Trabajo Práctico Transversal

1. Introducción

Una funcionalidad usualmente incorporada en los procesadores de texto es la búsqueda de palabras (cadenas de caracteres) dentro de un texto más grande. Esta funcionalidad se implementa utilizando herramientas que se estudian en esta materia, siendo las expresiones regulares (ER) una de las principales ya que, en base a un alfabeto, permiten definir el patrón o condición que la cadena buscada debe cumplir. Para la implementación de la búsqueda de una cadena dentro de un texto podría ser necesario averiguar si dos ER son equivalentes, para esto se siguen los siguientes pasos:

1. Convertir ambas expresiones regulares en autómatas finitos. Los autómatas finitos resultantes serán, por definición, no deterministas.
2. Convertir los autómatas finitos no deterministas obtenidos en autómatas finitos deterministas.
3. Minimizar ambos autómatas finitos deterministas.
4. Determinar si los autómatas finitos deterministas mínimos que se obtuvieron son isomorfos.

Si el último punto es afirmativo diremos que las ER originales son equivalentes.

El objetivo de este TP no es el desarrollo de todos los pasos antes detallados, solo nos concentraremos en el algoritmo de conversión de un **AFND** en un **AFD.**

Para esta implementación se utilizará:

- La **estructura de datos** propuesta en clases. Esto es muy importante ya que será la misma con la que trabajaremos en el segundo cuatrimestre, cuando veamos en Teoría de la Computación II (TCII) una introducción a la implementación de un lenguaje de programación estructurado.
- El **lenguaje C** dado que la herramienta a usar en TCII incorpora este lenguaje para el desarrollo del interprete y, por lo tanto, es necesario que el TPT se desarrolle con este mismo lenguaje para reutilizar código en el próximo cuatrimestre.

Seguramente se preguntarán por qué no se da la libertad de elegir el lenguaje y las estructuras que cada uno considere apropiadas, la respuesta es simplemente por una necesidad de dar continuidad no solo entre TCI y TCII, sino también entre Programación y TCI. El uso del lenguaje C y de las estructuras aquí propuestas resultarán fundamentales en Teoría de la Computación II, por lo que les pedimos que realicen el esfuerzo necesario para trabajar teniendo en cuenta estas premisas.

Por otra parte, existen entornos de trabajo que brindan múltiples herramientas que facilitan la programación en lenguajes de alto nivel de abstracción y que seguramente serán utilizadas por ustedes el día de mañana. Sin embargo, verán que en TCI y TCII se trabajará en un nivel de abstracción más cercano a la máquina y, dado que posiblemente sea la última oportunidad que tengan de hacerlo antes de adoptar alguno de estos entornos en auge, es una experiencia muy valiosa para su formación como programadores y en consecuencia como Analistas de Sistemas. Tengan en cuenta que a lo largo de su vida profesional se toparán con distintos lenguajes de programación: nuevos, de moda, clásicos, buenos y no tan buenos... y que deberán ser capaces de

adaptarse bajo distintas circunstancias. Será bueno entonces, profundizar el conocimiento del lenguaje C, puesto que nos permite una programación más cercana a la máquina, mediante la manipulación de estructuras básicas y la programación en consola.
Quedan invitados al desafío.

2. Forma de Trabajo

El trabajo se desarrollará en dos etapas, una primera etapa en la que se diseñará un TAD que permita modelar los objetos matemáticos **conjunto** y **lista.** Los elementos de ambos objetos serán cadenas de caracteres, pero a su vez también podrán contar con elementos de tipo lista o conjunto recursivamente con cualquier nivel de anidamiento o profundidad (citaremos ejemplos después de definir sus características).
La segunda etapa del desarrollo consistirá en utilizar el TAD diseñado para programar algunos algoritmos de Autómatas Finitos.

2.1 Primera Etapa

Una cadena es una secuencia de caracteres usualmente encerrada entre comillas dobles, por ejemplo “cuarenta”. En lo que sigue del texto, por una cuestión de practicidad, expresaremos las cadenas sin las comillas dobles.
Las características que debemos tener en cuenta de los objetos matemáticos **conjunto** y **lista** son las siguientes:
Conjunto:

- Se simbolizan con una llave que abre, la secuencia de elementos separados por comas y una llave que cierra. Por ejemplo: {uno, dos, tres}
- No posee elementos repetidos
- No tiene orden, esto es el conjunto {uno, dos, tres} es el mismo conjunto que {dos, uno, tres} o que {tres, dos, uno}
- Aritmética de conjuntos:
  - operaciones binarias: unión, intersección, diferencia y pertenencia
  - operaciones unarias: cardinal

Lista:

- Se simbolizan con un corchete que abre, la lista de elementos separados por comas y un corchete que cierra. Por ejemplo: [uno, dos, tres]
- Puede tener elementos repetidos
- Tiene orden, esto es la lista [uno, dos, tres] es distinta que [dos, uno, tres] o que [tres, dos, uno]
- Aritmética de Listas:
  - Operaciones binarias: agregar elementos por cabeza y por cola, remover elementos por cabeza y por cola, pertenencia
  - Operaciones unarias: tamaño de la lista

Así, ejemplos de los objetos a modelar son los siguientes:

- ➢ Conjuntos y listas simples y homogéneas: {uno, dos, tres} o [uno, dos, tres, uno, tres, dos]
- ➢ Conjuntos y listas anidadas y heterogéneas: {cuatro, [tres, uno, {dos, tres}], {seis, dos}} o [uno, uno, {dos, uno}, [tres, dos, uno]].

El TAD deberá permitir almacenar este tipo de estructuras y brindar las operaciones correspondientes a conjuntos y listas. Para la manipulación de las cadenas utilizaremos las librerías estándares de C como string.h, stdlib.h, etc.

2.2 Segunda Etapa

Los autómatas finitos pueden ser deterministas o no deterministas. En ambos casos su estructura es una 5-upla definida por $(Q, \Sigma, \delta, q_0, F)$ en donde:

Q es el conjunto de estados,

Σ es el conjunto de símbolos de entrada,

q_0 es el estado inicial,

F el conjunto de estados de aceptación

δ cuando el AF es determinista *es una función* de $Qx\Sigma$ en Q y cuando el AF es no determinista *es una relación* en $Qx\Sigma xQ$. Podemos pensar a δ como ternas o lista de tres elementos en la que las dos primeras componentes son el estado y el carácter leído y la tercer componente el o los estados destinos.

Por ejemplo:

Una Transición en un AFD $\delta\{q_0, a\} = q_1$ se puede representar como $\{q_0, a, q_1\}$

Una transición en un AFND $\delta\{q_0, a\} = \{q_1, q_2\}$, se puede representar como $\{q_0, a, \{q_1, q_2\}\}$

Como sabemos, los AFD son casos particulares de un AFND y podríamos pensar en almacenar a las transiciones de un AFD con una terna en la que su tercera componente es un conjunto de un solo elemento. Así, generalizando, se representa δ a como una terna o lista de tres elementos cuyo primer elemento es una cadena (el estado), el segundo elemento una cadena (el símbolo leído) y el tercer elemento un conjunto de cadenas (conjunto de estados destino).

Así un AF puede ser almacenado con la siguiente estructura:

$$[\{q_0, q_1, \ldots, q_n\}, \{a, b\}, \{[q_0, a, \{q_0, q_1\}], \ldots, [q_2, b, \{q_1\}]\}, q_0, \{q_2\}]$$

Donde:

$$Q = \{q_0, q_1, \ldots, q_n\}$$

$$\Sigma = \{a, b\}$$

$$\delta = \{[q_0, a, \{q_0, q_1\}], \ldots, [q_2, b, \{q_1\}]\}$$

$$q_0 = q_0$$

$$F = \{q_2\}$$

2.1. Representación de los datos

La estructura que utilizaremos será un árbol enlazado de manera dinámica. Esta estructura se manipula de manera similar a una lista enlazada simple, como las estudiadas en la asignatura

Programación. Al igual que en una lista enlazada simple, un árbol se forma con nodos. Cada nodo tiene dos hijos, uno izquierdo y uno derecho, que permiten la construcción de las ramas del árbol. En este caso particular, los nodos contarán con dos campos. Un campo de tipo entero, que permitirá definir el tipo de dato almacenado en cada nodo de la estructura (cadena, lista o conjunto). Otro campo será una unión de dos tipos de datos: cadena y árbol. El árbol es el que nos permitirá construir conjuntos y listas heterogéneas y de distinto nivel de anidamiento. En este punto es importante aclarar que no nos preocuparemos por la eficiencia de nuestra implementación. De hecho, la estructura de datos con las que vamos a trabajar no son para nada eficientes. Sin embargo, resultará lo suficientemente clara como para poder operar con ella mediante algoritmos sencillos y portadores de una cierta elegancia.
El componente básico de la estructura de árbol para almacenar los conjuntos y listas es muy simple. El nodo raíz será de tipo entero, y nos indicará el tipo de dato que se encuentra contenido en el árbol. El hijo izquierdo es un puntero al dato almacenado y el hijo derecho es un puntero al siguiente elemento en la estructura. Recordemos que un puntero, en el lenguaje C, es un tipo de dato en el que se almacenan direcciones de memoria. La forma básica del árbol es la que muestra la Figura 1.

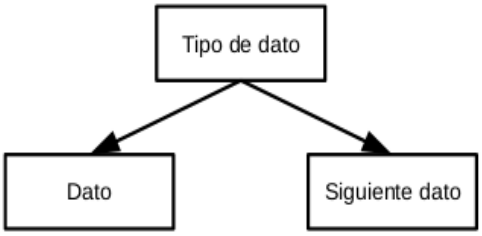
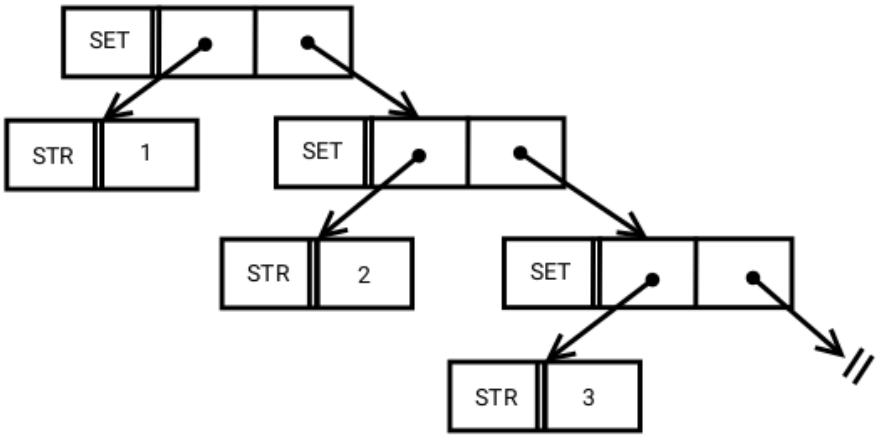


Figura 1: Gráfico básico del árbol para almacenar conjuntos y listas.

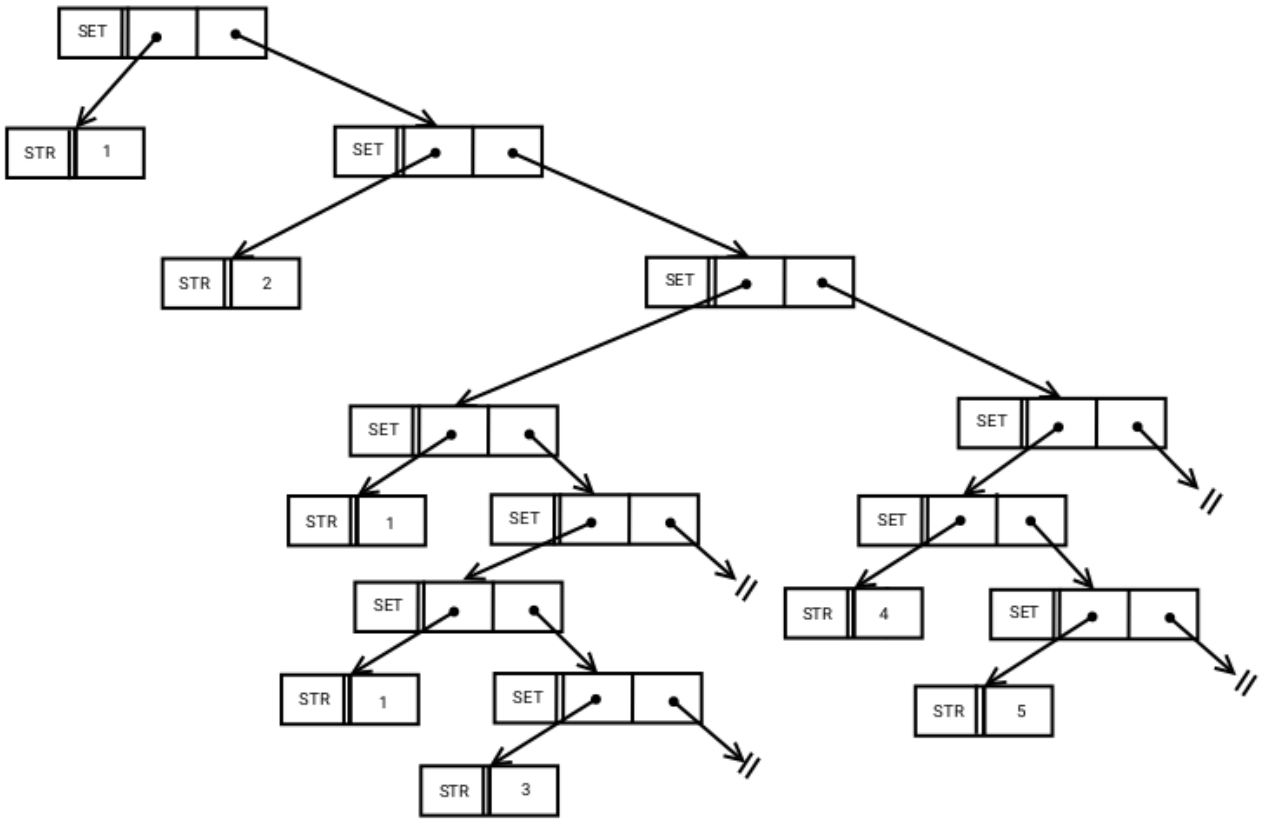
La clave está en que el Tipo de dato nos dará información de qué es lo que sigue en el árbol. La raíz podrá ser de tipo conjunto o lista, el hijo izquierdo apuntará o a una cadena o a un árbol y el hijo derecho apuntará al siguiente elemento.
Veamos un par de ejemplos de cómo se representarían algunos datos con esta estructura.

El primer ejemplo consiste en el conjunto {1,2,3} cuyos elementos son cadenas que, recordemos, omitimos ponerlas entre comillas.



En este caso, el hijo izquierdo de la estructura se completa con un solo nodo de tipo cadena.

El segundo ejemplo es un poco más complejo, se trata de un conjunto heterogéneo que contiene cadenas que representan enteros y conjuntos: {1, 2, {1, {1, 3}}, {4, 5}}



Se puede ver que la estructura es capaz de anidar distintos niveles de conjuntos.
Aunque a la vista parezca algo complejo se propone lo siguiente: observe el árbol y deduzca qué rama debería recorrer hasta el final para saber cuántos elementos tiene el conjunto de primer nivel, y ¿qué debería hacer para conocer la cantidad de elementos de los conjuntos de niveles más profundos?

Ejercicio:

1. Dibuje los siguientes árboles:
   - {cuatro, [tres, uno, {dos, tres}], {seis, dos}}
   - [uno, uno, {dos, uno}, [tres, dos, uno]]

2. Defina el AFD que acepta cadenas sobre {0,1} que representan números naturales divisibles por 3. Dibuje el árbol correspondiente a ese autómata.

3. Estructura de Datos

En el lenguaje C contamos con la posibilidad de declarar estructuras como tipos definidos por el usuario. Utilizaremos la siguiente definición del tipo de dato para el TAD:

```
#define STR 1
#define SET 2
#define LIST 3

struct dataType{
   int nodeType;
   union{
     char *dataStr;
     struct{
         struct dataType *dato;
         struct dataType *siguiente;
     }
   }
}
```

Surge tal vez una pregunta vinculada con la implementación de las operaciones básicas: ¿convendrá ordenar los elementos de un conjunto cuando los conjuntos no tienen a sus elementos ordenados? Dado que no buscamos eficiencia se puede responder esta pregunta tanto por un si como por un no. De cada uno dependerá si prefiere la búsqueda exhaustiva de un elemento en un conjunto o si en cambio prefiere poner más énfasis en un algoritmo que inserte los elementos de manera ordenada.

4. Objetivos a cumplir

La ***primera parte del trabajo práctico transversal*** consiste en implementar el TAD descripto en el apartado 2.1 utilizando la estructura de datos propuesta. Luego implementar un programa que utilice el TAD diseñado y realice las siguientes tareas:

1. Utilizando una única cadena como dato de entrada, cargar una lista o conjunto heterogéneo con un nivel de profundidad arbitrario, por ejemplo:

   {cuatro, [tres, uno, {dos, tres}], {seis, dos,{uno}}}

   En particular debe admitir un AF de la forma

   $$[\{q_0, q_1, \ldots, q_n\}, \{a, b\}, \{[q_0, a, \{q_0, q_1\}], \ldots, [q_2, b, \{q_1\}]\}, q_0, \{q_2\}]$$
2. Crear un árbol de almacenamiento del conjunto o lista ingresado
3. Mostrar el conjunto/lista

La ***segunda parte del práctico transversal*** consiste en agregar las siguientes funcionalidades al programa:

1. Dada una cadena indicar si pertenece al Lenguaje Regular cuyo AF haya sido ingresado.
2. Si el AF ingresado es un AFND convertirlo a AFD. Utilizar el siguiente algoritmo:

$afnd2afd(A) =$

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

$Q_B = \{\{q_0\}\}$

$mientras\ \exists R \in Q_b\ tal\ que\ \delta_B(R,a) esté\ indefinido\ \forall a \in \Sigma$

$\forall a \in \Sigma$

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

$Q_b = Q_b \cup \{\delta_B(R,a)\}$

$F_B = \{S \in Q_b | S \cap F \neq \emptyset\}$

$contestar\ (Q_B, \Sigma, \delta_B, \{q_0\}, F_B)$