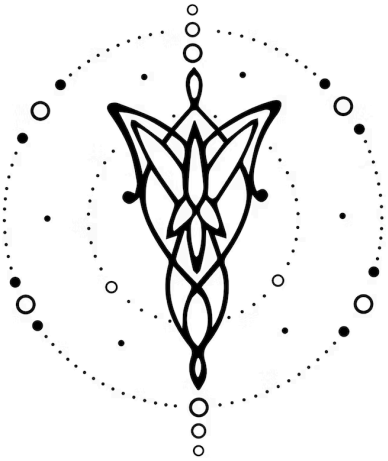

Especificación

Diseño e Implementación de un Lenguaje

v1.0.0

1. Equipo

Nombre	Apellido	Legajo	E-mail
Sol	Rodriguez	63029	solrodriguez@itba.edu.ar
Maria Agustina	Sanguinetti	63115	msanguinetti@itba.edu.ar
Uriel Ángel	Arias	63504	uarias@itba.edu.ar

2. Repositorio

La solución y su documentación serán versionadas en: AutomaTeX.

3. Dominio

Desarrollar un lenguaje de dominio específico (DSL) con tipado estático que permita la creación y simulación de autómatas finitos, de manera tal que una vez compilado un archivo escrito en este lenguaje se devuelva una representación equivalente de los autómatas definidos por medio de un documento en formato LaTeX, el cual contenga su respectivo diagrama de estados y tabla de transiciones.

Los usuarios podrán definir el autómata, especificando los distintos estados y sus tipos, el estado inicial, un estado trampa y aquellos que sean finales, determinar las transiciones entre ellos, y establecer el alfabeto que el mismo reconocerá. Además, se podrá simular la minimización de un autómata, permitiendo a los usuarios observar cómo el autómata definido inicialmente es modificado para obtener dicha representación mínima.

Cabe destacar que si bien se reconoce que se podría manejar la definición de autómatas sin especificar su tipo dadas las equivalencias probadas entre los mismos y la existencia de algoritmos para hacer la conversión de uno a otro, se hace uso de tipos de autómatas para reducir la complejidad del proyecto y aprovechar el uso de errores de compilación para marcar errores conceptuales propios de usuarios principiantes con los conceptos teóricos. Se trata entonces de un aspecto sobre el cual el proyecto podría seguir escalando para hacer que su uso sea menos restrictivo y permita la reutilización de los autómatas ya definidos.

Por otro lado, junto con la definición de autómatas se hace necesario el manejo de conjuntos que bien podrían reutilizarse para autómatas con propiedades similares, por ejemplo, al compartir los mismos estados. Es por esto que se provee de tipos de datos que si bien refieren a las propiedades de los autómatas, como estados, transiciones y un alfabeto, programáticamente se distinguen por usar notación y operaciones binarias básicas de conjuntos, entre las cuales se encuentran la unión, la intersección y la diferencia. Además se brindarán estructuras de control y estructuras iterativas para poder construirlos y reutilizarlos.

La implementación exitosa de este lenguaje proporcionará una herramienta poderosa para desarrolladores de sistemas basados en autómatas finitos, permitiéndoles modelar y verificar la correctitud de los autómatas definidos de una forma clara y fácil de utilizar para que el usuario no pierda tiempo en recordar la sintaxis de LaTeX.

4. Construcciones

El lenguaje desarrollado debería ofrecer las siguientes construcciones, prestaciones y funcionalidades:

(I). Se podrán crear autómatas de tipo DFA, NFA y LNFA (Lambda NFA), y estos podrán ser identificados a través de un nombre o etiqueta.

(II). Se podrán definir estados, transiciones y un alfabeto dentro de cada autómata.

(III). Se podrán crear estados, transiciones y alfabetos independientemente de los autómatas, es decir, como constantes o variables globales, dependiendo de si sufrirán modificaciones, por medio de las palabras reservadas *const* y *var* respectivamente.

(IV). Las constantes o variables globales podrán ser de tipo *states* para los estados, *transitions* para las transiciones, *alphabet* para el alfabeto y para todas ellas la letra inicial del nombre deberá estar en mayúscula.

(V). Para alterar el valor de una variable se deberá indicar su nombre y seguido de dos puntos (":") se deberá indicar el valor que se desea que tenga. Para ello se pueden emplear operadores, los cuales se explicarán más adelante, y, por ende, otras variables o constantes del mismo tipo. También se podrá llamar a sí misma. Por ejemplo, sea A una variable de states a la cual se desea agregar un estado *p*, entonces se la puede redefinir de la siguiente manera: A: A + {*p*}.

(VI). Se podrán definir transiciones utilizando una sintaxis de la forma *p*-*a*->*q* o *p*<-*a*->*q*, siendo *p* y *q* dos estados o conjuntos de estados y *a* un símbolo o conjunto de símbolos del alfabeto definido para un autómata, donde en el primer caso se define una transición del estado *p* a *q* consumiendo *a*, mientras que en el segundo se definen no solo lo equivalente a la primera sino también aquellas que parten de q y pasan a *p* consumiendo *a*. Si se quiere hacer referencia a más de 2 estados o símbolos, se podrán listar dichos estados o símbolos del alfabeto entre paréntesis, tal como se ve en el siguiente ejemplo: {*q*0, *q*1}-{*a*,*b*}->*q*2, que indica que los estados *q*0 y *q*1 al consumir *a* o *b* llegan al estado *q*2, es decir, que en una única sentencia se representan 4 transiciones distintas. También se podrían usar los () para los estados a los que se transiciona: {q1,q2}-a->{q1,q2}, que indica que q1 y q2 al consumir *a* llegan a q1 o q2, es decir, 4 transiciones distintas.

(VII). Los conjuntos serán definidos mediante llaves ("{", "}") y sus elementos se deberán separar con comas. No obstante, también se podrá definir un conjunto de un único elemento sin llaves, de manera tal que se asumirá que el conjunto solo tiene un elemento.

(VIII). Se empleará el símbolo @ para representar lambda.

(IX). Se empleará *VOID* como palabra reservada para representar el conjunto vacío. Otra manera de representarlo es mediante el uso de {}, pues al no contener elementos se entenderá que se trata del conjunto vacío. No se podrá incluir a *VOID* en un conjunto.

(X). Los autómatas podrán ser de tipo *DFA*, *NFA* y *LNFA*. Los mismos se definen de la forma *const tipoAutomata NombrAutomata: []* o *var tipoAutomata NombreAutomata: []*

(XI). Para especificar los estados, alfabeto y transiciones de un autómata se utilizan *states*, *alphabet* y *transitions* respectivamente. Es de carácter obligatorio definir los mismos dentro de un autómata para hacer posible su creación.

(XII). Dentro del conjunto *states* definido en la creación del autómata se deberán definir los estados "regulares" (no iniciales ni finales), finales y el inicial obligatoriamente, mientras que el trampa será opcional.

(XIII). Para ello, se deberán colocar los estados que participarán en dicho conjunto y usando una etiqueta se definirán estos estados. En el caso del estado inicial, la etiqueta a usar es la <i>, para los estados finales se empleará <*f*>, y para el trampa <t>. Por ejemplo para definir *p* y *q* como estados finales, se pondrá: <*f*>: {*p*,*q*}. Estos subconjuntos se podrán acceder de manera tal de poder reutilizarlos para crear otros autómatas o hacer otras operaciones, para lo cual se deberá especificar el subconjunto requerido de la siguiente

forma NombreAutomata.states.tipo donde tipo podrá ser reemplazado por las palabras reservadas *initial*, *final*, *trap* o *regular*.

(XIV). En caso de que el usuario no defina el estado trampa, por default se lo nombrará como *TRAP*, por ende dicha palabra será una palabra reservada.

(XV). Una vez creado el autómata, se podrán acceder y modificar - si fue definido con *var* - sus estados, transiciones y alfabeto usando el nombre del autómata seguido de un punto y el nombre de dicho conjunto. Por ejemplo, para acceder a los estados de un autómata se deberá escribir lo siguiente: *NombreAutomata.states*, donde se tendrá un conjunto formado por 4 subconjuntos uno de estados regulares, uno con el estado inicial, uno con los estados finales y uno con el estado trampa.

(XVI). Se podrán emplear operadores para realizar operaciones entre conjuntos del mismo tipo. Los operadores provistos son: el "+" para la unión, el "^" para la intersección y el "-" para la diferencia. De esta manera, se podrá utilizar cualquiera de estos operadores tanto para variables o constantes, como para conjuntos no vinculados a un nombre o etiqueta. Será válida una expresión de la forma A * B, siendo A y B dos variables o constantes que son del mismo tipo y "*" un operador dentro del conjunto {+, -, ^}, y de la forma A * {*a*, *b*, *c*} donde A es de tipo *alphabet* o *states*, dependiendo de si {a, b, c} son elementos del alfabeto o estados. Cabe destacar que estos operadores tienen la misma precedencia. Si se desea indicar cierto orden a la hora de resolver un conjunto de operaciones, se deben usar paréntesis, sino por defecto la lectura se llevará a cabo de izquierda a derecha.

(XVII). Se podrá aplicar la operación de minimizar un autómata desde cualquiera que esté definido como *DFA* mediante el llamado de la función *minimize*() y pasando como parámetro el autómata. Si bien se sabe que existe una equivalencia entre todos los tipos de manera tal que teóricamente podría aplicarse sobre cualquier autómata definido, decidimos limitar esta operación a los autómatas que inicialmente sean definidos como DFA. Cabe destacar que dado que el autómata será modificado para lograr su minimización, entonces este deberá ser *var*.

(XVIII). Así como se brindan funciones ya definidas en el lenguaje como minimize, también se permitirá que el usuario defina sus propias funciones por medio de la palabra reservada *def* seguida del nombre de la función y paréntesis que contendrán sus parámetros. Los mismos deberán listarse separados por comas y definirse de la forma tipoOperaciones *tipoElementos NombreVar*. En particular, *tipoOperaciones* refiere a 3 posibles tipos: *in* para parametros que solo se utilizan por su valor derecho, *out* para parametros que se utilizan solo para almacenar un valor de retorno e *in-out* que sirve como una combinación de estos dos anteriores, mientras que tipoElementos refiere a si se trata de elementos de tipo *states*, *alphabet* o *transitions*. Por último se deberá encerrar el bloque de código que la define por medio de corchetes.

(XIX). Los autómatas que se mostrarán en el documento LaTeX serán solo aquellos que sean definidos con nombre y en el caso de aquellos definidos como variables se tomará la última versión de los mismos tras ser modificados.

(XX). Se definen estructuras de control de tipo condicionales por medio de estructuras where-do-else donde análogamente a un if-else en algún lenguaje como C, se evalúa la expresión que le sigue a la palabra reservada where o else - si viene a continuación de un bloque where- y se define el bloque de código a ejecutar luego de la palabra reservada *do* entre corchetes ("*[*", "*]*").

(XXI). Para evaluar expresiones se podrán emplear is, in, and, or, not, exists, y se podrá usar la palabra reservada *element* para referirse justamente al elemento de un conjunto. Cabe destacar que *is* es semejante al == en lenguajes como C, y *not* niega la expresión. Asimismo, al igual que con los otros operadores, también se podrán emplear los paréntesis para indicar el orden en que se deberán analizar las expresiones.

(XXII). A su vez, se definen estructuras de tipo iterativas por medio de *for every* donde de manera análoga a un for-each en algún lenguaje como Java se encarga de iterar a través del conjunto de estados, alfabetos o transiciones, asignando un único elemento a una variable que se define con un nombre elegido por el usuario el lenguaje. Para esto, seguido de este nombre se indica con la palabra reservada *in* a qué conjunto pertenecen los elementos sobre los que se recorrerá, y a continuación se define el código a ejecutar con la palabra reservada *do* seguida del bloque de código a ejecutar encerrado entre corchetes.

5. Casos de Prueba

Se proponen los siguientes casos iniciales de prueba de **`aceptación`**:

(I). Un autómata donde se tenga un estado que es tanto final como inicial y al menos un estado no final ni inicial.

(II). Un autómata del tipo DFA que contenga un estado trampa y no se definan las transiciones hacia él, de manera tal que en la salida se incluyan dichas transiciones y flechas faltantes que llegan a este estado.

(III). Un autómata que tenga 2 estados finales.

(IV). Un autómata de tipo LNFA donde se tengan 3 transiciones lambda.

(V). Un autómata de tipo NFA en donde se tenga 2 transiciones que partiendo del mismo estado y consumiendo el mismo símbolo llegue a 3 estados diferentes.

(VI). Un autómata DFA donde se tenga al menos una transición que sale de un estado y llega al mismo estado, es decir, un estado donde su diagrama tenga un lazo.

(VII). Un autómata de tipo DFA donde un par de estados p y q estén presentes en transiciones de modo tal que al consumir el mismo símbolo del alfabeto, llámese a, desde p se llegue a q y viceversa. Es decir, en los cuales se defina una transición del tipo p<-a->q.

(VIII). Un conjunto de estados como constante donde se tengan 4 estados: uno inicial, 2 finales y uno regular. Luego que se referencie al definir dos autómatas distintos, uno DFA y otro NFA.

(IX). Un autómata LNFA o NFA cuyo conjunto de transiciones contenga únicamente una resta entre un conjunto de transiciones y otro conjunto idéntico definidos previamente.

(X). Un conjunto de transiciones de cualquier tipo como constante donde se utilice en la definición de dos autómatas de distinto tipo.

(XI). Un conjunto de transiciones como una variable donde se tengan transiciones que consuman lambda y luego de ser utilizado en la definición de los estados de un autómata de tipo LNFA se modifique para eliminar las transiciones lambda de manera tal de utilizarse en la definición de un autómata de tipo DFA.

(XII). Un autómata donde el conjunto de estados se forme a partir de la intersección del conjunto de estados de dos autómatas distintos previamente definidos.

(XIII). Un autómata donde su alfabeto sea la unión de los alfabetos de dos autómatas distintos previamente definidos.
(XIV). Un autómata donde su conjunto de estados sea resultado de aplicar la diferencia simétrica entre otros dos conjuntos de estados usados para dos autómatas definidos previamente.
(XV). Una función donde sus parámetros sean uno de tipo in y otro de tipo out que devuelve el conjunto de estados finales de un autómata con alguna modificación.
(XVI). Una función donde se tenga un parámetro de tipo in-out con un autómata y sobre el mismo bloque de evalúe al menos una expresión utilizando exists.
(XVII). Un *for every* donde se evalúe al menos una expresión con un *where-do* y un *else-do*.

Además, los siguientes casos de prueba de **rechazo**:

(I). Un autómata definido como un DFA con un estado que consuma un mismo símbolo del alfabeto para transicionar a más de un estado.
(II). Un autómata de cualquier tipo con un estado trampa y que cuente con alguna transición que partiendo del trampa llega hacia cualquier otro estado que no sea él mismo.
(III). Un autómata definido como un DFA o NFA que tenga una transición en donde no se consuma ningún símbolo del alfabeto.
(IV). Un autómata que no contenga un estado inicial o contenga más de uno.
(V). Un autómata que no tenga estados finales.
(VI). Un autómata que no tenga estados o alfabeto.
(VII). Un conjunto al que no se le coloca el "}" para terminar su definición.
(VIII). Una operación donde los valores con los que se opera no son del mismo tipo.
(IX). Una operación donde existan más "(" que ")" o viceversa.
(X). Un autómata de tipo LNFA o NFA al que se le aplique la minimización.

6. Ejemplos

Ejemplo 1: Creación de un autómata AFD, al cual se lo nombra *Automata1*, con los estados definidos en *states*, el alfabeto indicado en *alphabet* y las transiciones en *transitions*. Luego, dado que el autómata es *var* se puede minimizar. Entonces, se emplea *minimize()* y al ser la última modificación que se aplica al autómata, esta versión será lo que se muestre en Latex.

```
// Creamos un autómata AFD
var DFA Automata1: [

      // Declaramos los estados
     states: {
           <r>: u,            // Estado "regular" (no inicial ni final)
           <i>: s ,           // Estado inicial
           <f>: { w, q },     // Estados finales
     }
```

```
        // Definimos el alfabeto
        alphabet: {a, b}

        // Declaramos las transiciones
        transitions: {
                s-b->u ,
                w<-b->u,            // Equivale a w-b->u y u-b->w
                {s,u}-a->w,         // Equivale a s-a->w y u-a->w
                q-{a,b}->w,
                w-a->q
        }
]
minimize(Automata1)                 // Minimizo Automata1
```

Ejemplo 2: Creación de un autómata AFND, al cual se lo nombra *Automata2*, y de uno AFND-Lambda, *Automata3*, con los estados definidos en *states*, el alfabeto indicado en *alphabet* y las transiciones en *transitions*. Se agregan variables y constantes globales del tipo state, transitions y alphabet para definir estados, transiciones y alfabetos respectivamente que puedan ser utilizados por cualquier autómata. Para definirlas se hace uso de los operadores y también se muestran funciones definidas por el usuario del lenguaje que pueden modificar un autómata o algún conjunto almacenado como *var* o *const*. Los autómatas Automata2 y Automata3 aparecerán en el archivo final de LaTeX.

```
// Definimos valores constantes y variables que podrán ser empleados para crear
autómatas
// la letra inicial del nombre de dichas constantes debe estar en mayúscula

const states Q: { r, o, j }
const transitions R:  s-a->{w,q}    // = {s-a->w, s-a->q}
var alphabet A: { a,b }
const states T: t                   // t = {t}

// Creamos un autómata AFND
var NFA Automata2: [
        states: {
                <r>: Q,
                <i>: s ,
                <f>: { w, q }           // Estado trampa no fue definido, entonces
                                se nombrará a dicho estado como 'TRAP' en la salida
        }

        alphabet: A

        transitions: {
                q-a->r ,
                w<-b->r,
```

```
            R
      }
]

// Creamos nuevas constantes y redefinimos variables para armar el nuevo
autómata
// Usamos operadores
const states H: { h, o }
A: A + { cc, dd }                    // ⇔ A ∪ {cc, dd}
const transitions S1: {r-a->j, r-b->r, j-{a,b}->r}
const transitions S2: {j-a->r}
const transitions S3: S1 - S2        // ⇔ {r-a->j, r-b->j, j-b->r}
const states KO: {k, o}

// Creamos un autómata AFND-lambda
const LNFA Automata3: [
      states: {
            <r>: (Q^H)+(KO^T),       // = ({r,o,j} ∩ {h,o}) ∪ ({k,o} ∩ {t})
                                     = {r,o,j} ∩ {h,o}  = {r,j}
            <i>: s ,
            <f>: { w, q },
            <t>: bin                 // llamo al estado terminal 'bin'
      }

      alphabet: A

      transitions: {
            w<-@->r,    // = w consume lambda para llegar a r y viceversa
            s-cc->{w,q},
            w-{cc,dd}->r, S3
      }
]

def complement (in-out DFA A) [
      const states Q: A.states
      const states S: VOID
      for every element in Q.final do [
        S: S + {element}
      ]
      A.states.final: Q.regular
      A.states.regular: S
]

complement(Automata3)

def takeFinalStatesIncludedInW (in DFA A, in states W, out states Q) [
      Q: A.states
      for every element in Q.final[
        where (element is in W and element is not in Q.initial) do [
```

```
            Q.final: Q.final - element
          ]
      ]
]
```