



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

GARZÓN DOMÍNGUEZ GERARDO ISMAEL

2CV13

TEORÍA COMPUTACIONAL

PRÁCTICA 5:
AFN CONFIGURABLE

INTRODUCCIÓN

Si examinamos los autómatas finitos deterministas, notamos que la característica en común que tienen, es que, la función de transición, δ , define una sola transición para cada par estado-símbolo. En tal caso, la función δ es llamada una función total, y es por ello que estos autómatas son deterministas. [1]

Para los autómatas finitos no deterministas, se agrega la característica de que, para cada estado y símbolo, puede haber más de una transición definida. Formalmente, esto se puede lograr definiendo la función de transición tal que para un par estado-símbolo, esta tenga un rango el cual es un conjunto de estados. [1]

Un autómata finito no determinista esta definido mediante la 5-tupla: $M = (Q, \Sigma, \delta, q_0, F)$. [1]

Dónde:

Q : El conjunto finito de estados internos del autómata.

Σ : El alfabeto de entrada del autómata, toda cadena de entrada debe estar construida sobre este. Para que pueda ser procesada.

δ : La función de transición: $\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$. Una función que mapea un par (q_i, a) , $q_i \in Q, a \in \Sigma \cup \{\lambda\}$ a un subconjunto del conjunto potencia de Q .

$q_0 \in Q$: Un elemento perteneciente al conjunto de estados del autómata, se considera como el estado inicial del autómata, es decir desde este estado comenzaría su funcionamiento al procesar una cadena.

$F \subseteq Q$: Subconjunto del conjunto de estados internos del autómata. Este subconjunto representa los estados finales válidos del autómata, es decir que las cadenas que lleven al autómata a este estado como paso final, serán aceptadas. [1]

Como se puede observar, los autómata finitos deterministas, se diferencian de los autómatas finitos no deterministas, por la manera en que está definida la función de transición. A continuación se describirá cual es el mecanismo de operación de un autómata finito no determinista (abreviado como AFN):

De manera similar al AFD, un AFN, se encontrará inicialmente en el estado q_0 , a partir de este estado, leerá un símbolo de la cadena y se tomará el resultado de la función δ . Dado que el resultado es un subconjunto de 2^Q , se tomará cada estado del subconjunto como una opción para moverse al siguiente estado, para el cual se repite el proceso (recursividad). Nótese también, que dada la definición de la función δ , se puede aceptar la cadena vacía como argumento para δ . De esta manera el autómata podrá moverse de un estado a otro sin la necesidad de consumir un símbolo de la cadena. [1]

Para representar un AFN mediante un grafo de transiciones $G = (V, E)$, tal que V es el conjunto de vértices del grafo, y E es el conjunto de aristas que unen a los vértices, las aristas pueden ser representadas como pares no ordenados.

Para este caso en particular, si $M = (Q, \Sigma, \delta, q_0, F)$, un autómata finito no determinista. Entonces su grafo de transición asociado G_M será un grafo dirigido que tendrá al menos $|Q|$ vértices, esto es, que los

vértices están definidos por Q , mientras que, las aristas definidas como (q_i, q_j) y etiquetadas con un símbolo $a \in \Sigma \cup \{\lambda\}$, se encontrarán en el grafo si y solo si $q_j \in \delta(q_i, a)$. [1]. Esto implica que en un AFN pueden existir vértices etiquetados como λ , que se les conoce como transición- λ . [1]
El resto de la notación es idéntica a la notación para grafos de AFD's.

Nótese que un diagrama de transiciones se puede generar a partir de la tabulación de la función δ , como ya se menciono dado un par (q_i, a) y $q_j \in 2^Q$. Así que dicha función δ se puede visualizar mediante:

Estados / Símbolo	a
q_i	q_j

Figura 1.0 Forma tabular de una función de transición δ

La tabla se puede expandir para $|Q|$ renglones y $|\Sigma \cup \{\lambda\}|$ columnas.

Por lo tanto una cadena es aceptada por una AFN si hay alguna secuencia de posibles movimientos que lleve al autómata a un estado de aceptación $q_f \in F$ al consumir todos los símbolos de la cadena. Una cadena se rechaza sólo si no hay una secuencia posible de Movimientos mediante tales que lleven al autómata a un estado final valido $q_f \in F$. [1]

La siguiente figura es un ejemplo de un autómata finito no determinista en la forma de un grafo dirigido.

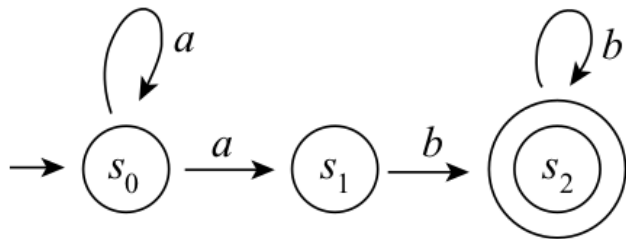


Figura 1.1: Autómata finito no determinista en la forma de un grafo dirigido. [2]

DESARROLLO E IMPLEMENTACIÓN

En esta práctica, se realizará un programa que genere un autómata de manera dinámica. Con “dinámica” se hace referencia a que la definición del autómata (en especial la función de transición) no es estática / programada fuertemente; o sea que no se encuentra dentro del código, sino que se genera al momento de ejecutar el programa dado un archivo de entrada. Esto dará la posibilidad de generar autómatas sin la necesidad de re-compilar para cada modificación que se le realice a este.

A continuación se describirá el razonamiento detrás del funcionamiento del programa. También se mostrarán y explicarán las funciones claves que permiten que este programa sea posible.

Nota: el término “nul terminado/terminada” significa que al final de una secuencia de bytes se encuentra el byte con valor 0.

Una de las funciones nuevas es *substr_find*, la cual busca la primera ocurrencia de una subcadena, dentro de una cadena. En seguida, se describe el funcionamiento interno de esta función:

```
char* substr_find(char* str, const char* seq)
{
    const char *save_sep = seq, nul = '\0';
    char* save_str = NULL;

    if(*seq == nul)
        return save_str;

    while(1)
    {
        seq = save_sep;
        while(*str == *seq && *str != nul && *seq != nul)
            ++str, ++seq;

        if(*seq == nul)
        {
            save_str = str - (seq - save_sep);
            break;
        }

        if(*str != nul)
            ++str;
        else
            break;
    }

    return save_str;
}
```

Notemos primero que, se guarda el apuntador donde inicia la secuencia *seq*. También, se comprueba si la función se encuentra en un caso limite donde *seq* es la cadena vacía, en este caso, se regresa *NULL*, ya que se considera indeterminable en que parte de la secuencia *str* se encuentra una subcadena vacía, pues en teoría, hay infinitas de esta en cualquier punto de cualquier cadena. Ahora, analicemos como se busca una subcadena dentro de una cadena: Ambos bucles *while* terminarán si se llega al byte nul en alguna de las dos secuencias. Si se observa el *while* interno, observamos que este también itera sobre *str* y *seq*, esto mientras los bytes de ambas secuencias sean iguales.

Figura 2.0: Definición de la función *substr_find*, para buscar subcadenas dentro de una cadena.

Nótese que en el *while* externo se comprueba si el *while* interno finalizó debido a que se llegó al valor nul en *seq*, esto significa que *seq* es una subcadena de *str*, este caso es también una de las condiciones de salida del *while* externo, antes de salir, se calcula en que posición es que comienza esta subcadena, esto mediante el apuntador al inicio de *seq* que se guardó al comenzar la función, los bytes de diferencia entre el apuntador al valor nul de *seq* y el inicio son la longitud de esta secuencia, si se le resta al apuntador donde *str* se encuentra, se llega al punto donde la subcadena *seq* comienza dentro de *str*. En caso contrario y de que *str* aun no apunte a su byte nul, entonces se continua buscando la subcadena pero esta vez desde el siguiente carácter. Si *seq* no es subcadena de *str*, se regresa *NULL*.

La función de la figura 2.0, es parte esencial de la nueva función *str_sep*, la cual es introducida en el modulo de cadenas, podría decirse que esta hace el trabajo duro para que sea posible generar un autómata de manera dinámica. A grandes rasgos, esta función tokeniza [3] una cadena, es decir, genera una subcadena cada que se encuentra un delimitador (otra subcadena) dentro de esta, generando una lista de subcadenas de la cadena original. A continuación se explica con mayor detalle.

```
array_t* str_sep(char* str, const char* sep)
{
    array_t* string_list = array_new(1);
    size_t strl = my_strlen(str), sepl = my_strlen(sep);

    const char nul = '\0', *end_str = str+strl;
    char* t = NULL;

    while(t = substr_find(str, sep), str <= end_str)
    {
        if(str != t)
            array_add(string_list, str);

        if(t != NULL)
            *t = nul, str = t + sepl;
        else
            break;
    }

    size_t last = get_size(string_list)-1;
    if(my_strlen(get_item(string_list, last)) != 0)
        last += 1;
    set_item(string_list, last, NULL);

    return string_list;
}
```

Figura 2.1: Definición de la función *str_sep*, para tokenizar una cadena.

nul terminada, además *str* se incrementa por la longitud del delimitador, así, este apunta ahora al menos 1 byte después del delimitador dentro de *str*, por lo que *str* es un sufijo del valor original/previo de *str* entonces, el bucle se repite para volver a buscar a *sep* en la secuencia restante de *str*, o comprobar si es momento de salir del bucle debido a que se llego por lo menos 1 byte después de la secuencia *str* original (la que llega como argumento de la función). En caso de que *seq* no exista en *str*, el bucle termina, y efectivamente se tiene toda la cadena *str* agregada a la lista.

Por último, se observa que se agrega un centinela *NULL* que es útil para crear conjuntos de cadenas si es necesario. Nótese que por la manera en la que está descrito el algoritmo para *tokenizar*, se agregará una cadena vacía al final de la lista cuando la secuencia *str*, que se le pase a la función, contenga el delimitador al final de la cadena, en dado caso, se reemplaza dicho elemento por el centinela *NULL*. Otro detalle importante, es que la secuencia *str* es mutada, por lo cual, *str* deberá apuntar a bloques de memoria inicializados de manera dinámica, o bloques de memoria que no pertenezcan a la sección de “solo lectura” del programa. La siguiente figura ilustra el resultado de una cadena tokenizada y el resultado de la función 2.1.

```
, , T H E R E , I S , N O , P A I N , nul
nul nul T H E R E nul I S nul N O nul P A I N nul nul
```

Figura 2.2: Dada la secuencia “„THERE,IS,NO,PAIN,.”. El primer renglón representa la cadena *str* pasada como argumento a la función *str_sep*, el segundo renglón muestra la mutación la cadena tras procesarse dicha función con el separador “,”.

El argumento *str* es la secuencia a tokenizar, la secuencia *sep*, es el delimitador que se buscará dentro de *str*.

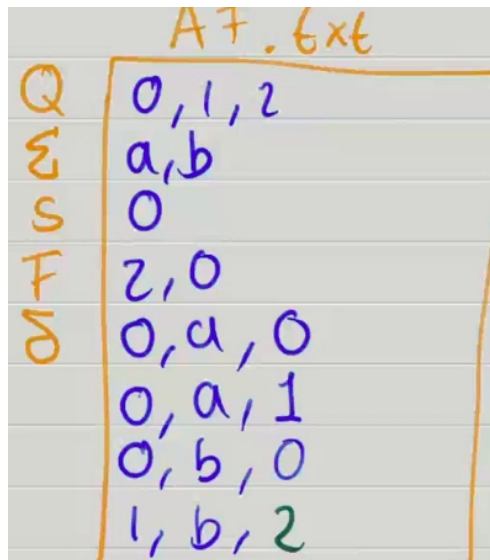
El bucle *while* ejecutará la función *substr_find* y luego evalúa la expresión que controla la salida (cuando *str* apunta después del bloque de memoria que contiene dicha secuencia). En caso de que *str* sea distinto de *t*, significa que el inicio del delimitador *sep*, no se encuentra en donde *str* apunta, por lo cual, esa parte de la secuencia se puede agregar a la lista de tokens. En base al valor de *t*, se puede saber si *sep* existe en *str*, en caso afirmativo, se agrega un terminador nul en esa parte de la cadena, para que la secuencia previamente agregada a la lista sea

La siguiente figura muestra los contenidos del objeto *string_list* (la lista de subcadenas generadas) por la función *str_sep* dados los argumentos explicados en la fiugra 2.2.

void* e0	"THERE\0"
void* e1	"IS\0"
void* e2	"NO\0"
void* e3	"PAIN\0"
Void* e4	NULL

Figura 2.3: Los elementos contenidos en el arreglo *string_list* son apuntadores a una parte de la cadena *str* recién procesada, por lo cual, si se modifica la secuencia *str*, los cambios se verán reflejados en los elementos del arreglo *string_list*.

Ahora se discutirá el funcionamiento del programa. Se explicarán las partes más elementales unicamente.



A7.txt

Q	0, 1, 2
Σ	a, b
S	0
F	2, 0
δ	0, a, 0
	0, a, 1
	0, b, 0
	1, b, 2

La primera característica a notar es que, es un archivo de texto, aunque en general, eso solo hace una diferencia en plataformas como *windows*, por los saltos de linea. En el programa se consideraron dos casos, saltos de línea de sistemas operativos **nix*: "*\n*" y *windows* "*\r\n*". (en general, debería de funcionar correctamente, pero el programa solo fue probado en *linux*). La siguiente características son: el renglón 1 corresponde al conjunto de estados internos del autómata, el segundo al alfabeto de entrada, el tercero, es el estado inicial, y el cuarto, el conjunto de estados de aceptación. Desde el 5 hasta el ultimo renglón, se espera la definición de la función delta.

Algunos aspectos a notar en esta implementación son:

Figura 2.4: Ejemplo de un archivo reconocible por el programa. Todos los archivos con este formato deberán poder ser procesados.

Se considera que la definición del autómata no tiene errores, pues no se comprobará si los estados finales, o el inicial, corresponden a los estados del renglón 1. El autómata espera por lo menos la definición de los primeros 4 renglones en el archivo de texto. Si no se proveen, es probable que el programa termine con errores. Si la función sigma no se provee, no habrá manera de determinar la validez de las cadenas, así que todas las cadenas introducidas serán rechazadas. Para definir la función delta desde el archivo de texto, se espera desde 1 a n renglones conformados por al menos 2 elementos y como máximo 3. Si se introducen 2 elementos, se considerará que el resultado de la función para ese par es el conjunto vacío, si se introducen 3 o más valores, solo se considerará el tercer valor como parte de, o todo el resultado para tal par (estado, símbolo) dependiendo de cuantas veces aparezca dicho par en el archivo, sin importar el orden de aparición. Si se introduce 1 elemento, el programa terminará con errores. En este caso, tampoco se comprueba si los estados resultado pertenecen a los estados del autómata, aunque en general, por la manera en la que se estructuró la función *dynamic_delta* del programa (vea figura 2.6), cuando el autómata llegue a un estado en el cual la función delta está indefinida, entonces se regresa el conjunto vacío, por lo que no se completa el camino para la cadena y

por lo tanto se considerará inválida, por lo que, esto da la posibilidad de que el conjunto vacío sea representado por cualquier estado que no esté en el dominio de la función delta o ningún símbolo.

Otro aspecto a notar, es que los estados son representados por cualquier secuencia de caracteres, ya que un nombre de estado es considerado hasta que se encuentra una coma (vea la figura 2.4).

Con la anterior explicación se podrá proceder a algunos detalles de la implementación de este programa. Se puede decir que el programa tiene 2 fases, primero lee el archivo y genera un contexto en base a este, y la segunda fase permite probar las cadenas para comprobar su validez o invalidez al autómata cargado.

```
typedef struct
{
    struct { const char *st, *sy; }; //key
    array_t* result;
}mapping_t;
```

Una parte importante del programa es el tipo compuesto *mapping_t*, este tipo es una estructura que representa una 3-tupla (estado, símbolo, conjunto de estados).

Figura 2.4: Esta estructura representa una 3-tupla, donde los primeros dos elementos son una llave para poder buscar dentro de un arreglo, (o cualquier tipo de lista lineal en su defecto).

La primera fase le corresponde a dos funciones, la primera lee y carga el texto del archivo en un objeto *str_t*, después se extrae el bloque de memoria que contiene dicho objeto para procesarlo (esto corresponde a la segunda función comentada, y es la que se explicará en breve), usando principalmente la función discutida en la figura 2.2.

La función de interés tiene por nombre *parse_and_create_context*, tiene como objetivo tokenizar los contenidos del archivo de texto y generar un contexto en base a estos, este contexto está representado mediante la estructura:

```
struct context
{
    automata_t* automaton;
    array_t
    *delta_mapping, // array of mapping_t*
    *path, // array of char*
    *empty_set; // array void*
    str_t* expr;
}context;
```

Figura 2.5: Esta variable global representa el contexto de ejecución del programa, contiene el autómata generado, un arreglo de mapeos (fig 2.4) usado por la función delta (fig 2.6), el camino actual que el autómata está atravesando, y la expresión actual con la que se está probando el autómata. También contiene el conjunto vacío (útil para la función delta)

De manera breve, puede decirse que la función *parse_and_create_context*, sigue los siguientes pasos:

1. Dados los contenidos del archivo como una secuencia de caracteres nul terminada. A partir de dicha secuencia se genera un arreglo de subcadenas de esta usando *str_sep* (fig 2.1). Se espera el delimitador nueva línea del sistema (“\n” para sistemas **nix*, “\r\n” en *windows*).
2. De los primeros 4 renglones se generan los estados, el alfabeto, el estado inicial, y estados finales.
3. Del resto de renglones se generan n objetos de tipo *mapping_t* los cuales se agregan al elemento del contexto: *delta_mapping*.

Tras generar el contexto se puede pasar a la fase de prueba. Esta fase claramente depende de la función delta, en la implementación, esta función es llamada *dynamic_delta*, y como se mencionó anteriormente, esta depende del arreglo *delta_mapping*. Esta función es bastante simple:

```

array_t* dynamic_delta(void* obj, const char* symbol)
{
    mapping_t key = { {obj, symbol}, NULL }, *cast = NULL;
    void** m = find_item(context.delta_mapping, &key, 0, key_cmp);

    array_t* retval = NULL;
    if(m != NULL)
    {
        cast = *m;
        retval = cast->result;
    }
    else
        retval = context.empty_set;

    return retval;
}

```

Esta función generará, un objeto tipo *mapping_t* a partir de los argumentos recibidos (estado, simbolo), en este objeto solo es significativo el primer elemento, que se considera la llave de búsqueda. Después se realiza una búsqueda lineal comparando las llaves en el arreglo *delta_mapping* con la llave generada.

Figura 2.6: Función *dynamic_delta*, esta es la función delta que el autómata del contexto recibe al crearse.

Si se encuentra el elemento (delta definida para el un par (estado, simbolo)), la función *find_item* regresa un apuntador a la posición dentro del arreglo donde está contenido este elemento, así que hace falta de-referenciar para obtener el elemento en cuestión, en este caso *dynamic_delta* regresa el conjunto de estados correspondientes al par. Si no se encuentra el elemento (delta indefinida para el par (estado, símbolo)), *find_item* regresa *NULL*, en este caso *dynamic_delta* regresa el conjunto vacío. Podría decirse que el hecho de que se regrese el conjunto vacío es una forma de “completar” el autómata, con la excepción de que el conjunto vacío no es un pozo, y la cadena no se sigue procesando, lo cual basta para los propósitos de esta práctica, ya que solo es necesario encontrar caminos validos.

Ahora podemos proceder a la parte de prueba del programa. Se mostrarán las instrucciones de como ejecutar el programa así como un ejemplo, (el cual es el autómata de la práctica anterior, pero esta vez definido en el archivo de texto *afn.txt*)

El programa necesita un segundo argumento al llamarlo desde la terminal, este argumento es la ruta del archivo donde se encuentra la definición del autómata, así que para llamarlo se utilizará el comando: *./automata afn.txt*

Se puede salir del programa con la palabra clave “exit”, y también se puede actualizar el contexto con la palabra “load”, o sea que se repetirá el proceso de lectura y generación de autómata, así, no es necesario terminar e iniciar el programa repetidamente. Cuando el programa reciba valide una cadena para el autómata, se imprimirá esta y los caminos que llegaron al estado de validez, de lo contrario no se imprimirá nada.

Estados/símbolos	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_3\}$	$\{q_2\}$
$*q_2$	$\{q_3\}$	$\{q_3\}$
q_3	$\{q_3\}$	$\{q_3\}$

Figura 2.7. Función de transiciones para el autómata dado en el archivo de texto 1.txt, A continuación se probarán las cadenas “aaab” y “ababa” en este autómata.

```
>> aaab
Cadena: "aaab" validada con el camino:
>>(0)>>(0)>>(0)>>(1)>>(2)
```

Figura 2.8. Para la función 2.7, la cadena “aaab” genera un solo camino que lleva a q_2 , hay otros caminos que caen en pozo, como de q_1 a q_3 , para llegar a q_3 , se necesita una que b sea el ultimo símbolo de la cadena.

```
>> ababa
>> |
```

Figura 2.9. Para la función 2.7, la cadena “ababa” es invalida, ya que cuando el autómata se encuentra en la última b de la cadena, el símbolo siguiente lo lleva a q_3 , que no es un estado de aceptación.

Por lo que pude notar, este autómata aceptará expresiones que comiencen con cualquier cantidad de b’s, seguidas de cualquier cadena formada por los símbolos a, b, y cuyos últimos dos símbolos sean la cadena “ab”.

Estados/símbolos	a	b
$\rightarrow q_1$	$\{q_2, q_5\}$	$\{q_4\}$
q_2	$\{q_2\}$	$\{q_3\}$
$*q_3$	$\{q_6\}$	$\{q_6\}$
$*q_4$	$\{q_6\}$	$\{q_6\}$
$*q_5$	$\{q_6\}$	$\{q_5\}$
q_6	$\{q_6\}$	$\{q_6\}$

Figura 2.10. Función de transiciones para el autómata dado en el archivo de texto 2.txt. En este autómata se probarán las cadenas “ab”, “aaab”, “abab”.

```
>> ab
Cadena: "ab" validada con el camino:
>>(1)>>(2)>>(3)

Cadena: "ab" validada con el camino:
>>(1)>>(5)>>(5)
```

Figura 2.11. Para el autómata dado en 2.10, la cadena “ab” generará dos caminos válidos.

```
>> aaab
Cadena: "aaab" validada con el camino:
>>(1)>>(2)>>(2)>>(2)>>(3)
```

Figura 2.12. Para el autómata dado en 2.10, la cadena “aaab” genera un camino válido.

```
>> abab
>> |
```

Figura 2.13. Para el autómata dado en 2.10, la cadena “abab” no es válida, ya que una las cadenas que tienen una b después de una a, llegan al estado de pozo q_6 .

Este autómata aceptará las expresiones formadas sobre el lenguaje $\{a,b\}$ y que no tengan una b precedida por una a.

Estados/ símbolos	a	+	-	.
$\rightarrow q_0$	$\{q_4\}$	$\{q_1\}$	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_1, q_4\}$	ϕ	ϕ	$\{q_2\}$
q_2	$\{q_3\}$	ϕ	ϕ	ϕ
$*q_3$	$\{q_3\}$	ϕ	ϕ	ϕ
q_4	ϕ	ϕ	ϕ	$\{q_3\}$

Figura 2.15. Función de transiciones para el autómata del archivo 3.txt. Este se probará con las cadenas “a.aa”, “+a.a”, “+aa.aa”.

```
>> a.aa
Cadena: "a.aa" validada con el camino:
>>(0)>>(4)>>(3)>>(3)>>(3)
```

Figura 2.16. La cadena “a.aa” es válida con un camino.

```
>> +a.a
Cadena: "+a.a" validada con el camino:
>>(0)>>(1)>>(1)>>(2)>>(3)

Cadena: "+a.a" validada con el camino:
>>(0)>>(1)>>(4)>>(3)>>(3)
```

Figura 2.17. La cadena “+a.a” generará dos caminos válidos para el autómata 2.15, esta cadena es aceptada.

```
>> +aa.aa
Cadena: "+aa.aa" validada con el camino:
>>(0)>>(1)>>(1)>>(1)>>(2)>>(3)>>(3)

Cadena: "+aa.aa" validada con el camino:
>>(0)>>(1)>>(1)>>(4)>>(3)>>(3)>>(3)
```

Figura 2.18. La cadena “+aa.aa” generará dos caminos válidos para el autómata 2.15, esta cadena es aceptada.

Asumiendo el lenguaje de entrada {a,+,-,.}, este autómata aceptará todas las expresiones que opcionalmente comiencen con un + o -, y que a estos símbolos les preceda:

1. Una o más a’s a las cuales les precede un punto al cual le precede cualquier cantidad de a’s.
2. Un punto, al cual le precede una o más a’s.

Con estos ejemplos se nota el uso y la relación entre los autómatas y las expresiones regulares.

CONCLUSIÓN

A pesar de que los conceptos de AFN y AFD ya están claros hay algunas cosas que comentar respecto a esta práctica como que, definitivamente aumento en complejidad, sin embargo, fue posible simplificar y generalizar una solución para este problema conociendo definiciones adecuadas como la *tokenización* de una cadena. Como comentario personal, puedo decir que esta práctica fue útil en el aspecto que me hizo revisar algunas de las funciones que había implementado para el modulo de cadenas y realizar algunas modificaciones para mejorarlas. Algunas de las funciones mejoradas fueron: los constructores de los objetos cadena y la función hash para cadenas, ya que ahora el hash de una cadena se guarda en vez de calcularlo cada vez que se requiera, esto beneficia también ciertas funciones, en especial aquellas para conjuntos, ya que ahora también es posible buscar elementos en un conjunto por simple referencia de un objeto tipo *str_t*, igualmente ahora es posible agregar elementos a un conjunto pasando una referencia a un objeto tipo *str_t* (se agregaron funciones para estos propósitos). También se agrego una función para leer líneas desde *stdin* y cargarlas en objetos *str_t*.

Otra parte que me gustaría recalcar sobre esta práctica, es que, debido a que como la complejidad aumento, se volvió un poco más tedioso mantener registro manual de que objetos están utilizando memoria dinámica, así que opte por crear un nuevo modulo que mantiene registro de los apuntadores a bloques de memoria dinámica, en teoría, esto hace el programa más lento, ya que en sí, este modulo no es tan elaborado como un sistema completo de manejo de memoria, sin embargo, para la escala de estos programas, el tiempo extra que se requiere manejando la memoria es insignificante, así que realice tal cosa más como un ejercicio de aprendizaje y para mantener orden y algo de independencia entre las diferentes partes del programa.

Como comentario extra, cuando se hace alusión a un “objeto”, no se hace referencia a instancias de una clase, o a las abstracciones de la programación orientada a objetos, ya que está no está definida en el lenguaje C, sin embargo, es una manera de abstraer conceptos para poder explicar de manera más conveniente la resolución de los problemas presentados.

```
==146796==
==146796== HEAP SUMMARY:
==146796==      in use at exit: 0 bytes in 0 blocks
==146796==    total heap usage: 169 allocs, 169 frees, 12,817 bytes allocated
==146796==
==146796== All heap blocks were freed -- no leaks are possible
==146796==
==146796== For lists of detected and suppressed errors, rerun with: -s
==146796== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 3.0: El modulo de memoria se encarga de liberar toda la memoria ocupada antes de que el programa termine, ya sea por alguna llamada a la función *exit*, o por que se llega al final de la función *main*. La información de esta imagen se obtuvo con el programa *valgrind*.

REFERENCIAS

- [1] P. Linz, An Introduction to Formal Languages and Automata, 6ta ed. Davis, California: Jones & Bartlett, 2017, pp. 51-52.
- [2] J. Anderson, Automata Theory with Modern Applications. 1ra ed. New York: Cambridge University Press 2006, pp. 44, 51.
- [3] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, "Compilers Principles, Techniques & Tools", 2da ed. Boston, Massachusetts, USA: Addison-Wesley 2007. p. 111.