Lenguajes de programación

Licenciatura en Ciencias de la Computación

Universidad de Sonora

Lenguajes de dominio específico I

Eduardo Acuña Yeomans

eduardo.acuna@unison.mx

REG

Un lenguaje para lenguajes regulares

Un autómata finito determinista (DFA) consiste de

- Un conjunto finito de *estados*, denotado Q.
- Un conjunto finito de *símbolos de entrada*, denotado Σ.
- Una *función de transición*, denotada $\delta : Q \times \Sigma \to Q$.
- Un *estado inicial*, denotado $q_0 \in Q$.
- Un conjunto de *estados de aceptación*, denotado $F \subseteq Q$.

Denotamos de forma sucinta un DFA con notación de quíntupla,

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

donde A es el nombre del DFA y las componentes de la tupla de acuerdo a la notación usual.

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

Se define la función de transición extendida $\hat{\delta} : Q \times \Sigma^* \to Q$ de forma inductiva sobre la estructura de una cadena de entrada.

$$\hat{\delta}(q, \varepsilon) = q$$
$$\hat{\delta}(q, \alpha \cdot x) = \delta(\hat{\delta}(q, \alpha), x)$$

donde $q \in Q$, $x \in \Sigma$ y $\alpha \in \Sigma^*$.

El *lenguaje* $\mathcal{L}(A)$ de un DFA $A = (Q, \Sigma, \delta, q_0, F)$ se define como,

$$\mathcal{L}(A) = \left\{ \alpha \mid \hat{\delta}(q_0, \alpha) \in F \right\}$$

Un autómata finito no determinista (NFA) consiste de

- Un conjunto finito de *estados*, denotado Q.
- Un conjunto finito de *símbolos de entrada*, denotado Σ.
- Una *función de transición*, denotada $\delta : Q \times \Sigma \to \mathcal{P}(Q)$.
- Un *estado inicial*, denotado $q_0 \in Q$.
- Un conjunto de *estados de aceptación*, denotado $F \subseteq Q$.

Los NFA se denotan como quíntuplas igual que los DFA.

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

Se define la función de transición extendida $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ de forma inductiva sobre la estructura de una cadena de entrada.

$$\hat{\delta}(q, \varepsilon) = \{q\}$$

$$\hat{\delta}(q, \alpha \cdot x) = \bigcup_{i=1}^{k} \delta(p_i, x)$$

donde $\hat{\delta}(q, \alpha) = \{p_1, \ldots, p_k\}$, $q \in Q$, $x \in \Sigma$ y $\alpha \in \Sigma^*$.

El *lenguaje* $\mathcal{L}(A)$ de un NFA $A = (Q, \Sigma, \delta, q_0, F)$ se define como,

$$\mathcal{L}(A) = \left\{ \alpha \mid \hat{\delta}(q_0, \alpha) \cap F \neq \emptyset \right\}$$

Los autómatas finitos no deterministas con transiciones instantáneas son iguales a los NFA, salvo que la función de transición δ cambia su dominio.

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

donde ε es la cadena vacía y $\varepsilon \notin \Sigma$.

Teorema de Kleene

Un lenguaje es regular si y sólo si es reconocido por algún autómata finito determinista. Construcciones estándar nos permiten extender este teorema a NFA y ε-NFA.

El conjunto vacío es regular

Consideramos el DFA $A = (\{q_0\}, \Sigma, \delta, q_0, \emptyset)$, donde para toda $x \in \Sigma$,

$$\delta(q_0, x) = q_0.$$

Ya que $\hat{\delta}(q_0, \alpha) \notin \emptyset$, $\mathcal{L}(A) = \emptyset$.

El conjunto unitario con la cadena vacía es regular

Consideramos el DFA $A = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_0\})$, donde para todo $q \in Q$

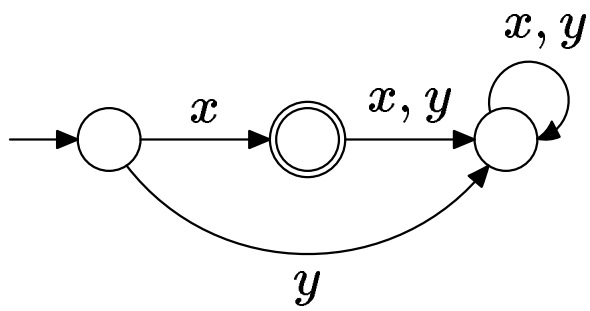
$$\delta(q, x) = q_1 .$$

Por definición, $\hat{\delta}(q_0, \varepsilon) = q_0$. Ya que q_0 es elemento del conjunto de estados de aceptación: $\varepsilon \in \mathcal{L}(A)$.

Consideremos cualquier cadena no vacía $\alpha \cdot x$, se tiene que,

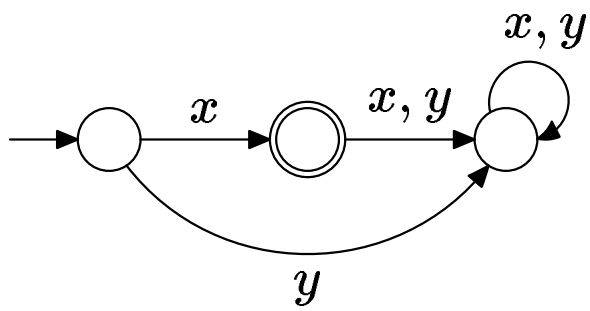
$$\begin{aligned} \hat{\delta}(q_0, \alpha \cdot x) &= \delta(\hat{\delta}(q_0, \alpha), x) \\ &= q_1 \\ &\notin \{q_0\} \end{aligned}$$

El conjunto unitario con un elemento x del alfabeto es regular



donde $y \in \Sigma \setminus \{x\}$.

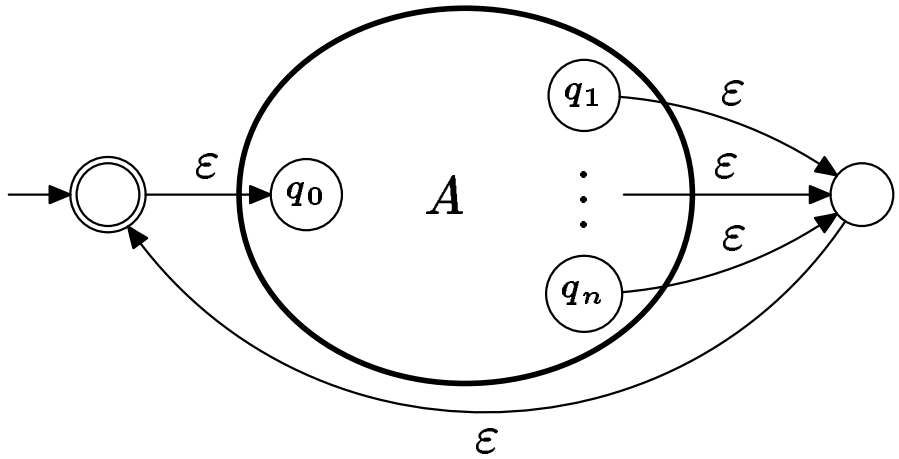
Todo $C \subseteq \Sigma$ *es regular*



para todo $x \in C$ y $y \in \Sigma \setminus C$.

Si L es regular, entonces L^* es regular

Sea $A = (Q, \Sigma, \delta, q_0, \{q_1, \ldots, q_n\})$ un DFA tal que $\mathcal{L}(A) = L$ y sea B el siguiente ε-NFA,



Si L es regular, entonces $\overline{L}$ es regular

Sea $A = (Q, \Sigma, \delta, q_0, F)$ un DFA tal que $\mathcal{L}(A) = L$.

Sea $B = (Q, \Sigma, \delta, q_0, Q \setminus F)$.

$$\begin{aligned}
\mathcal{L}(B) &= \left\{ \alpha \mid \hat{\delta}(q_0, \alpha) \in Q \setminus F \right\} \\
&= \left\{ \alpha \mid \hat{\delta}(q_0, \alpha) \notin F \right\} \\
&= \overline{\left\{ \alpha \mid \hat{\delta}(q_0, \alpha) \in F \right\}} \\
&= \overline{\mathcal{L}(A)}
\end{aligned}$$

REG: algunas propiedades de lenguajes regulares

Si L_1 *y* L_2 *son regulares, entonces* $L_1 \cup L_2$ *es regular*

Si L_1 *y* L_2 *son regulares, entonces* $L_1 \cap L_2$ *es regular*

Si L_1 *y* L_2 *son regulares, entonces* $L_1 L_2$ *es regular*

Sintaxis abstracta

La forma de las expresiones del lenguaje

Definimos algunas estructuras para representar variantes de las expresiones del lenguaje. En general, todas las expresiones van a representar algún conjunto de cadenas *regular*.

(**struct** regular () `#:transparent`)

El conjunto vacío es regular

(**struct** nothing regular () `#:transparent`)

Sintaxis abstracta

Variantes

El conjunto unitario con la cadena vacía es regular

(**struct** just-epsilon regular () `#:transparent`)

Sintaxis abstracta

Variantes

El conjunto unitario con un elemento x del alfabeto es regular

```
(struct just-char regular (value)
  #:transparent
  #:guard (lambda (c type)
            (unless (char? c)
              (raise-argument-error type "char?" c))
            c))
```

Sintaxis abstracta

Variantes

Todo $C \subseteq \Sigma$ *es regular*

(**struct** char-set regular (*predicate*)
  `#:transparent`
  `#:constructor-name` make-char-set)

Sintaxis abstracta

Variantes

Si L es regular, entonces L^* es regular

(**struct** reg-kleene regular (*set*)
  `#:transparent`
  `#:constructor-name` make-reg-kleene)

Sintaxis abstracta

Variantes

Si L es regular, entonces $\overline{L}$ es regular

(**struct** reg-comp regular (*set*)
  `#:transparent`
  `#:constructor-name` make-reg-comp)

Sintaxis abstracta

Variantes

Si L_1 y L_2 son regulares, entonces $L_1 \cup L_2$ es regular

(**struct** reg-union regular (*left right*)
  `#:transparent`
  `#:constructor-name` make-reg-union)

Sintaxis abstracta

Variantes

Si L_1 y L_2 son regulares, entonces $L_1 \cap L_2$ es regular

(**struct** reg-inter regular (*left right*)
  `#:transparent`
  `#:constructor-name` make-reg-inter)

Sintaxis abstracta

Variantes

Si L_1 y L_2 son regulares, entonces $L_1 L_2$ es regular

(**struct** reg-conc regular (*left right*)
  `#:transparent`
  `#:constructor-name` make-reg-conc)

Sintaxis abstracta

Variantes

El módulo de AST exporta la siguiente interfaz:

```
(provide regular?
         nothing nothing?
         just-epsilon just-epsilon?
         just-char just-char? just-char-value
         make-char-set char-set? char-set-predicate
         make-reg-kleene reg-kleene? reg-kleene-set
         make-reg-comp reg-comp? reg-comp-set
         make-reg-union reg-union? reg-union-left reg-union-right
         make-reg-inter reg-inter? reg-inter-left reg-inter-right
         make-reg-conc reg-conc? reg-conc-left reg-conc-right)
```

Constructores inteligentes

Subconjuntos del alfabeto

- El constructor (char-set $x_1 \ldots x_n$) toma cero o más objetos que denotan caracteres y regresa el conjunto más pequeño que los contiene.
- El constructor (char-set-comp $x_1 \ldots x_n$) toma cero o más objetos que denotan caracteres y regresa el conjunto más pequeño que contiene símbolos del alfabeto distintos a ellos.
- El constructor (char-range *from upto*) toma dos objetos que denotan caracteres y regresa un intervalo cerrado de símbolos en el orden inducido por los puntos de código de Unicode.

También se definen algunos conjuntos regulares prácticos, como el que solo contiene caracteres *alfabéticos*, *minúsculas*, *mayúsculas*, *numéricos*, *puntuación*, *espacios en blanco*, etc.

Constructores inteligentes

Subconjuntos del alfabeto

Utilizamos una función auxiliar para regresar un conjunto de caracteres a partir de cero o más objetos que denotan caracteres.

```
(define (set-of-chars who xs)
  (define (adjoin-char x soc)
    (cond [(string? x)
           (set-union soc (list→seteqv (string→list x)))]
          [(coerce-char x)
           ⇒ (lambda (c) (set-add soc c))]
          [else
           (error who "not a char-like value:  v" x)]))
  (foldl adjoin-char (seteqv) xs))
```

Constructores inteligentes

Clausura de Kleene

Sea L un lenguaje regular,

$$(L^*)^* = L^*$$
$$\emptyset^* = \{\varepsilon\}$$
$$\{\varepsilon\}^* = \{\varepsilon\}$$

Constructores inteligentes

Complemento

Sea L un lenguaje regular,

$$\overline{\overline{L}} = L$$

Constructores inteligentes

Unión

Sean L y R dos lenguajes regulares,

$$\emptyset \cup R = R$$
$$L \cup \emptyset = L$$
$$\Sigma^* \cup R = \Sigma^*$$
$$L \cup \Sigma^* = \Sigma^*$$
$$L \cup L = L$$

Constructores inteligentes

Intersección

Sean L y R dos lenguajes regulares,

$$\emptyset \cap R = \emptyset$$
$$L \cap \emptyset = \emptyset$$
$$\Sigma^* \cap R = R$$
$$L \cap \Sigma^* = L$$
$$L \cap L = L$$

Constructores inteligentes

Concatenación

Sean L y R dos lenguajes regulares,

$$\emptyset \cdot R = \emptyset$$
$$L \cdot \emptyset = \emptyset$$
$$\{\varepsilon\} \cdot R = R$$
$$L \cdot \{\varepsilon\} = L$$

Constructores inteligentes

Otros constructores

Con los tipos de expresion que hemos elegido es posible implementar constructores útiles que regresen expresiones que representan conjuntos equivalentes.

- El constructor (just-string s) toma una cadena de caracteres y regresa un conjunto unitario que la contiene.
- El constructor (reg-maybe *rset*) regresa un conjunto regular como *rset* que incluye la cadena vacía.
- El constructor (reg-repeat *lo hi rset*) toma una cota inferior *lo*, una cota superior *hi* y un conjunto regular *rset* y regresa el conjunto con todas las concatenaciones de *rset* con si mismo, al menos *lo* veces y a lo más *hi* veces.

Derivada de lenguajes regulares

Nulabilidad

El criterio de nulabilidad consiste en determinar si un lenguaje regular contiene como elemento la cadena vacía. Sean L y R cualesquiera dos lenguajes regulares.

$$
\begin{aligned}
\nu(\emptyset) &= \emptyset \\
\nu(\{\varepsilon\}) &= \{\varepsilon\} \\
\nu(\{c \in \Sigma\}) &= \emptyset \\
\nu(L^*) &= \nu(\{\varepsilon\}) \\
\nu(\overline{L}) &= \begin{cases} \{\varepsilon\} & \text{si } \nu(L) = \emptyset \\ \emptyset & \text{si } \nu(L) = \{\varepsilon\} \end{cases} \\
\nu(L \cup R) &= \nu(L) \cup \nu(R) \\
\nu(L \cap R) &= \nu(L) \cap \nu(R) \\
\nu(L \cdot R) &= \nu(L) \cap \nu(R)
\end{aligned}
$$

Derivada de lenguajes regulares

Reglas de derivación

La derivada de un lenguaje regular L con respecto a un símbolo x del alfabeto Σ, se denota $\partial_x L$, y se define como el conjunto de cadenas α, tal que $x \cdot \alpha \in L$.

Podemos pensar la derivada como un proceso donde a filtramos todas las cadenas en L que comienzan con x y posteriormente se elimina el prefijo x de cada cadena.

Derivada de lenguajes regulares

Reglas de derivación

Sean L y R dos lenguajes regulares y $x, y \in \Sigma$ y $C \subset \Sigma$ tal que $x \in C$ y $y \notin C$, se definen las siguientes reglas de derivación.

$$
\begin{aligned}
\partial_x \emptyset &= \emptyset \\
\partial_x \{\varepsilon\} &= \emptyset \\
\partial_x C &= \{\varepsilon\} \\
\partial_y C &= \emptyset \\
\partial_x L^* &= \partial_x L \cdot L^* \\
\partial_x \overline{L} &= \overline{\partial_x L} \\
\partial_x (L \cup R) &= \partial_x L \cup \partial_x R \\
\partial_x (L \cap R) &= \partial_x L \cap \partial_x R \\
\partial_x (L \cdot R) &= (\partial_x L \cdot R) \cup (\nu(L) \cdot \partial_x R)
\end{aligned}
$$

Análisis léxico

Estructura del analizador

- El analizador léxico (*lexer* es un procedimiento que toma un puerto de entrada y regresa un *token* que representa un lexema reconocido a partir de los caracteres del puerto.
- Se construye un lexer a partir de una lista de reglas, cada regla consiste de un lenguaje regular L y un procedimiento de acción A.
- El objetivo del lexer es despachar algún procedimiento de acción cuando se reconoce de la entrada una cadena que pertenece al lenguaje regular asociado.

Conforme leemos caracteres del puerto, derivamos el lenguaje de cada regla, recordando la coincidencia más larga encontrada.

Cuando ya no hay caracteres en el puerto, o llegamos a que todos los lenguajes rechazan la cadena leída, despachamos la acción asociada a la coincidencia más larga.

Si dos o más reglas aceptan la coincidencia más larga, elegimos la primera conforme aparecen en la lista de reglas.