Tecnológico de Monterrey

Evidencia 1

Un lenguaje para definir autómatas

Implementación de métodos computacionales

Profesora: María Valentina Narváez Terán

Bruno Fernando Zabala Peña - A00838627

25/05/2025

Introducción

El siguiente reporte muestra la metodología seguida para la creación de un lenguaje para definir autómatas, el cual fue desarrollado por medio de Racket. Para esto, se definió la lista de lexemas, la sintaxis del lenguaje y un esquema de colores para su visualización, así como un script que permite verificar que un código es válido según las reglas establecidas o los errores detectados.

Palabras clave: automatas, racket, lenguaje de programación, expresiones regulares

Metodología

El primer paso para la creación del lenguaje fue la creación de una lista de lexemas válidos para el mismo, los cuáles cumplen distintas funciones.

Lexemas

Lexema	Descripción	Expresión regular
automaton	Constructor para un autómata	#rx"^automaton"
{	Llave abierta, se usa para comenzar la definición de los estados de un autómata	#rx"^\\{"
}	Llave de cierre, se usa para terminar la definición de estados de un autómata	#rx"^\\}"
Identificador	Utilizados para guardar	#rx"^[a-zA-Z][a-zA-Z0-9_]*

	listas de caracteres o números o la definición de un autómata.	"
start_state	Define el estado inicial del autómata	#rx"^start_state"
states	Define los estados del autómata	#rx"^states"
end_states	Define los estados finales del autómata	#rx"^end_states"
Estados	Inician con Q y van seguidos de un número (ej. Q1)	#rx"^Q[0-9]+"
alphabet	Lista de caracteres utilizados por el autómata	#rx"^alphabet"
->	Asignación	#rx"^->"
[	Corchete abierto, se usa para iniciar listas	#rx"^\\["
]	Corchete cerrado, se usa para cerrar listas	#rx"^\\]"
,	Se usa para agregar elementos a una lista o parámetros múltiples a una función	#rx"^,"
addTran	Función para agregar transiciones a un autómata, recibe un	#rx"^.addTran"

	estado inicial, un estado objetivo y los caracteres que provocan la transición	
(	Paréntesis abierto, se usa para comenzar a pasar parámetros a una función	#rx"^\\("
)	Paréntesis cerrado, se usa para terminar de pasar parámetros a una función	#rx"^\\)"
// Comentarios	Comentario de una línea, admite cualquier tipo de carácter, ignorado por el programa	#rx"^//[^\r\n]*"
char	Caracteres, usados para las transiciones	#rx"^'(\\\\.|[^'])'"
string	Cadenas de texto, usadas para probar el automata definido	#rx"^\"([^\"\\]|\\\\.)*\""
Número de un dígito	Números, usados para las transiciones	#rx"^[0-9]+"
Espacio en blanco (space, salto de línea, tab)	Espacios entre lexemas, usado en el proceso de tokenización y coloreo	#rx"^[\t\r\n]+"

Esquema de colores

Lexema	Color
automaton	#f5c2e7
{ }	#f9e2af
Identificadores	#eebebe
start_state	#89b4fa
states	#89b4fa
end_states	#89b4fa
Estados	#94e2d5
alphabet	#cba6f7
->	#f38ba8
[ ]	#fab387
Caracteres ('a','b')	#eba0ac
addTran	#b4befe
,	#9399b2
test	#b4befe

( )	#f9e2af
// Comentario	#6c7086
Número de un dígito	#a6adc8
Caracteres no válidos (errores)	#f38ba8

Para el esquema de colores, se utilizó como referencia la paleta de colores de Catppuccin (https://catppuccin.com/palette/).

Antes de realizar cualquier función, el programa lee el contenido de un archivo de texto, el cual contiene un programa que define a un autómata de prueba, y lo almacena.

```
; Lee un archivo de texto con un automata y guarda su contenido en una cadena
(define (leer-archivo ruta)
    (define contenido (file->string ruta))
    (string-trim contenido)
)


(define cadena_automata (leer-archivo "evidencia/idk.txt"))
```

Tokenización

Para clasificar los lexemas del código de entrada, se utilizó un proceso de tokenización recursiva, en el que:

- Se toma la cadena que define al autómata
- Se aplica la lista de expresiones regulares al inicio de la cadena
- Si existe un match, se guarda el resultado y se pasa a la siguiente expresión regular.

- Si el match de la expresión regular actual es más largo (se identifican más caracteres) que el match guardado, se actualiza y se guarda el actual. En caso de que la coincidencia sea más corta, no se actualiza.
- El proceso anterior se repite hasta que se agota la lista de expresiones regulares, el match se guarda en una lista de tokens y se pasa al siguiente lexema de la cadena para realizar el mismo procedimiento.

```
; Función recursiva para tokenizar
(define (tokenizar texto)
    (define (encontrar-mejor-match patrones texto mejor-actual)
        (cond
            [(null? patrones) mejor-actual]
            [else
                (define actual (car patrones))
                (define etiqueta (car actual))
                (define regex (cadr actual))
                (define match (regexp-match regex texto))

                (if (and match
                        (or (not mejor-actual)
                            (> (string-length (car match)) (string-length (cdr mejor-actual)))
                        )
                    )

                    (encontrar-mejor-match (cdr patrones) texto (cons etiqueta (car match)))
                    (encontrar-mejor-match (cdr patrones) texto mejor-actual)
                )
            ]
        )
    )
```

```
(define (tokenizar-recursivo texto-restante tokens)
    (cond
        [(string=? texto-restante "") (reverse tokens)]

        [else
            (define match (encontrar-mejor-match reglist texto-restante #f))

            (if match
                (if (equal? (car match) "espacio")
                    (tokenizar-recursivo
                        (substring texto-restante (string-length (cdr match)))
                        (cons match tokens)
                    )

                    (tokenizar-recursivo
                        (substring texto-restante (string-length (cdr match)))
                        (cons match tokens)
                    )
                )

                ; Identificar lexemas no validas caracter por carácter
                (begin
                    (substring texto-restante 0 1)
                    (tokenizar-recursivo
                        (substring texto-restante 1)
                        (cons (list "error" (substring texto-restante 0 1)) tokens)
                    )
                )
            )
        ]
    )
)

(tokenizar-recursivo texto '())
)
```

Esto devuelve una lista de listas en el formato:

((tipo_lexema . lexema) (tipo_lexema . lexema) …)

```
((automaton . automaton) (espacio .  ) (identificador . auto_2) (llave-abierta . {)
```

Para todo el contenido leído del archivo. En el caso de que se detecten lexemas no válidos, estos se almacenan como un token de tipo “error” y se colocan carácter por carácter entre paréntesis para simular los *squiggles* que se muestran al programar en un editor o IDE.

Coloreado y generación del HTML

Para asociar cada tipo de lexema con un color, se creó una función que devuelve un valor hexadecimal de acuerdo al tipo de token que se recibe.

```
; Funcion para obtener el codigo de acuerdo al tipo de token detectado
(define (obtener-color tipo-token)
    (case tipo-token
        [("automaton") "#f5c2e7"]
        [("llave-abierta" "llave-cierre") "#f9e2af"]
        [("identificador") "#eebebe"]
        [("start_state" "states" "end_states") "#89b4fa"]
        [("estado") "#94e2d5"]
        [("alphabet") "#cba6f7"]
        [("asignacion") "#f38ba8"]
        [("corchete-abierto" "corchete-cierre") "#fab387"]
        [("char") "#eba0ac"]
        [("string") "#f5c2e7"]
        [("addTran") "#b4befe"]
        [("test") "#b4befe"]
        [("coma") "#9399b2"]
        [("par-abierto" "par-cierre") "#f9e2af"]
        [("comentario") "#6c7086"]
        [("digito") "#a6adc8"]
        [("error") "#f38ba8"]
        [else "#cdd6f4"]
    )
)
```

Para generar el HTML se sigue el siguiente proceso:

- Se llama a la función de tokenización para obtener la lista de tokens del programa.
- Para cada token, se extrae el tipo de token con car y el contenido del mismo con cdr. Si el tipo de token es un espacio, este se pasa sin formatear, en caso contrario, el contenido del token se introduce a un span de HTML y se aplica un estilo *in-line* con el color que se obtiene al llamar a la función correspondiente, obteniendo una línea en el formato:

  Contenido token

- Se hace un append de todas las líneas generadas con apply.

Para guardar las líneas en un archivo HTML, se especifica la ruta deseada. Al inicio del archivo, se introduce la línea:

<!DOCTYPE html>\n<html style='background-color: #1e1e2e; color: #cdd6f4'>\nAutomata\n<head>\n<title>Automata</title>\n</head>\n<body>\n

La cual establece un título y un color de fondo personalizado. Después, se imprimen las líneas generadas dentro de un <pre> </pre> para preservar el formato y se cierra el archivo.

```
(define (guardar-html contenido ruta)
    (call-with-output-file ruta
        (lambda (out)
            (fprintf out "<!DOCTYPE html>\n<html style='background-color:□ #1e1e2e; color:■ #cdd6f4'>\nAutomata\n<head>\n<title>Automata</title>\n</head>\n<body>\n")
            (display "<pre>" out)
            (display contenido out)
            (display "</pre>\n" out)
            (fprintf out "</body>\n</html>")
        )

        #:exists 'replace
    )
)
```

Para obtener un resultado similar al siguiente:

evidencia > automata_con_errores.html > html

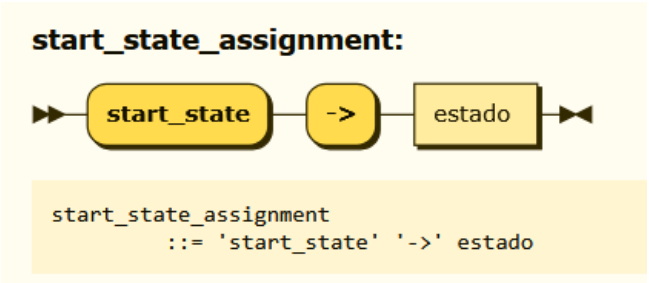
```
<!DOCTYPE html>
<html style='background-color: □#1e1e2e; color: ■#cdd6f4'>
Automata
<head>
<title>Automata</title>
</head>
<body>
<pre><span style='color: ■#f5c2e7'>automaton</span> <span style='color: ■#eebebe'>auto_2</span><span style='color: ■#f9e2af'>{</
    <span style='color: ■#89b4fa'>states</span> <span style='color: ■#f38ba8'>-></span> <span style='color: ■#fab387'>[</span><s
    <span style='color: ■#89b4fa'>start_state</span> <span style='color: ■#f38ba8'>-></span> <span style='color: ■#94e2d5'>Q0</s
    <span style='color: ■#89b4fa'>end_states</span> <span style='color: ■#f38ba8'>-></span> <span style='color: ■#fab387'>[</spa
    <span style='color: ■#cba6f7'>alphabet</span> <span style='color: ■#f38ba8'>-></span> <span style='color: ■#fab387'>[</span>
<span style='color: ■#f9e2af'>}</span>

<span style='color: ■#f38ba8'>(-)</span><span style='color: ■#f38ba8'>(-)</span><span style='color: ■#f38ba8'>(-)</span><span st

<span style='color: ■#6c7086'>// Esto es un comentario, boing</span>

<span style='color: ■#eebebe'>nums</span> <span style='color: ■#f38ba8'>-></span> <span style='color: ■#fab387'>[</span><span st
<span style='color: ■#eebebe'>auto_2</span><span style='color: ■#b4befe'>.addTran</span><span style='color: ■#f9e2af'>(</span><s
<span style='color: ■#eebebe'>auto_2</span><span style='color: ■#b4befe'>.addTran</span><span style='color: ■#f9e2af'>(</span><s
<span style='color: ■#eebebe'>auto_2</span><span style='color: ■#b4befe'>.addTran</span><span style='color: ■#f9e2af'>(</span><s
<span style='color: ■#eebebe'>auto_2</span><span style='color: ■#b4befe'>.addTran</span><span style='color: ■#f9e2af'>(</span><s
<span style='color: ■#eebebe'>auto_2</span><span style='color: ■#b4befe'>.test</span><span style='color: ■#f9e2af'>(</span><span
</body>
</html>
```

Automata

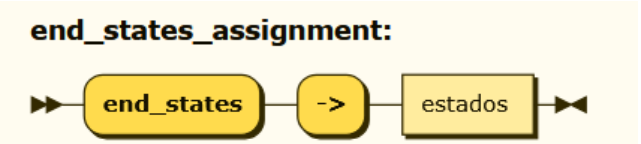
```
automaton auto_2{
        states -> [Q0, Q1, Q2]
        start_state -> Q0
        end_states -> [Q2]
        alphabet -> ['+', '-', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}

(-)(-)(-)(-)(-)(-)

// Esto es un comentario, boing

nums -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
auto_2.addTran(Q0, Q1, ['+', '-'])
auto_2.addTran(Q1, Q2, nums)
auto_2.addTran(Q2, Q2, nums)
auto_2.addTran(Q0, Q2, nums)
auto_2.test("hola")
```

Validación de sintaxis

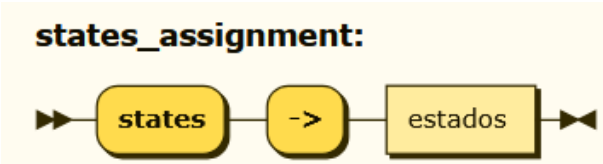
El primer paso para este proceso fue generar diagramas de sintaxis que permitan definir cómo se estructura el lenguaje en casos de: declaración del autómata, asignación de valores a identificadores (tanto reservados para el autómata como creados por el usuario), comentarios y utilización de métodos como addTran.




```
start_state_assignment
        ::= 'start_state' '->' estado
```



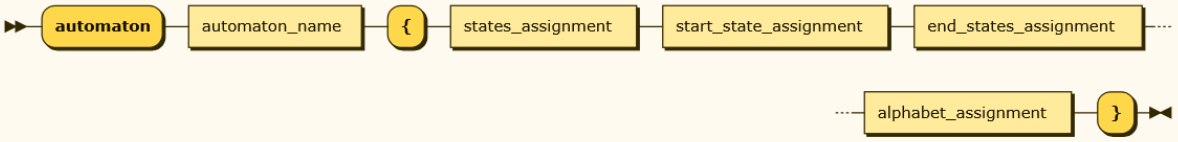

```
end_states_assignment
        ::= 'end_states' '->' estados
```




```
states_assignment
        ::= 'states' '->' estados
```


```
alphabet_assignment
        ::= 'alphabet' '->' simbolos
```

automaton_declaration:




```
automaton_declaration
        ::= 'automaton' automaton_name '{' states_assignment start_state_assignment end_states_assignment alphabet_assignment '}'
```

transitions:


```
transitions
        ::= automaton_name '.addTran' '(' estado_inicial ',' estado_objetivo ',' caracteres ')'
```

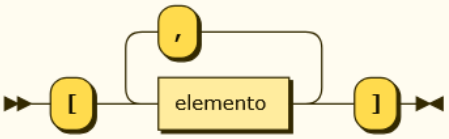
variable_assignment:


```
variable_assignment
        ::= variable_name '->' caracteres
```

comment:


```
comment  ::= '//' comment_content
```

list:




```
list     ::= '[' elemento ( ',' elemento )* ']'
```

Para el proceso de verificación, se realizó un parsing con descenso recursivo de la siguiente manera:

- Se obtiene la lista de tokens con la función de tokenización
- Se define una función auxiliar que llama a una función que maneja los tokens individuales hasta procesar toda la lista de tokens.

- Se revisa a modo de secuencia que se esté pasando a los estados adecuado de acuerdo a lo definido en los diagramas de estados (por ejemplo, al llamar a la función desde el estado inicio, se espera que el siguiente token sea de tipo automaton para pasar al siguiente).
- En caso de que se detecte un token que no debería estar de acuerdo a la secuencia definida, se reporta el error y se sigue procesando la lista de tokens.

```
; Estado inicial
[(equal? estado "inicio")
    (if (equal? tipo "automaton")
        (verificar-tokens tokens-restantes "nombre-automata" errores)

        (verificar-tokens
            tokens-restantes
            "error"
            (cons (format "Error: Se esperaba 'automaton', pero se encontró '~a'" contenido) errores)
        )
    )
]
```

Si se recibe el token correcto, pasa a:

```
[(equal? estado "nombre-automata")
    (if (equal? tipo "identificador")
        (verificar-tokens tokens-restantes "llave-abierta" errores)

        (verificar-tokens
            tokens-restantes
            "error"
            (cons (format "Error: Se esperaba un identificador, pero se encontró '~a'" contenido) errores)
        )
    )
]
```

- Y repite el proceso hasta haber terminado la lista de tokens.

Para el archivo HTML, en caso de que no se encuentren errores, se despliega un mensaje en verde diciendo que la sintaxis es correcta, en caso contrario, se muestra una lista de errores en color rojo:

Automata

```
automaton auto_2{
        states -> [Q0, Q1, Q2]
        start_state -> Q0
        end_states -> [Q2]
        alphabet -> ['+', '-', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}


// Esto es un comentario, boing

nums -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
auto_2.addTran(Q0, Q1, ['+', '-'])
auto_2.addTran(Q1, Q2, nums)
auto_2.addTran(Q2, Q2, nums)
auto_2.addTran(Q0, Q2, nums)
auto_2.test("hola")
```

Análisis completado:

El código es sintácticamente correcto.

Automata

```
automaton auto_2{
        states -> [Q0, Q1, Q2]
        start_state -> Q0
        end_states -> [Q2]
        alphabet -> ['+', '-', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}

(-)

// Esto es un comentario, boing

nums ->
auto_2.addTran(Q0, Q1, ['+', '-'])
auto_2.addTran(Q1, Q2, nums)
auto_2.addTran(Q2, Q2, nums)
auto_2.addTran(Q0, Q2, nums)
auto_2.test((")ho
```

Errores de sintaxis:

- Error: Token inesperado '(-)' fuera del autómata
- Error: Se esperaba una lista de símbolos, pero se encontró 'auto_2'
- Error: Se esperaba una cadena después de .test(, pero se encontró '(")'

Pruebas

Códigos válidos

Automata

```
automaton auto_1{
        states -> [Q0, Q1, Q2, Q3]
        start_state -> Q0
        end_states -> [Q3]
        alphabet -> ['a', 'x', 'y', '-', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}
// Lista de numeros
nums -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
auto_1.addTran(Q0, Q1, ['a', 'x', 'y'])
auto_1.addTran(Q1, Q1, ['a', 'x', 'y'])
auto_1.addTran(Q1, Q2, ['-'])
auto_1.addTran(Q2, Q3, nums)
auto_1.addTran(Q3, Q3, nums)

Análisis completado:

El código es sintácticamente correcto.
```

Automata

```
automaton auto_2{
        states -> [Q0, Q1, Q2]
        start_state -> Q0
        end_states -> [Q2]
        alphabet -> ['+', '-', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}


// Esto es un comentario, boing

nums -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
auto_2.addTran(Q0, Q1, ['+', '-'])
auto_2.addTran(Q1, Q2, nums)
auto_2.addTran(Q2, Q2, nums)
auto_2.addTran(Q0, Q2, nums)
auto_2.test("hola")

Análisis completado:

El código es sintácticamente correcto.
```

Códigos con errores

Automata

```
automaton auto_2{
        states -> [Q0, Q1, Q2]
        start_state -> Q0
        end_states -> [Q2]
        alphabet -> ['+', '-', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}

(-)(-)(-)(?)

// Esto es un comentario, boing

nums -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
auto_2.addTran(Q0, Q1, ['+', '-'])
auto_2.addTran(Q1, Q2, nums)
auto_2.addTran(Q2, Q2, nums)
auto_2.addTran(Q0, Q2, nums)
auto_2.test("hola")
```

Errores de sintaxis:

- Error: Token inesperado '(-)' fuera del autómata
- Error: Token inesperado '(-)' fuera del autómata
- Error: Token inesperado '(-)' fuera del autómata
- Error: Token inesperado '(?)' fuera del autómata

Automata

```
automaton auto_2{
        states -> [Q0, Q1, Q2]
        start_state ->
        end_states -> [Q2]
        alphabet -> ['+', '-', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}

// Esto es un comentario, boing

nums -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
auto_2.addTran(Q0, Q1, ['+', '-'])
auto_2.addTran(Q1, Q2, nums)
auto_2.addTran(Q2,
auto_2.addTran(Q0, Q2, nums)
auto_2.test("hola")
```

Errores de sintaxis:

- Error: Se esperaba un estado después de 'start_state ->', pero se encontró 'end_states'
- Error: Se esperaba un estado después de la coma, pero se encontró 'auto_2'

Automata

```
automaton auto_2{
        states -> [Q0, Q1, Q2]
        start_state -> Q0
        end_states -> [Q2]
        alphabet -> ['+', '-', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}


(/) Esto es un comentario, boing

nums -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
auto_2.addTran(Q0, Q1, ['+', '-'])
auto_2.addTran(Q1, Q2, nums)
auto_2.addTran(Q2, Q2, nums)
auto_2.addTran(Q0, Q2, nums)
auto_2.test("hola")
```

Errores de sintaxis:

- Error: Token inesperado '(/)' fuera del autómata
- Error: Se esperaba -> o .addTran, pero se encontró 'es'
- Error: Se esperaba -> o .addTran, pero se encontró 'comentario'
- Error: Se esperaba -> o .addTran, pero se encontró 'nums'

Automata

```
automaton auto_2{
        states -> [Q0, Q1, Q2]
        start_state -> Q0
        end_states -> [Q2]
        alphabet -> ['+', '-', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]



// Esto es un comentario, boing

nums -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
auto_2.addTran(Q0, Q1, ['+', '-'])
auto_2.addTran(Q1, Q2, nums)
auto_2.addTran(Q2, Q2, nums)
auto_2.addTran(Q0, Q2, nums)
auto_2.test("hola")
```

Errores de sintaxis:

- Error: Token inesperado '// Esto es un comentario, boing' en instrucciones del autómata

Desglose de contribuciones individuales

Trabajo individual