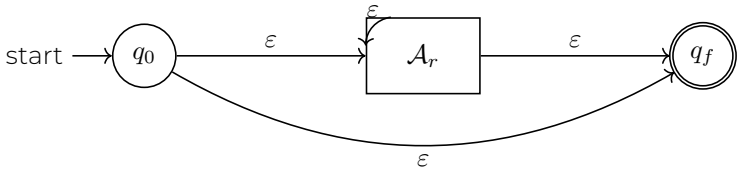


Se verifica que $L(\mathcal{A}_{r\cdot s}) = L(\mathcal{A}_r)L(\mathcal{A}_s) = L(r)L(s) = L(r \cdot s)$.

3. Cerradura de Kleene (r^*): Se construye $\mathcal{A}_{r^*} = (Q, \Sigma, \delta, q_0, \{q_f\})$ donde:

$$Q = Q_r \cup \{q_0, q_f\}$$

$$\delta(q,a) = \begin{cases} \{q_{0,r}, q_f\} & \text{si } q = q_0 \text{ y } a = \varepsilon \\ \delta_r(q,a) & \text{si } q \in Q_r \setminus \{q_{f,r}\} \\ \{q_{0,r}, q_f\} & \text{si } q = q_{f,r} \text{ y } a = \varepsilon \\ \emptyset & \text{en otro caso} \end{cases}$$



Se verifica que $L(\mathcal{A}_{r^*}) = (L(\mathcal{A}_r))^* = (L(r))^* = L(r^*)$.

En todos los casos, las propiedades estructurales se preservan por construcción. □

Ejemplo 6 *Construcción de Thompson para* $(a+b)^*abb$

Aplicamos la construcción de Thompson a la expresión regular $(a+b)^*abb$:

1. Construir autómatas básicos para a y b
2. Aplicar unión: $(a+b)$
3. Aplicar cerradura: $(a+b)^*$
4. Construir autómatas para a, b, b
5. Aplicar concatenaciones sucesivas: $(a+b)^*abb$

El autómata resultante tiene 10 estados y reconoce el lenguaje deseado.

0.4.4 Transformación AFN-$\varepsilon \to$ AFN: Eliminación de ε-transiciones

Definición 22 *ε-cerradura*

Sea $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ un AFN-ε y $S \subseteq Q$. La ε-cerradura de S se define inductivamente:

$$\varepsilon\text{-cl}(S) = \bigcup_{i=0}^{\infty} R^i(S)$$

donde $R^0(S) = S$ y $R^{i+1}(S) = R^i(S) \cup \{q' \mid \exists q \in R^i(S) : q' \in \delta(q, \varepsilon)\}$.
Equivalentemente, $\varepsilon\text{-cl}(S)$ es el menor conjunto tal que:

1. $S \subseteq \varepsilon\text{-cl}(S)$
2. Si $q \in \varepsilon\text{-cl}(S)$ y $p \in \delta(q, \varepsilon)$, entonces $p \in \varepsilon\text{-cl}(S)$

Teorema 6 *Eliminación de ε-transiciones*

Para todo AFN-ε $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ existe un AFN $\mathcal{A}' = (Q, \Sigma, \delta', q_0, F')$ sin ε-transiciones tal que $L(\mathcal{A}) = L(\mathcal{A}')$.

Construcción. Definimos $\mathcal{A}' = (Q, \Sigma, \delta', q_0, F')$ mediante:

$$\delta'(q, a) = \varepsilon\text{-cl}\left(\bigcup_{p \in \varepsilon\text{-cl}(\{q\})} \delta(p, a)\right) \quad \text{para } a \in \Sigma$$

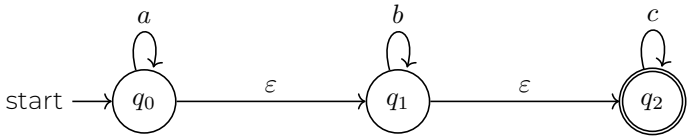
$$F' = \{q \in Q \mid \varepsilon\text{-cl}(\{q\}) \cap F \neq \emptyset\}$$

Intuitivamente, $\delta'(q, a)$ computa los estados alcanzables desde q leyendo a con cualquier cantidad de ε-transiciones antes y después. Los estados finales incluyen aquellos desde los cuales se puede alcanzar un estado final original mediante ε-transiciones.

La equivalencia $L(\mathcal{A}) = L(\mathcal{A}')$ se verifica por inducción sobre la longitud de las cadenas aceptadas. □

Ejemplo 7 Eliminación de ε-transiciones

Considérese el AFN-ε que reconoce $a^*b^*c^*$:



Paso 1: Calcular ε-cerraduras

$$\begin{aligned}\varepsilon\text{-cl}(\{q_0\}) &= \{q_0, q_1, q_2\}\\ \varepsilon\text{-cl}(\{q_1\}) &= \{q_1, q_2\}\\ \varepsilon\text{-cl}(\{q_2\}) &= \{q_2\}\end{aligned}$$

Paso 2: Construir nueva función de transición
A partir de aquí construimos las transiciones correspondientes. Abreviamos ε-cl como EC. La función de transición del AFN resultante se define como:

$$\delta'(q,a) = \varepsilon\text{-cl}\left(\bigcup_{p\in\varepsilon\text{-cl}(\{q\})} \delta(p,a)\right)$$

Calculemos cada transición explícitamente:

$$\begin{aligned}\delta'(q_0,a) &= \text{EC}\left(\delta(q_0,a)\cup\delta(q_1,a)\cup\delta(q_2,a)\right)\\ &= \text{EC}(\{q_0\}\cup\emptyset\cup\emptyset) = \text{EC}(\{q_0\}) = \{q_0,q_1,q_2\}\\ \delta'(q_0,b) &= \text{EC}\left(\delta(q_0,b)\cup\delta(q_1,b)\cup\delta(q_2,b)\right)\\ &= \text{EC}(\emptyset\cup\{q_1\}\cup\emptyset) = \text{EC}(\{q_1\}) = \{q_1,q_2\}\\ \delta'(q_0,c) &= \text{EC}\left(\delta(q_0,c)\cup\delta(q_1,c)\cup\delta(q_2,c)\right)\\ &= \text{EC}(\emptyset\cup\emptyset\cup\{q_2\}) = \text{EC}(\{q_2\}) = \{q_2\}\\ \delta'(q_1,a) &= \text{EC}\left(\delta(q_1,a)\cup\delta(q_2,a)\right)\\ &= \text{EC}(\emptyset\cup\emptyset) = \text{EC}(\emptyset) = \emptyset\\ \delta'(q_1,b) &= \text{EC}\left(\delta(q_1,b)\cup\delta(q_2,b)\right)\\ &= \text{EC}(\{q_1\}\cup\emptyset) = \text{EC}(\{q_1\}) = \{q_1,q_2\}\\ \delta'(q_1,c) &= \text{EC}\left(\delta(q_1,c)\cup\delta(q_2,c)\right)\\ &= \text{EC}(\emptyset\cup\{q_2\}) = \text{EC}(\{q_2\}) = \{q_2\}\\ \delta'(q_2,a) &= \text{EC}(\delta(q_2,a)) = \text{EC}(\emptyset) = \emptyset\\ \delta'(q_2,b) &= \text{EC}(\delta(q_2,b)) = \text{EC}(\emptyset) = \emptyset\\ \delta'(q_2,c) &= \text{EC}(\delta(q_2,c)) = \text{EC}(\{q_2\}) = \{q_2\}\end{aligned}$$

Nota: Observe que aplicamos δ a estados individuales (como está definida), no a conjuntos, y luego tomamos la unión de los resultados sobre todos los estados en la ε-cerradura.
Paso 3: Determinar estados finales
Como $\varepsilon\text{-cl}(\{q_i\}) \cap F \neq \emptyset$ para $i \in \{0,1,2\}$, se tiene $F' = \{q_0, q_1, q_2\}$.

La complejidad temporal del algoritmo es $O(|Q|^2 \cdot |\Sigma|)$ utilizando búsqueda en profundidad para calcular las ε-cerraduras.

0.4.5 Transformación AFN → AFD: Construcción de Subconjuntos

Teorema 7 **Construcción de subconjuntos**

Para todo AFN $\mathcal{N} = (Q_N, \Sigma, \delta_N, q_0, F_N)$ existe un AFD $\mathcal{D} = (Q_D, \Sigma, \delta_D, q_{0,D}, F_D)$ tal que $L(\mathcal{N}) = L(\mathcal{D})$.

Construcción. Construimos $\mathcal{D}$ mediante:

$$\begin{aligned} Q_D &= \{S \subseteq Q_N \mid S \text{ es alcanzable desde } \{q_0\}\} \\ q_{0,D} &= \{q_0\} \\ \delta_D(S,a) &= \bigcup_{q \in S} \delta_N(q,a) \quad \text{para } S \in Q_D, a \in \Sigma \\ F_D &= \{S \in Q_D \mid S \cap F_N \neq \emptyset\} \end{aligned}$$

La idea fundamental es que cada estado del AFD representa el conjunto de estados en los que el AFN podría estar después de leer cierta cadena. El AFD "simula en paralelo" todas las computaciones posibles del AFN.

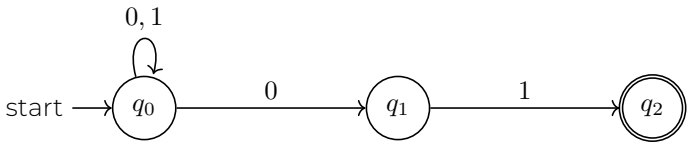
Corrección: Se verifica por inducción sobre la longitud de las cadenas que para toda $w \in \Sigma^*$:

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

Por tanto, $w \in L(\mathcal{D})$ si y solo si $\hat{\delta}_D(\{q_0\}, w) \cap F_N \neq \emptyset$, lo cual ocurre si y solo si $\hat{\delta}_N(q_0, w) \cap F_N \neq \emptyset$, es decir, $w \in L(\mathcal{N})$. □

Ejemplo 8 **Construcción de subconjuntos**

Considérese el AFN que reconoce cadenas sobre $\{0,1\}$ terminadas en 01:

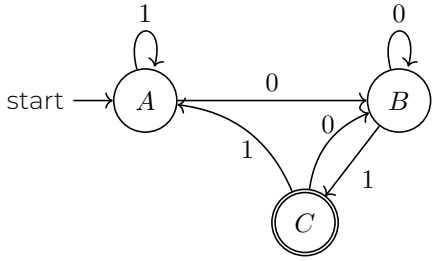


Construcción sistemática:

Estado	$\delta_D(S,0)$	$\delta_D(S,1)$	Final
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$	
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$	
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$	✓

Table 2: Tabla de transiciones del AFD construido

Renombrando estados: $A = \{q_0\}$, $B = \{q_0, q_1\}$, $C = \{q_0, q_2\}$:



La complejidad de la construcción es $O(2^{|Q_N|} \cdot |\Sigma|)$ en el peor caso, aunque en la práctica solo se construyen estados alcanzables, resultando típicamente en $O(|Q_N| \cdot |\Sigma|)$ estados.

0.4.6 Transformación AFD $\rightarrow$ AFD$_{\text{min}}$: Minimización

Definición 23 **Estados k-distinguibles**

Sea $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ un AFD. Dos estados $p, q \in Q$ son **k-distinguibles** si existe $w \in \Sigma^*$ con $|w| \leq k$ tal que:

$$\hat{\delta}(p, w) \in F \iff \hat{\delta}(q, w) \notin F$$

Estados p y q son **distinguibles** si son k-distinguibles para algún $k \geq 0$.

Teorema 8 **Algoritmo de tabla de marcado**

El algoritmo de tabla de marcado computa correctamente la relación de distinguibilidad entre estados en tiempo $O(|Q|^2 \cdot |\Sigma|)$.

Algoritmo. Se construye una tabla triangular para cada par de estados $\{p, q\}$ con $p \neq q$:

Inicialización: Marcar $\{p, q\}$ si $p \in F \iff q \notin F$ (estados 0-distinguibles).

Iteración: Repetir hasta que no haya cambios:

$$\text{Si } \exists a \in \Sigma : \{\delta(p, a), \delta(q, a)\} \text{ está marcado, entonces marcar } \{p, q\}$$

Terminación: Estados p y q son equivalentes ($p \sim q$) si y solo si $\{p, q\}$ no está marcado.

La relación $\sim$ es de equivalencia y particiona Q en clases de equivalencia. □

Definición 24 **AFD cociente**

Sea $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ un AFD y $\sim$ la relación de equivalencia de estados. El **AFD cociente** $\mathcal{M}/\sim$ se define como:

$$\mathcal{M}/\sim = (Q/\sim, \Sigma, \delta', [q_0], F')$$

donde:

$$\delta'([q], a) = [\delta(q, a)]$$
$$F' = \{[q] \mid q \in F\}$$

El AFD cociente es el AFD mínimo único (salvo isomorfismo) que reconoce $L(\mathcal{M})$.

Ejemplo 9 *Minimización de AFD*

Considérese el AFD que reconoce $\{a,b\}^*\{a,b\}$:

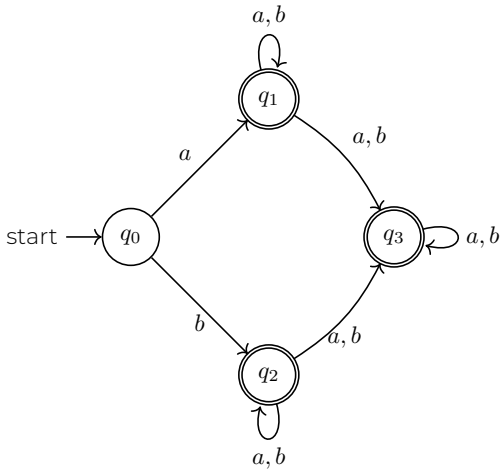
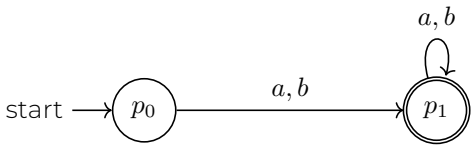


Tabla de marcado:

	q_0	q_1	q_2	q_3
q_1	×			
q_2	×			
q_3	×			

Inicialmente marcamos $\{q_0,q_1\}$, $\{q_0,q_2\}$, $\{q_0,q_3\}$ porque $q_0 \notin F$ mientras que $q_1,q_2,q_3 \in F$.
Verificando $\{q_1,q_2\}$: Para todo $a \in \{a,b\}$, $\delta(q_1,a) = q_3$ y $\delta(q_2,a) = q_3$, ambos no marcados. Por tanto, $q_1 \sim q_2$.
Verificando $\{q_1,q_3\}$ y $\{q_2,q_3\}$: Ambos pares tienen el mismo comportamiento y permanecen sin marcar, por lo que $q_1 \sim q_2 \sim q_3$.
AFD minimizado:



donde $p_0 = [q_0]$ y $p_1 = [q_1] = [q_2] = [q_3]$.

0.4.7 Transformación $\text{AFD}_{\text{min}} \rightarrow$ MDD

Definición 25 *Máquina Discriminadora Determinista*

Una MDD es una séxtupla $\mathcal{D} = (Q,\Sigma,\delta,q_0,F,\lambda)$ donde (Q,Σ,δ,q_0,F) es un AFD y $\lambda : F \rightarrow \mathcal{T}$ asigna a cada estado final una categoría léxica del conjunto $\mathcal{T}$ de tipos de tokens.

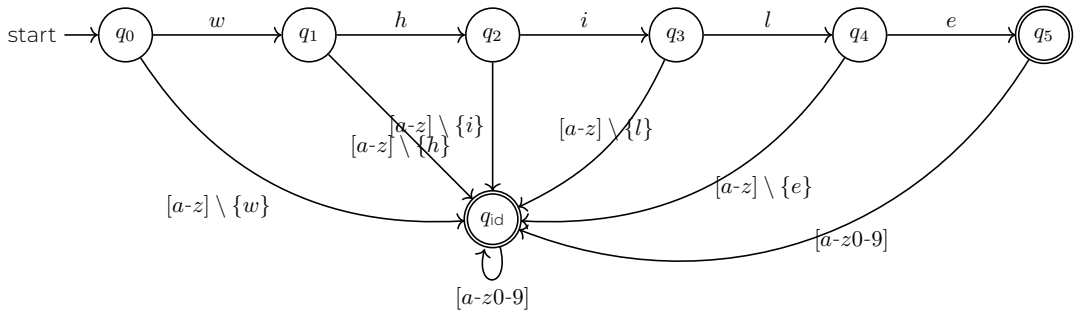
La construcción de la MDD para IMP procede agregando etiquetas a los estados finales del AFD minimizado según la especificación léxica:

1. Para cada categoría $t \in \mathcal{T}$, el estado final que reconoce el patrón correspondiente recibe la etiqueta $\lambda(q) = t$.
2. En caso de ambigüedad (múltiples patrones coinciden), se establece un orden de prioridad:

palabras reservadas > identificadores > operadores > delimitadores

Ejemplo 10 *MDD para palabras reservadas e identificadores*

La palabra reservada `while` y los identificadores comparten prefijos. La MDD resultante discrimina mediante estados finales etiquetados:



con $\lambda(q_5) = \texttt{WHILE}$ (con mayor prioridad) y $\lambda(q_{\text{id}}) = \texttt{ID}$.

0.4.8 Política de Maximal Munch y Retroceso

Definición 26 *Maximal Munch*

La política de **maximal munch** establece que el analizador léxico debe reconocer el lexema más largo posible que coincida con algún patrón válido. Formalmente, si $w = uv$ con $u, uv \in L(\mathcal{D})$, el lexer selecciona uv en lugar de u.

Definición 27 *Algoritmo de retroceso*

Sea $\mathcal{D}$ una MDD y $w = a_1 \cdots a_n$ la cadena de entrada. El analizador mantiene:

- q_{actual}: estado actual
- $q_{\text{último\_final}}$: último estado final visitado
- $i_{\text{última\_posición}}$: posición del último estado final

Cuando no existe transición para a_{i+1} desde q_{actual}:

- Si $q_{\text{último\_final}}$ existe: retroceder a $i_{\text{última\_posición}}$, emitir token $\langle \lambda(q_{\text{último\_final}}), a_1 \cdots a_{i_{\text{última\_posición}}} \rangle$
- Si no: error léxico en posición i

Ejemplo 11 Maximal munch en acción

Entrada: `whilexyz`
Con patrones `while` → `WHILE` y $[a\text{-}z]^+$, → `ID`:

1. Leer `w`, `h`, `i`, `l`, `e` → alcanza estado final `WHILE`
2. Leer `x` → transita a estado `ID`
3. Leer `y`, `z` → permanece en `ID`
4. Fin de entrada → emitir ⟨`ID`, "whilexyz"⟩

El resultado es un único token `ID`, no `WHILE` seguido de error o de otro token.

0.4.9 Implementación de la Función Lexer

La función `lexer` implementa la MDD con las políticas de maximal munch y retroceso:

```haskell
type Token = (TokenType, String)

lexer :: String → Either LexError [Token]
lexer = go (1, 1) []
  where
    go :: (Int, Int) → [Token] → String → Either LexError [Token]
    go _ tokens [] = Right (reverse tokens)
    go pos tokens input =
      case maximalMunch mdd input pos of
        Just (tok, rest, newPos) →
          go newPos (tok : tokens) rest
        Nothing →
          Left $ LexError "Invalid token" pos

maximalMunch :: MDD → String → (Int, Int)
             → Maybe (Token, String, (Int, Int))
maximalMunch mdd input pos =
  runMDD mdd input pos Nothing 0
  where
    runMDD :: MDD → String → (Int, Int)
           → Maybe (State, Int) → Int
           → Maybe (Token, String, (Int, Int))
    runMDD _ [] _ lastFinal consumed =
      mkToken lastFinal consumed input
    runMDD mdd (c:cs) pos lastFinal consumed =
      case transition mdd currentState c of
        Just nextState →
          let newLast = if isFinal nextState
                        then Just (nextState, consumed + 1)
                        else lastFinal
          in runMDD mdd cs (updatePos pos c) newLast (consumed + 1)
        Nothing →
          mkToken lastFinal consumed input
```

donde `mkToken` construye el token a partir del último estado final visitado, retrocediendo si es necesario, y `updatePos` actualiza la posición (línea, columna) considerando saltos de línea.

Bibliography

[1] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Pearson, 2006.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education, 2006.

[3] M. S. Romero. *Compiladores: Unidad 2: Análisis Léxico*. Facultad de Ciencias, UNAM, 2026-1. https://lambdasspace.github.io/CMP/notas/cmp_n08.pdf

[4] Michael Sipser. *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning, 2012.

[5] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[6] Tobias Nipkow, Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014.