Un análisis IMPortante

Arias Villarroel, Alejandra Valentina
del Valle Vera, Nancy Elena
Martínez Hidalgo, Paola Mildred
Martínez Mejía, Eduardo
Sánchez Victoria, Leslie Paola
Facultad de Ciencias, UNAM

Resumen
Para diseñar e implementar un analizador léxico es necesario aplicar conceptos de la teoría de autómatas y lenguajes formales. Este documento tiene como propósito documentar el proceso de desarrollo necesario para transformar una expresión regular en una máquina miscriminadora determinista, que permitirá la implementación de un programa que se encargue de dividir un texto de entrada en una secuencia de tokens.

Índice

1. Introducción

El trabajo desarrollado se enfoca en la implementación de un analizador léxico, correspondiente a una de las fases del proceso de traducción en un compilador. La introducción tiene como objetivo presentar los fundamentos teóricos necesarios para comprender el funcionamiento general de un compilador y contextualizar el desarrollo del analizador léxico.

1.1. El compilador

Un *compilador* es un programa que traduce el código fuente de un lenguaje de programación a un programa equivalente escrito en código máquina, código intermedio u otro lenguaje de programación.

Su objetivo principal es transformar un programa escrito en un lenguaje complejo, pero comprensible para los humanos, en un formato más simple que la computadora pueda leer, interpretar y ejecutar. Además, debe asegurarse de que la lógica del código de salida coincida con la del código de entrada y que no se pierda nada al traducir el código[1].

El proceso de traducción consta de dos partes: análisis y síntesis. El análisis se encarga de entender el

código fuente para extraer su estructura y significado, generando así una representación intermedia. Por otra parte, la síntesis toma dicha representación para producir el código objetivo equivalente que puede ser ejecutado por una máquina.

Cada parte se divide en una secuencia de fases, en las que cada una transforma una representación del programa fuente en otra[2].

Etapa de análisis

En el **análisis léxico** el código fuente se divide en lexemas, que son fragmentos individuales que corresponden a patrones específicos del lenguaje[1]. Estos fragmentos se agrupan en secuencias de caracteres significativas conocidos como *componentes léxicos* o *tokens* que representan las palabras clave, identificadores, operadores y otros elementos del lenguaje de programación.

En el **análisis sintáctico** los *tokens* se procesan para verificar que la sintaxis del código sea correcta, generalmente mediante la construcción de un árbol sintáctico que representa la estructura lógica de los elementos en el programa[1].

A partir del árbol de sintaxis, en el **análisis semántico** el compilador valida que el significado del código sea consistente con con las reglas del lenguaje. Esto implica verificar que las operaciones y construcciones sean correctas en el contexto del programa[1].

Etapa de síntesis

Una vez completada la etapa de análisis, se **genera un código intermedio**. Este código debe ser portable, es decir, debe ser independiente de la arquitectura de la máquina y debe preservar toda la funcionalidad del programa original.

En este punto el compilador puede realizar **optimizaciones** al código intermedio con el fin de mejorar la eficiencia del código objetivo[2].

Finalmente, el código intermedio optimizado se traduce al lenguaje objetivo, **generando el código** máquina que será ejecutado directamente por la computadora.

1.2. Expresiones regulares

Las expresiones regulares son una notación algebraica para definir lenguajes que toman una perspectiva declarativa[3]. Proporcionan una forma eficaz de especificar patrones y sientan las bases para el reconocimiento de los *tokens*[4].

Definición 1.1 (Expresiones regulares)**.** Sea Σ un alfabeto finito. Las expresiones regulares, abreviado ER, sobre Σ se definen inductivamente de la siguiente manera[5]:

1. $\emptyset$ es una ER.
2. ϵ es una ER.
3. Cualquier $a \in \Sigma$ es una ER.
4. Si R y S son ER, entonces
   a) $(R \cdot S)$ es una ER.
   b) $(R + S)$ es una ER.
   c) (R^*) es una ER.
5. Solo las expresiones formadas con estas reglas son expresiones regulares.

Los lenguajes generados por expresiones regulares son llamados lenguajes regulares[6].

Teorema 1.1. *Un lenguaje es regular si y solo si una expresión regular lo describe.*

1.3. Autómatas Finitos

Los autómatas finitos son máquinas teóricas que se utilizan para reconocer patrones descritos por expresiones regulares. Tienen un conjunto de estados y su *control* pasa de un estado a otro en respuesta a las *entradas externas*[7]. De esta forma, al recibir una cadena de entrada, la aceptan o rechazan según provenga del lenguaje o no.

En otras palabras, los autómatas finitos sirven como abstracciones del código que se requiere para implementar las expresiones regulares[8].

Definición 1.2. Un *autómata finito* (AF) se define

como una 5-tupla

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

donde:

$$\begin{array}{rcl} Q & : & \text{conjunto finito de estados,} \\ \Sigma & : & \text{alfabeto finito (símbolos de entrada),} \\ \delta & : & \text{función de transición,} \\ q_0 & \in & Q \text{ es el estado inicial,} \\ F & \subseteq & Q \text{ conjunto de estados de aceptación.} \end{array}$$

1.3.1. Autómata Finito Determinista

Son aquellos que sólo pueden estar en un único estado después de leer cualquier secuencia de entrada, es decir, para cualquier estado determinado y cualquier símbolo leído, la función de transición devuelve exactamente un estado al cual ir.

Definición 1.3. Un autómata finito determinista, abreviado AFD, es un autómata finito tal que:

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

es la función de transición.

1.3.2. Autómata Finito No Determinista

Tienen la capacidad de estar en varios estados a la vez, lo que les da la posibilidad de conjeturar algo acerca de su entrada, ya que, con cada símbolo leído, su función de transición devuelve un conjunto de cero, uno o más estados.

Definición 1.4. Un autómata finito no determinista, abreviado AFN, es un autómata finito tal que

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

es la función de transición y $\mathcal{P}(Q)$ es un conjunto de posibles estados siguientes.

Serán de utilidad pues nos permitirán *programar* soluciones para los problemas utilizando un lenguaje de programación de alto nivel, ya que este tipo de autómatas pueden ser transformados a un autómata finito determinista que puede *ejecutarse* en una computadora convencional[7].

Definición 1.5. Un autómata finito no determinista con transiciones épsilon, abreviado AFN-ε, es un autómata finito tal que

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

es la función de transición.

Los siguientes teoremas[5] nos permiten justificar formalmente el uso de expresiones regulares y autómatas finitos para reconocer los componentes léxicos del lenguaje IMP.

Teorema 1.2 (Teorema de equivalencia)**.** *Dado un autómata finito determinista existe un autómata finito no determinista equivalente, y viceversa.*
Teorema 1.3. *Sea R una expresión regular. Entonces existe un autómata finito no determinista con transiciones épsilon que acepta L(R).*
Teorema 1.4. *Si un lenguaje L es aceptado por un AFN-ε entonces L es aceptado por un AFD sin transiciones épsilon y viceversa.*
Teorema 1.5. *Sea R una expresión regular. Entonces existe un autómata finito no determinista que acepta L(R).*

1.4. Máquina Discrimindaora Determinista

El objetivo del análisis léxico es implementar un programa, denominado lexer[9;10], que se encargue de dividir un texto de entrada en una secuencia de tokens. Su construcción se basa en un autómata finito determinista, diseñado para reconocer los patrones léxicos de un lenguaje.

Para resolver ambigüedades, comunmente se adopta la política de *maximal munch* que consiste en seleccionar el prefijo más largo de la entrada que constituya un token válido[11]. Cuando múltiples definiciones de tokens coinciden con una misma secuencia de caracteres, se utiliza una convención adicional conocida como *longest match*[12], que da preferencia al token definido en primer lugar. Como el enfoque de seleccionar el token con la mayor cantidad de caracteres puede fallar, es necesario realizar un retroceso hasta la posición del último token reconocido y continuar el análisis desde ahí[9].

La máquina de estados utilizada en esta etapa recibe el nombre de Máquina Discriminadora Determinista[13].

2. Metodología

Esta sección tiene como propósito presentar el fundamento teórico que respalda la implementación de un analizador léxico, abarcando desde la definición formal de la gramática del lenguaje IMP hasta la construcción del autómata finito determinista mínimo, cuya función es reconocer el lenguaje definido por la unión de las expresiones regulares asociadas a las distintas categorías léxicas.

2.1. El lenguaje IMP

Considérese la siguiente gramática definida para el lenguaje IMP. Permite manipular números, valores de verdad, expresiones aritméticas y booleanas, asignaciones, estructuras de control y listas.

```
Aexp ::= n
       | x
       | Aexp + Aexp
       | Aexp - Aexp

Bexp ::= true
       | false
       | Aexp = Aexp
       | Aexp ≤ Aexp
       | not Bexp
       | Bexp and Bexp

Asig ::= x := Aexp

Com ::= skip
      | x := Aexp
      | Com ; Com
      | if Bexp then Com else Com
      | while Bexp do Com
      | for Asig in Aexp do Com end

List ::= []
       | Aexp : List
       | Bexp : List
```

2.2. Categorías léxicas

Se reconocen las siguientes categorías léxicas para el lenguaje IMP.

1. Identificadores
2. Números: De tipo entero.
3. Operadores aritméticos: $+, -$
4. Operadores relacionales: $=, \leq$, `not, and`
5. Palabras reservadas: `if, then, else, while, do, for, in, end`
6. Símbolos especiales: `;`, `:`, `:=`, `[`, `]`

2.3. Expresiones regulares

Identificadores

Son una secuencia de caracteres que nombran variables dentro del programa. Un identificador debe comenzar con una letra del abecedario y puede opcionalmente terminar con un único dígito.

$$(\texttt{a}+..+\texttt{z}+\texttt{A}+..+\texttt{Z})\cdot(\texttt{a}+..+\texttt{z}+\texttt{A}+..+\texttt{Z})^*\cdot(\epsilon+\texttt{1}+...+\texttt{9})$$

Números

Son todo el conjunto de los números enteros.

$$\texttt{0}+(\texttt{-}+\epsilon)\cdot(\texttt{1}+...+\texttt{9})\cdot(\texttt{0}+...+\texttt{9})^*$$

Operadores aritméticos

Son los símbolos que se utilizan para realizar cálculos matemáticos.

$$(\texttt{+}+\texttt{-})$$

Operadores relacionales

Son los símbolos que se utilizan para comparar dos valores o expresiones. El resultado de una comparación debe ser un valor de verdad.

$$(\texttt{=}+\leq+\texttt{n}\cdot\texttt{o}\cdot\texttt{t}+\texttt{a}\cdot\texttt{n}\cdot\texttt{d})$$

Palabras reservadas

Son términos que tienen un significado predefinido por el lenguaje y no pueden ser utilizados como nombres de variables.

$$(\texttt{s}\cdot\texttt{k}\cdot\texttt{i}\cdot\texttt{p}+\texttt{i}\cdot\texttt{f}+\texttt{t}\cdot\texttt{h}\cdot\texttt{e}\cdot\texttt{n}+\texttt{e}\cdot\texttt{l}\cdot\texttt{s}\cdot\texttt{e}+\texttt{w}\cdot\texttt{h}\cdot\texttt{i}\cdot\texttt{l}\cdot\texttt{e}$$

$$+\texttt{d}\cdot\texttt{o}+\texttt{f}\cdot\texttt{o}\cdot\texttt{r}+\texttt{i}\cdot\texttt{n}+\texttt{e}\cdot\texttt{n}\cdot\texttt{d}+\texttt{t}\cdot\texttt{r}\cdot\texttt{u}\cdot\texttt{e}+\texttt{f}\cdot\texttt{a}\cdot\texttt{l}\cdot\texttt{s}\cdot\texttt{e})$$

Símbolos especiales

Son caracteres que permiten definir y organizar la estructura del programa.

$$(\texttt{;}+(\texttt{:}\cdot\texttt{=})+\texttt{:})$$

Listas

Es un caso especial que permite hacer uso de listas en el lenguaje.

$$(\texttt{[}\cdot\texttt{]})$$

2.4. Autómata Finito No Determinista con transiciones épsilon

El Teorema 1.5 nos permite hacer uso del algoritmo de construcción de Thompson para generar autómatas a partir de expresiones regulares[5].

Algoritmo 2.1. Sea Σ un alfabeto finito y R una expresión regular.

Para todo símbolo en R, se construye un sub-autómata $M = (Q, \Sigma, \delta, q_0, q_f)$ siguiendo:

1. Si $R_i = \emptyset$:

$$\begin{aligned} Q &= \{q_0, q_f\} \\ \Sigma &= \emptyset \\ \delta &= \{\delta(q_0, a) = \emptyset\}, \quad \forall a \in \Sigma \end{aligned}$$

2. Si $R_i = \varepsilon$:

$$\begin{aligned} Q &= \{q_0, q_f\} \\ \Sigma &= \{\varepsilon\} \\ \delta &= \{\delta(q_0, \varepsilon) = \{q_f\}\} \end{aligned}$$

3. Si $R_i = a$, con $a \in \Sigma$:

$$\begin{aligned} Q &= \{q_0, q_f\} \\ \Sigma &= \{a\} \\ \delta &= \{\delta(q_0, a) = \{q_f\}\} \end{aligned}$$

Iterativamente, para cada par de sub-autómatas $M_i = (Q_1, \Sigma_1, \delta_1, q_1, q_{f_1})$ y $M_j = (Q_2, \Sigma_2, \delta_2, q_2, q_{f_2})$ correspondientes a las subexpresiones R_i y R_j, se construye:

1. Si $R = R_i \cdot R_j$ (concatenación):

$$\begin{aligned} Q &= Q_1 \cup Q_2 \\ \Sigma &= \Sigma_1 \cup \Sigma_2 \\ \delta &= \delta_1 \cup \delta_2 \cup \{\delta(q_{f_1}, \varepsilon) = \{q_2\}\} \\ q_0 &= q_1 \\ q_f &= q_{f_2} \end{aligned}$$

2. Si $R = R_i + R_j$ (unión):

$$\begin{aligned} Q &= Q_1 \cup Q_2 \cup \{q_0, q_f\} \\ \Sigma &= \Sigma_1 \cup \Sigma_2 \\ \delta &= \delta_1 \cup \delta_2 \cup \{\delta(q_0, \varepsilon) = \{q_1, q_2\}, \\ &\qquad \delta(q_{f_1}, \varepsilon) = \{q_f\}, \\ &\qquad \delta(q_{f_2}, \varepsilon) = \{q_f\}\} \\ q_0 &= q_0 \\ q_f &= q_f \end{aligned}$$

3. Si $R = R_i^*$ (estrella de Kleene):

$$\begin{aligned} Q &= Q_1 \cup \{q_0, q_f\} \\ \Sigma &= \Sigma_1 \cup \{\varepsilon\} \\ \delta &= \delta_1 \cup \{\delta(q_0, \varepsilon) = \{q_1, q_f\}, \\ &\qquad \delta(q_{f_1}, \varepsilon) = \{q_1, q_f\}\} \\ q_0 &= q_0 \\ q_f &= q_f \end{aligned}$$

2.5. Autómata Finito No Determinista

En este punto, al AFN-ε obtenido se le deben eliminar las transiciones épsilon para obtener un AFN equivalente.

No existe un algoritmo completamente independiente para eliminar transiciones épsilon, en cambio, este procedimiento suele integrarse con otros algoritmos, como el de conversión de un AFN a un AFD[14].

No obstante, con fines expositivos, esta sección puede considerarse como la primera parte del algoritmo de conversión de un AFN a AFD, inducido por la demostración del Teorema 1.4[5].

Definición 2.1. Una ε-cerradura(q) denota todos los estados p tales que existe una transición de q a p que consume la cadena vacía ε.

Definición 2.2. Si P es un conjunto de estados, entonces

$$\varepsilon - \text{cerradura}(P) = \bigcup_{p \in P} \varepsilon - \text{cerradura}(p)$$

Definición 2.3. La función de transición δ extendida para AFN-ε, denotada $\hat{\delta}$, transforma $\delta(q, w)$ a un subconjunto de estados Q. Se define $\hat{\delta}$ como sigue.

1. $\hat{\delta}(q, \epsilon) = \epsilon - cerradura(q)$
2. Para $w \in \Sigma^*$ y $a \in \Sigma$

   $\hat{\delta}(q, wa) = \epsilon - cerradura(P)$, donde

   $P = \{p | \text{para alguna } r \in \hat{\delta}(q, w), p \in \delta(r, a)\}$

Además se extienden δ y $\hat{\delta}$ a conjuntos de estados mediante:

$$\delta(R, a) = \bigcup_{q \in R} \delta(q, a)$$

$$\hat{\delta}(R, a) = \bigcup_{q \in R} \hat{\delta}(q, a)$$

Algoritmo 2.2. Sea $E = (Q_E, \Sigma, \delta_E, q_E, F_E)$ un AFN-ε. Se construye un autómata finito determinista $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ mediante los siguientes pasos:

1. Se define el conjunto de estados del AFD como el conjunto potencia de los estados del AFN-ε:

   $$Q_D = \wp(Q_E)$$

2. El estado inicial del AFD es el ε-cerradura del estado inicial del AFN:

   $$q_D = \varepsilon\text{-cerradura}(q_E)$$

3. La función de transición se define para cada conjunto de estados $P \subseteq Q_E$ y cada símbolo $a \in \Sigma$ como:

   $$\delta_D(P, a) = \varepsilon\text{-cerradura}\left(\bigcup_{p \in P} \hat{\delta}_E(p, a)\right)$$

4. El conjunto de estados finales del AFD está formado por todos aquellos subconjuntos que contienen al menos un estado final del AFN-ε:

   $$F_D = \{S \subseteq Q_D \mid S \cap F_E \neq \emptyset\}$$

2.6. Autómata Finito Determinista

A continuación, se presenta el paso final para obtener un AFD una vez eliminadas las transiciones épsilon.[7].

Algoritmo 2.3. Para cada $S \in Q_D$

1. Sea $S = \{p_1, \ldots, p_k\}$.
2. Para cada símbolo $a \in \Sigma$, obtener:

   $$\delta_D(S, a) = \varepsilon\text{-cerradura}\left(\bigcup_{i=1}^{k} \delta_E(p_i, a)\right)$$

2.7. Autómata Finito Determinista Mínimo

Una vez obtenido el AFD, resulta conveniente minimizarlo con el fin de reducir la complejidad espacial y computacional durante su implementación.

Cabe señalar que, desde el punto de vista teórico, el AFD no minimizado ya es capaz de reconocer los componentes léxicos de IMP.

Definición 2.4. Sean $p, q \in Q$ dos estados de un autómata finito determinista. Se dice que p y q son *equivalentes*, denotado $p \approx q$, si y solo si

$$\forall w \in \Sigma^* \left(\hat{\delta}(p, w) \in F \iff \hat{\delta}(q, w) \in F\right)$$

Esta relación $\approx$ es una relación de equivalencia.

Definición 2.5. Dado un estado $p \in Q$, su clase de equivalencia bajo la relación $\approx$ se denota por $[p]$ y se define como:

$$[p] = \{q \in Q \mid q \approx p\}$$

Cada estado $p \in Q$ pertenece exactamente a una clase de equivalencia, y se cumple que:

$$p \approx q \iff [p] = [q]$$

El siguiente algoritmo permite determinar qué pares de estados son equivalentes[15].

Algoritmo 2.4. Sea $M = (Q, \Sigma, \delta, s, F)$ un AFD sin estados inaccesibles.

1. Escribir una tabla con todos los pares no ordenados de estados $\{p, q\}$, inicialmente sin marcar.
2. Marcar el par $\{p, q\}$ si $p \in F$ y $q \notin F$, o viceversa.
3. Repitir el siguiente paso hasta que no haya más cambios:
   - Si existe un par $\{p, q\}$ no marcado tal que para algún símbolo $a \in \Sigma$, el par $\{\delta(p, a), \delta(q, a)\}$ ya está marcado, entonces marcar también el par $\{p, q\}$.
4. Al finalizar, se cumple que:

$$p \approx q \iff \{p, q\} \text{ no está marcado.}$$

Definición 2.6. Sea $M = (Q, \Sigma, \delta, s, F)$ un autómata finito determinista. El autómata mínimo asociado, denotado $M/\approx$, se define como:

$$M/\approx = (Q', \Sigma, \delta', s', F')$$

donde

$$\begin{aligned} Q' &= \{[p] \mid p \in Q\} \\ \delta'([p], a) &= [\delta(p, a)] \\ s' &= [s] \\ F' &= \{[p] \mid p \in F\} \end{aligned}$$

Además, si $p \approx q$, entonces $\delta(p, a) \approx \delta(q, a)$ para todo $a \in \Sigma$.

El siguiente teorema[16] nos permite hacer uso del AFD mínimo para construir la Máquina Discriminadora Determinista.

Teorema 2.1. *Sea L un lenguaje regular sobre un alfabeto Σ. Entonces existe un AFD que acepta L y que tiene exactamente n estados, donde n es el número de clases de equivalencia de $\approx_L$. Cualquier otro AFD que acepte L debe tener más estados que M, o debe ser equivalente a M, excepto por los nombres de los estados.*

3. Implementación

Una vez comprendidos los fundamentos teóricos y el proceso necesario para transformar una expresión regular en un autómata finito determinista, el siguiente paso consiste en implementar cada una de las etapas descritas en un lenguaje de programación, en este caso Haskell.

3.1. Expresiones regulares

La entrada del programa consiste en el conjunto de expresiones regulares que describen las cadenas pertenecientes a las categorías léxicas definidas en la sección 2.2.

Para construir el AFD que reconozca el lenguaje de todas las categorías léxicas, es necesario unir disjuntamente cada una de las expresiones regulares. De este modo, se obtiene una única expresión regular compuesta que representa la unión de los lenguajes reconocidos por cada categoría. Esta expresión será la base para generar el AFD equivalente, a partir del cual se podrá realizar el análisis léxico de cualquier cadena de entrada.

Representamos una expresión regular en Haskell definiendo el siguiente tipo de dato.

```
data RegEx = Empty
           | Epsilon
           | Character Char
           | Union RegEx RegEx
           | Concat RegEx RegEx
           | Kleene RegEx
```

3.2. Autómatas Finitos

AFN-ϵ

La representación de un autómata finito no determinista con transiciones épilon se define como:

```
type State = Int
type Symbol = Maybe Char
type Delta = (State, Symbol, State)

data NFAE = NFAE {
    states      :: Set State,
    alphabet    :: Set Symbol,
    transitions :: Set Delta,
    start       :: State,
    final       :: State
}
```

El tipo `Symbol` utiliza la mónada `Maybe` para representar la cadena vacía como `Nothing` y un caracter `c` como `Just c`.

La función principal `toNFAE :: RegEx ->NFAE` convierte el dato `RegEx` en un `NFAE` utilizando la función `thompson :: RegEx ->[State] ->NFAE` que implementa el algoritmo de construcción de Thompson.

AFN

La representación de un autómata finito no determinista se define de la siguiente manera:

```
type DeltaNoE = (State, Char, [State])

data NFA = NFA {
    statesNFA      :: Set State,
    alphabetNFA    :: Set Char,
    transitionsNFA :: Set DeltaNoE,
    startNFA       :: State,
    finalNFA       :: [State]
}
```

Se deja de utilizar el tipo `Maybe Char` para representar los caracteres del alfabeto, ya que el objetivo es eliminar todas las transiciones con la cadena vacía. Asimismo, se permite la existencia de múltiples estados finales, los cuales se representan mediante una lista de estados `[State]`, en lugar de un único estado final.

La función principal `toNFA :: NFAE ->NFA` se encarga de transformar un $AFN - \epsilon$ en un AFN equivalente. Para ello se utiliza como función auxiliar `nfaEdeltaHat :: NFAE ->[(State, Maybe Char, Set State)]`, la cual, para cada estado q y símbolo del alfabeto x, devuelve una lista de tuplas de la forma `[(q, x,` $\hat{\delta}(q,x)$`)]`.

A partir de esta lista, se obtienen los estados del nuevo autómata uniendo el estado inicial del `NFAE` con los estados alcanzables indicados por el tercer elemento en cada tupla.

Para obtener la lista de los estados finales del `NFA` se comprueba si el estado final del `NFAE` pertenece a la $\epsilon - cerradura$ de un estado alcanzable q, si esto sucede entonces se añade q a la lista de nuevos estados finales.

Para obtener las transiciones, se recorre la lista generada por la función `nfaEdeltaHat`. Si el estado q pertenece a los estados alcanzables y la función $\hat{\delta}$ existe para algún símbolo del alfabeto, entonces se agrega la tupla a las transiciones del `NFA` .

AFD

La representación de un autómata finito determinista se define de la siguiente manera:

```
type DeltaDFA = (State, Char, State)

data DFA = DFA {
    states      :: Set State,
    alphabet    :: Set Char,
    transitions :: Set DeltaDFA,
    start       :: State,
    final       :: [State]
}
```

Similar al AFN se permite la existencia de múltiples estados finales.

La función principal `toDFA:: NFA ->DFA` se encarga de transformar un `AFN` a un `AFD` equivalente, donde se asume que el `AFN` no tiene $\epsilon - transiciones$.

Para esto se usan algunas funciones auxiliares:

`startSet = Set.singleton (startNFA nfa)` nos indica que el estado inicial del `AFN` se mantiene.

`move :: Set State ->Char ->Set State` calcula todos los estados alcanzables desde un conjunto de estados con un símbolo dado.

`build :: [Set State] ->Set (Set State) -> Map.Map (Set State, Char) (Set State) ->(Set (Set State), Map.Map (Set State, Char) (Set State))` en esta función tenemos como parámetros la cola de estados por procesar, estados visitados y mapa de transiciones acumuladas, su función es construir los estados del `AFD`,con `move` calcula hacia dónde se mueve el estado actual, identifica los estados que no han sido visitados y actualiza las transiciones.

`stateIDs :: Map.Map (Set State) State` renombra a los conjuntos de estados del `AFN` asignando un número único.

`deltaDFA = Set.fromLis` genera las transicionnes del `DFA`.

`finals` calcula los estados finales y contruye el nuevo autómata.

`deltaHat :: DFA ->State ->Char ->Maybe State` proporciona datos sobre el comportamiento de los estados y esto nos permite implementar la verificación de equivalencia entre estados necesaria para la minimización del `AFD`.

AFD mínimo

Para obtener el autómata finito determinista mínimo, se emplea la función `minimize :: DFA ->DFA`, la cuál toma como entrada el AFD original y devuelve otro equivalente, pero con un número menor o igual de estados.

La función `nonEquivalentStates :: DFA ->[(State, State)]` ejecuta los dos pasos principales del algoritmo de minimizacion, devolviendo la lista de todos los estados no equivalentes.

La función `step1 :: DFA ->[State] ->[(State, State)] ->[(State, State)]` obtiene todos los pares triviales no equivalentes $\{(q, q') \mid q \notin F \cap q' \in F\}$.

La función `step2 :: DFA ->[(State, State)] ->[(State, State)]` toma la lista resultante de `step1` y agrega todas las parejas de estados tales que con una misma entrada, transitan a un par de estados existente en la lista.

Finalmente, teniendo todos los pares de estados no equivalentes, la función `equivalenceClasses :: DFA ->Set (Set State)` devuelve los estados en conjuntos de su respectiva clase. Estos son mapeados a un solo estado representante de cada clase, que se convertirán en los estados oficiales del AFD minimo, con solo las funciones de transicion correspondientes a estos.

3.3. Máquina Discriminadora Determinista

A partir del conjunto de expresiones regulares proporcionado como entrada, ya es posible construir el autómata finito determinista mínimo equivalente. Sobre este autómata se define la función `lexerIMP`, encargada de realizar el análisis léxico sobre una cadena de entrada:

```
lexerIMP :: String -> [TokenIMP]
lexerIMP input = lexerDo dfa input
 where dfa = minimize $
                toDFA $
                toNFA $
                toNFAE regex
```

La función auxiliar `lexerDo` simula la ejecución del AFD mínimo sobre la cadena `input`, aplicando las políticas de *maximal munch* y *longest match*.

A continuación, se describe el algoritmo que implementa la máquina discriminadora determinista, el cual es simulado por `lexerDo`.

Algoritmo 3.1. Sea M el AFD mínimo obtenido a partir de la expresión regular que define el lenguaje IMP y sea $w = w_1 w_2 ... w_n$ la cadena del argumento de `lexerIMP`.

Usar w como entrada de M y comenzar desde w_1 con el estado inicial.

1. Para cada w_j $1 \leq j \leq n$ hacer:

   Aplicar $\delta(q, w_j) = q'$

   - Caso $q' \neq \emptyset$ y q' es estado final

     Marcar q' como el último estado que ha reconocido un token para la subcadena

$w_i...w_j$ con $1 \leq i < j \leq n$ e intentar buscar el token más largo, es decir, aplicar $\delta(q', w_{j+1})$

- Caso $q' \neq \emptyset$ y q' no es estado final

  Seguir consumiendo caracteres de w, es decir, aplicar $\delta(q', w_{j+1})$

- Caso $q' = \emptyset$

  Si existe un último token reconocido, reiniciar el proceso desde q_0 con la última parte de la cadena w que no fue reconocida como parte del token.

  Si no existe un último token reconocido, regresar un token de error con el carácter w_j no reconocido y reiniciar el proceso desde q_0 con $w_{j+1}...w_n$

2. El procedimiento anterior se aplica recursivamente hasta que se ha consumido toda la cadena w y se devuelve la lista de tokens reconocidos.

El algoritmo implementa *maximal munch* porque hasta que no sea posible extender más la coincidencia con un token válido, se continúa avanzando caracter por caracter de la cadena de entrada original.

A su vez, este procedimiento permite utilizar *longest match* ya que si dos o más patrones pueden coincidir con el mismo prefijo de la cadena de entrada, el algoritmo siempre selecciona el token más largo que se pueda reconocer y en caso de que existan varias coincidencias de igual longitud, se resuelve según el orden definido por el autómata.

Finalmente, para identificar la categoría léxica a la que pertenece cada componente reconocido, se definen los siguientes tokens:

```
data TokenIMP = TIdentifier String
              | TNumber Int
              | TPlus
              | TMinus
              | TEqual
              | TLeq
              | TNot
              | TAnd
              | TSkip
              | TIf
              | TThen
              | TElse
              | TWhile
              | TDo
              | TIn
              | TEnd
              | TTrue
              | TFalse
              | TSemicolon
              | TAssign
              | TCons
              | TList
              | TError String
```

El token especial `TError` permite identificar aquellos componentes léxicos que no pertenecen al lenguaje descrito por ninguna de las expresiones regulares correspondientes a las categorías léxicas.

4. Resultados & Discusión

4.1. Resumen del desarrollo

Para el análisis de los resultados obtenidos, se considera el caso en el que la entrada del programa está conformada por todas las expresiones regulares descritas en la sección 2.2 a menos que se indique lo contrario.

Se desarrolló un programa implementado en Haskell que, a partir de un archivo de texto donde se especifican las expresiones regulares correspondientes a las categorías léxicas del lenguaje IMP, permite ejecutar una función `lexer`. Dicha función, al recibir una cadena de entrada, identifica y clasifica los tokens que la componen de acuerdo con las categorías léxicas definidas.

El proceso de pasar de una expresión regular a un autómata finito determinista mínimo es bastante costoso, dado que cada módulo se pudo implementar independientemente, la unión de todos ellos causó que la cons-

trucción del AFD mínimo tarde unos minutos más a lo que se acostumbra.

A pesar de lo anterior, el programa logra identificar los tokens que pertenecen al lenguaje, sin importar si semánticamente es equivalente o no a la gramática propuesta para IMP.

4.2. Ejemplos

A continuación, se muestran algunos ejemplos de ejecución del analizador léxico. Cada entrada representa una cadena de texto que el usuario introduce, y el resultado corresponde a la lista de *tokens* reconocidos por el analizador.

- `Ejemplo 1:` Un sólo número

```
> lexerIMP "2"
  [TNumber 2]
```

- `Ejemplo 2:` Operación aritmética simple

```
> lexerIMP "2+2"
  [TNumber 2,TPlus,TNumber 2]
```

- `Ejemplo 3:` Expresión con un símbolo no reconocido

```
> lexerIMP "2+2*4"
  [TNumber 2,TPlus,TNumber 2,
  TError "*",TNumber 4]
```

  El carácter '*' no pertenece al lenguaje definido, por lo que el analizador genera el token de error TError

- `Ejemplo 4:` Expresión con múltiples operadores

```
> lexerIMP "2+2:4:[]"
  [TNumber 2,TPlus,TNumber 2,TCons,
TNumber 4,TCons,TList]
```

- `Ejemplo 5:` Expresión con espacios

```
> lexerIMP "9 - 7"
  [TNumber 9,TMinus,TNumber 7]
```

- `Ejemplo 5:` Expresión con números negativos

```
> lexerIMP "6 - -4"
  [TNumber 6,TMinus,TNumber (-4)]
```

4.3. Aplicación de políticas

Para mostrar que la máquina discriminadora determinista aplica las políticas de longest match y maximal munch, es necesario ajustar las expresiones regulares de entrada para que el lexer reconozca las secuencias =, == y =>.

No es necesario definir nuevos tokens ya que el objetivo es observar cómo el analizador léxico maneja estas secuencias bajo su comportamiento actual.

```
> lexerIMP "4=4"
[TNumber 4,TEqual,TNumber 4]

> lexerIMP "4=<4"
[TNumber 4,TEqual,TError "<",TNumber 4]

> lexerIMP "4=>4"
[TNumber 4,TIdentifier "=>",TNumber 4]

> lexerIMP "4==4"
[TNumber 4,TIdentifier "==",TNumber 4]
```

El último ejemplo muestra con mayor claridad la aplicación de la política de longest match: la cadena == no se divide en dos tokens `TEqual`, sino que se reconoce como un único identificador. Esto sucede porque no existe un token específico para ==, y, debido a la implementación desde el código fuente, el único patrón que coincide con dicha secuencia es el correspondiente a `TIdentifier`.

Es decir, si se hubiera definido un token específico para ==, como `TEqualEqual`, la máquina discriminadora determinista habría seleccionado este último en lugar de dos `TEqual`, ya que prioriza el reconocimiento del token más largo posible.

Por otro lado, la política de maximal munch se presenta en la manera en la que el analizador intenta consumir la mayor cantidad posible de caracteres para formar un token válido, como ocurre con las cadenas == y => que son más largas que =.

En el caso de <=, dado que = es un token válido mientras que < no está definido como token ni es reconocido por las expresiones regulares, la cadena se separa en dos partes: = se reconoce como `TEqual` y < se clasifica como un error léxico.

4.4. Conclusiones

El desarrollo de un analizador léxico para la detección de patrones específicos de un lenguaje, conocidos como *tokens*, nos permitío comprender tanto la importancia como la complejidad de esta etapa dentro del proceso de compilación. Esta fase representa el primer acercamiento a una abstracción de la estructura general de un programa, y es necesario garantizar que todos los componentes del lenguaje sean reconocidos de manera eficiente y precisa.

La implementación formal, que abarcó desde la construcción de las expresiones regulares hasta la máquina discriminadora correspondiente, puso a prueba no solo nuestra comprensión de los conceptos básicos de Compiladores y Lenguajes de Programación, sino también nuestras habilidades de traducción e implementación en Haskell. Así, aun cuando en muchas ocasiones teniamos la idea general de como hacer las transformaciones, la forma de plasmarlo en el código no siempre fue correcta, llevando a malos resultados o tiempos de ejecución ridiculamente altos. Sin embargo, esto nos permitió valorar la importancia de la precisión, optimización y claridad en la programación funcional.

4.5. Trabajo a fututo

Es relevante identificar optimizaciones en las funciones o estructuras de datos empleadas, con el fin de reducir el tiempo requerido para transformar una expresión regular en su correspondiente máquina discriminadora. Además, es necesario detectar y eliminar funciones auxiliares innecesarias o redundantes.

Finalmete, una posible extensión al lenguaje *IMP* podría incluir más operaciones básicas, como la multiplicación o la división, así como la introducción de variaciones a las estructuras de control existentes, con el objetivo de facilitar la lectura y escritura de los programas.

Referencias

[1] R. Awati and R. Sheldon, "What is a compiler?" https://www.techtarget.com/whatis/definition/compiler, 2025, consultado el 15 de septiembre de 2025.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston: Addison-Wesley, 2008.

[3] R. Motwani, "Cs 154 handouts," https://theory.stanford.edu/~rajeev/CS154/Handouts.html, 2009, consultado el 18 de septiembre de 2025.

[4] P. Sharma, "Regular expressions in compiler design," https://vocal.media/education/regular-expressions-in-compiler-design, 2024, consultado el 15 de septiembre de 2025.

[5] E. V. Gurovich, *Introducción a autómatas y lenguajes formales*, 2nd ed. Coyoacán, Ciudad de México: Universidad Nacional Autónoma de México, Facultad de Ciencias, Prensas de Ciencias, 2015.

[6] M. Sipser, *Introduction to the theory of computation*, 3rd ed. Cengage Learning, 2012.

[7] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages and computation*, 3rd ed. Pearson, 2012.

[8] K. D. Cooper and L. Torczon, *Engineering a compiler*, 2nd ed. Houston, Texas: Morgan Kaufmann, 2012.

[9] A. W. Li and K. Mamouras, "Efficient algorithms for the uniform tokenization problem," in *Proceedings of the ACM on Programming Languages (OOPSLA)*, vol. 9, no. OOPSLA1. ACM, 2025, pp. Article 133, 27 pages, publicado en abril de 2025. [Online]. Available: https://doi.org/10.1145/3720498

[10] W. Yang, C.-W. Tsay, and J.-T. Chan, "On the applicability of the longest-match rule in lexical analysis," *Computer Languages, Systems & Structures*, vol. 28, no. 2, pp. 273–288, 2002, recibido el 25 de abril de 2002; aceptado el 28 de junio de 2002.

[11] T. Reps, ""maximal-munch" tokenization in linear time," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 1, pp. 1–24, 1998. [Online]. Available: https://research.cs.wisc.edu/wpis/papers/toplas98b.pdf

[12] M. S. Romero, "De las expresiones regulares a los autómatas finitos deterministas," https://lambdaspace.github.io/CMP/notas/cmp_n08.pdf, 2026, unidad 2: Análisis léxico, Curso de Compiladores, Facultad de Ciencias, UNAM. Consultado el 15 de septiembre de 2025.

[13] F. Moreno, "Tema 2: Análisis léxico," https://uhu.es/francisco.moreno/gii_pl/docs/Tema_2.pdf, n.d., presentación de PowerPoint. Universidad de Huelva, Departamento de Tecnologías de la Información. Consultado el 15 de septiembre de 2025.

[14] M. Mohri, "Generic ϵ-removal algorithm for weighted automata," in *Automata Implementation: 4th International Workshop on Implementing Automata, WIA'99, Potsdam, Germany, July 1999, Revised Papers*, ser. Lecture Notes in Computer Science. Springer, 2001, vol. 2214, pp. 149–163. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-44674-5_19

[15] D. C. Kozen, *Automata and computability*. New York: Springer, 1997.

[16] E. Rich, *Automata, computability and complexity: theory and applications*, 1st ed. Pearson Education, 2007.