Trabajo práctico transversal

Cátedra de Teoría de la Computación I

Año 2018

1. Introducción

Para averiguar si dos expresiones regulares son equivalentes debemos seguir 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 resultantes en autómatas finitos deterministas.
3. Minimizar ambos autómatas finitos deterministas.
4. Determinar si los autómatas finitos deterministas mínimos obtenidos en el paso anterior son isomorfos.

En caso de ser afirmativo el último punto podremos decir que las expresiones regulares originales son equivalentes.

Para recorrer este camino vamos a comenzar por algunos puntos. No se pueden desarrollar todos los algoritmos en un año simplemente porque hay otras muchas cosas que hacer en la materia. **Este año nos concentraremos sólo en el algoritmo de conversión** de un *afnd* en un *afd*, pero con una restricción: se utilizará la estructura de datos sugerida en clases, pues será la misma con la que se trabajará en el segundo cuatrimestre, cuando veamos en Teoría de la Computación II cómo construir un lenguaje de programación. Naturalmente, otra restricción derivada del uso de la estructura que construiremos es el uso del lenguaje C.

En el transcurso del desarrollo del trabajo práctico transversal irán surgiendo problemas que se resolverán en su debido momento. Si algún estudiante encuentra mejores opciones, cuya implementación resulte más sencilla, no habrá problema en aceptar la sugerencia de adoptar dicha implementación, siempre y cuando respete el espíritu de las estructuras de datos básicas y el lenguaje adoptado por la Cátedra. Sin duda alguien se preguntará por qué no se da la libertad de elegir el lenguaje y las estructuras que cada uno considere apropiados y la respuesta es simplemente por una necesidad de

continuidad entre las asignaturas del primer y segundo cuatrimestre. 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 estas premisas en cuenta. Por otra parte, hoy existen entornos de trabajo que brindan múltiples herramientas que facilitan la programación y que seguramente serán utilizadas por todos ustedes el día de mañana, sin embargo, verán que en estas materias se trabajará en un nivel cercano a la máquina dado que posiblemente sea la última oportunidad que tengan de hacerlo antes de adoptar alguno de esos entornos. Tengan en cuenta que en su ciclo de vida como programador 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 escarbar con un poco más de profundidad en estructuras básicas y aventurarse a la programación en consola. Quedan invitados al desafío.

2. Métodos

Las estructuras consisten en árboles cuya raíz nos indicará de qué tipo de dato se trata. Pueden ser datos simples (un número o un carácter), o datos estructurados (cadenas, listas o conjuntos). En esta etapa **no nos estamos preocupando por la eficiencia**. De hecho, las estructuras de datos con las que vamos a trabajar no son para nada eficientes. Sin embargo, resultarán lo suficientemente claras como para poder operar con ellas mediante algoritmos sencillos y portadores de una cierta elegancia. Los datos estructurados podrán contener otros tipos estructurados o tipos simples. En caso de contener datos estructurados, estos podrán tener varios niveles de anidamiento. Los datos simples sólo utilizarán la parte izquierda del árbol (que será una hoja, ya que no habrá más que dos niveles en un dato simple: la raíz y las hojas).

2.1. Estructura de datos

El componente básico de la estructura es muy simple, el nodo raíz siempre será de tipo entero, y nos indicará el tipo de dato que se encuentra contenido en el árbol. Recordemos que un puntero, en el lenguaje C, simplemente está dirigido a una posición de memoria. Naturalmente es importante saber qué tipo de dato se encuentra almacenado en ese lugar, sin embargo, si en este árbol hacemos que la raíz sea de un tipo determinado **siempre**, entonces podremos indicar en ese espacio *qué es lo que sigue para abajo en el árbol*. La forma básica de *cualquier* tipo de dato es la que muestra la Figura 1.

La clave está en que el *Tipo de dato* nos va a dar información de qué es lo que sigue en el árbol. La raíz será de tipo `int`, pero lo que siga en las ramas

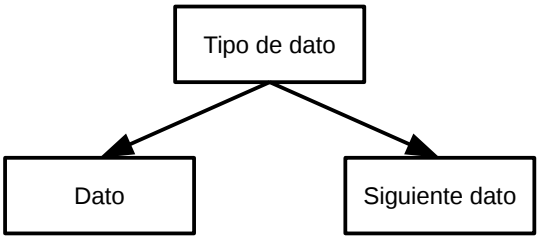


Figura 1: Gráfico básico de el tipo base de la estructura de datos. Este árbol puede representar **cualquier** tipo.

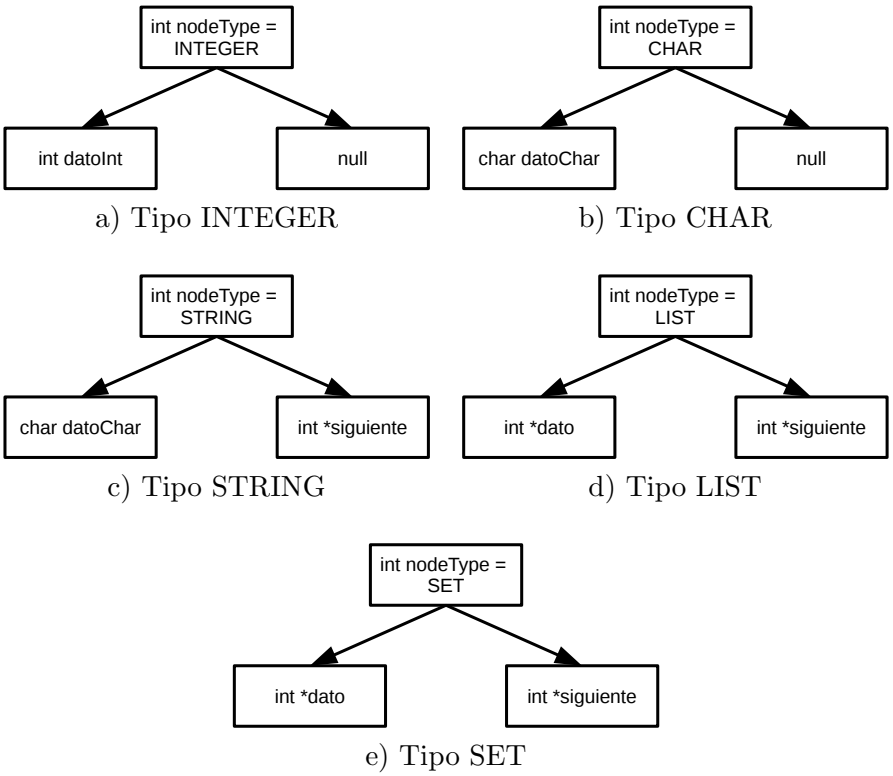


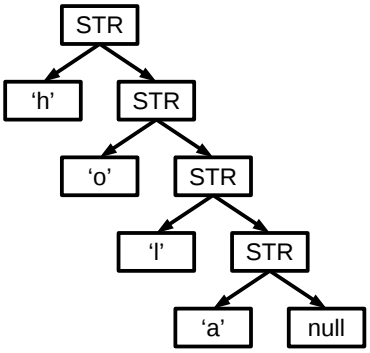
Figura 2: Los distintos tipos soportados por la estructura base.

dependerá justamente del tipo declarado allí. Recordemos que necesitamos dos tipos de datos primitivos (enteros y caracteres) y tres tipos de datos estructurados (cadenas, listas y conjuntos). Las listas y las cadenas tendrán una forma similar pero se distinguirán para poder operar con ellas de manera diferenciada. En la Figura 2 encontramos representados los distintos tipos.

2.2. Ejemplo

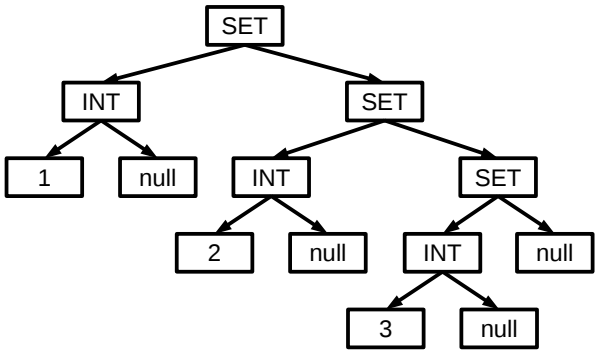
Veamos a continuación algunos ejemplos de cómo se presentarían con estas estructuras algunos datos.

- El primer ejemplo consiste en la cadena "`hola`"



  En este caso simplemente se trata de una secuencia de caracteres. Quizá se preguntará si no sería más simple que el hijo izquierdo fuera un puntero a una secuencia de caracteres tal como las que usamos habitualmente. En realidad tiene que ver con una generalización: así se representarán los enteros sin límite al extender las estructuras para trabajar con aritmética de grandes números en el lenguaje que se está desarrollando en Teoría de la Computación II, por lo tanto, por ahora respetaremos esta manera de representar las cadenas.

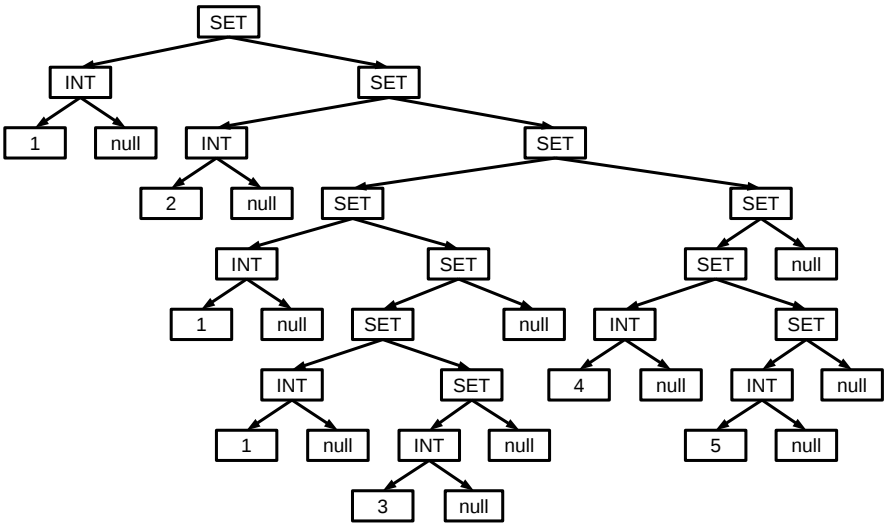
- El segundo ejemplo es el conjunto $\{1, 2, 3\}$



  Aquí se puede observar la construcción de un conjunto homogéneo en el que sólo se contienen enteros. En este caso, el hijo izquierdo de la estructura para conjunto se completa con el entero `datoInt` que justamente indicará que el elemento actual del conjunto es un número entero.

- El tercer ejemplo es un poco más complejo, se trata de un conjunto heterogéneo que contiene enteros y conjuntos: $\{1, 2, \{1, \{1, 3\}\}, \{4, 5\}\}$

  En este caso 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 cómo debería hacer para conocer la cantidad de elementos de los conjuntos de niveles más profundos?.

- Se propone el siguiente ejercicio: realice la gráfica del árbol que debe construirse para la expresión ({“q0”, “q1”}, {’a’, ’b’}, {(“q0”, ’a’, “q0”), (“q0”, ’b’, “q1”) (“q1”, ’b’, “q1”), “q0”, {“q1”}). Las comillas simples y las comillas dobles no deben insertarse en el árbol, sólo sirven para indicar si se trata de un carácter o de una cadena.

De este modo, tenemos disponibles estructuras de datos para operar con enteros, caracteres, cadenas, listas y conjuntos. Veremos a continuación cuáles son las operaciones que nos interesan para cada uno de estos tipos de datos.

3. Desarrollo

Se deberán desarrollar algoritmos para concatenar caracteres y cadenas, realizar la unión de conjuntos, buscar un elemento en un conjunto, así como obtener un elemento de una lista y modificar o insertar un elemento en una lista. Para realizar todas estas operaciones trabajaremos con el lenguaje de programación C.

3.1. Lenguaje C

En el lenguaje C contamos con la posibilidad de declarar estructuras como tipos definidos por el usuario. Utilizaremos las siguientes definiciones para los distintos tipos de datos:

```
#define INT 1024
#define CHAR 1025
#define STRING 1032
#define LIST 1033
#define SET 1034
```

La estructura por defecto es la `dataType`, la definición de la misma es la siguiente:

```
struct dataType{
   int nodeType;
   struct dataType *dato;
   struct dataType *siguiente;
}
```

A partir de este tipo de dato base se construyen los tipos de datos disponibles, por ejemplo:

```
struct intType{
   int nodeType;
   int valor;
}
```

ó

```
struct setType{
   int nodeType;
   struct dataType *dato;
   struct dataType *siguiente;
}
```

Pensándolo bien el tipo conjunto y el tipo lista responderán a la misma estructura, lo cual es razonable dado que la estructura `dataType` trata justamente de ser lo suficientemente general para abarcar los tipos simples y los estructurados.

A no asustarse, parece complicado pero no lo será tanto cuando nos adentremos en su desarrollo.

Queda una última pregunta: ¿convendrá ordenar los elementos de un conjunto? 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.

3.2. Algoritmos

Los algoritmos para realizar uniones de conjuntos, búsqueda de elementos en un conjunto, o concatenación de caracteres y cadenas forman parte del desarrollo del práctico y deberán ser realizados individualmente.

- La primera parte del práctico transversal consistirá en implementar las estructuras de datos y permitir la carga de un autómata finito, la carga de cadenas y la corrida de un autómata para ver si acepta o no una cadena determinada.

  Esta primera parte deberá entregarse antes del primer parcial y formará parte del mismo como un bloque y como tal deberá recuperarse en caso de no ser aprobado.

- La segunda parte consistirá en implementar el algoritmo de conversión de un *afnd* en el *afd* equivalente, cuyo pseudocódigo se muestra a continuación:

  $afnd2afd(A) =$
  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 $\forall a \in \Sigma$
  $\forall a \in \Sigma$
  $\delta_B(R, a) = \bigcup_{q \in R} \delta(q, a)$
  $Q_B = Q_B \cup \{\delta_B(R, a)\}$
  $F_B = \{S \in Q_B | S \cap F \neq \emptyset\}$
  contestar $(Q_B, \Sigma, \delta_B, \{q_0\}, F_B)$

  Este segundo algoritmo deberá estar implementado para el segundo parcial y se evaluará de la misma manera en el bloque de programación correspondiente