Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

Trabajo Práctico 1

Implementación de AFDs

Teoría de Lenguajes

Primer Cuatrimestre de 2015

Grupo: Autores del Autómata Automático Autodestructivo

Apellido y Nombre	LU	E-mail
Matayoshi, Leandro	79/11	leandro.matayoshi@gmail.com
Unbekant, Ignacio	722/10	ignacio.niesz@gmail.com
Vega, Leandro	698/11	leandrogvega@gmail.com

Índice

1. Descripción de la Implementación

1.1. Parser de la regex

Las expresiones regulares están representadas por una clase a la cual llamamos Regex. Cada instancia tiene 3 atributos:

- nombre = ' OR, CONCAT, . . . ,SIMOBOLO'
- valor = 'x' en el caso en el que la regex representa un símbolo (con x perteneciente al listado de caracteres válido), ' ' en el resto de los casos.
- argumentos = un arreglo con una única regex si la regex es unaria (PLUS, STAR), con n regex para CONCAT y OR, o ' ' en el caso de los símbolos.

La función que parsea la regex: "parsear_regex" se llama recursivamente por cada uno de los argumentos, manteniendo el estado del archivo

1.2. Armado del autómata finito:

Para realizar esta función nosotros partimos de un lenguaje regular parseado, el cual está explicado en la sección **Parseo de Regex**. A este mismo lo leeremos e iremos armando el autómata finito correspondiente. La forma de generar nuestro autómata finito es por medio del método de Thompson. Básicamente el algoritmo funciona de forma recursiva y lo detallaremos a continuacion:

- Se fija si debe realizar *CONCAT*, *PLUS*, *STAR*, *OR*, *OPT* o *SIMBOLO*.
- Si es *SIMBOLO* crea un automata que tiene dos estados y una transición con el símbolo correspondiente.
- En caso de no ser *SIMBOLO* va mirando los argumentos y, en cada caso, llama recursivamente a la función para que cree el autómata del mismo. Luego dentro del nivel en el que nos encontramos usamos la funcion *armarConcar*, *armarPlus* o *armarOr* según corresponda, de esta manera, cada una se encargarán de armar el autómata tal como lo describe Thompson. Cabe destacar que no hemos hecho un *armarStar* ni *armarOpt*, esto se debe a que el primero sólo agrega una transición más con respecto a *armarPlus*, y el segundo es un caso idéntico al *armarOr* definiendo uno de los parámetros como un automata con transición lambda.
- En el caso del *CONCAT* y *OR* que pueden tener n argumentos, hemos optado que el *armarConcat* y el *armarOr* sólo puedan ir armando de a dos autómatas y luego, al obtener el armado de ambos, buscar recurisvamente el autómata del siguiente argumento y repite el paso anterior sucesivamente.

1.3. Determinizar Autómata:

En esta función usamos el algoritmo visto en la clase práctica, calculamos las clausuras para cada estado. Para poder obtener la primera clausura obtenemos el estado inicial y sobre ese aplicamos una clausura lambda y obtenemos el primer conjunto. Luego sobre este conjunto aplicamos la clausura para cada alfabeto del autómata no determinístico, sin contar el alfabeto "*lambda*".
Continuamos analizando como nos indica el algoritmo analizando cada conjunto conseguido para cada alfabeto, siempre y cuando no lo hayamos analizado ya. Finalizando cuando hayamos analizado todos sin obtener nuevos conjuntos sin analizar.
Dejando la teoría del algoritmo a un lado y basandonos en la implementacion, vamos a usar una cola en donde los nuevos conjuntos que vamos a ir consiguiendo lo vamos a meter en la cola, siempre y cuando no se haya analizado ni esté ya dentro de la cola. Luego sacaremos la cabeza de la cola y procederemos a analizar dicho conjunto como se describe en la teoría del algoritmo mencionada.
Además vamos a tener una tabla en donde cada fila es una clausura y cada columna son las letras del alfabeto, de esta forma a pesar de ir desencolando de la cola no vamos a perder las clausuras ya analizadas, por que las necesitamos luego para armar el autómata determinístco. Armar el autómata es

sencillo, leeremos la tabla y resolveremos de la misma forma que lo hicimos en clase, poniendo todas las clausuras como estados, el estado inicial será la primer clausura, los estados finales los que en su clausura tienen un estado final del autómata no determinístico, y las transiciones uniendo la tabla, para el estado de la clausura y un alfabeto con la clausura que este en su casilla de la tabla.

1.4. Minimizar Autómata:

Para minimizar un autómata vamos a usar el algoritmo de minimizar basado en conjuntos. Para esto vamos a empezar partiendo los conjuntos de estados en clases, inicialmente se separaran en estados finales y no finales.
Luego, el algoritmo nos indicaba que deberíamos armar una tabla para cada clase, en donde cada fila es un estado perteneciente a la clase y cada columna es el alfabeto del autómata inicial. Y en cada casillero vamos a indicar el conjunto al que pertenece el estado destino de la transicion del estado fila y el alfabeto columna.
Finalizadas todas las tablas de cada clase, miraremos si dentro de la tabla agarrando cada estado si las filas son iguales o distintas. Si son distintas habra una nueva clase donde uno de los estados estará en esta nueva clase, y el otro en una nueva clase, donde agruparemos los que tengan las mismas filas que el primer estado en la primer clase nueva, y los que tengan las mismas fila que el segundo estado en la segunda clase nueva, y asi sucesivamente.
La implementación en este caso es bastante trivial a la explicación dada, en donde vamos a armar dicha tablas para cada clase de la particion, esto lo haremos iterativamente hasta que la nueva particion generada sea igual a la anterior. Para armar luego el autómata sera leer las tablas de la última partición generada y unir como lo indica el algoritmo.

1.5. Cadena pertenece al lenguaje:

Lo implementamos lo más directo posible. Esto es, nos paramos en el estado incial y en el primer caracter de la cadena, buscamos la transición en nuestro autómata para ese estado y ese caracter y, una vez encontrado, pasamos al siguiente estado. Repetiremos este paso hasta completar la longitud de la cadena. Una vez finalizada la misma, miramos si el estado en el que estamos parado es un estado final, en caso de serlo la cadena pertence al lenguaje y caso contrario no.
Hay que hacer dos salvedades para esto. La primera es, ¿Qué pasa si la cadena es vacía? Bueno como solución a esto tendremos que chequear la cadena previamente y, en caso de ser vacía, si nuestro estado inicial es también final entonces la cadena será válida, la cadena no pertenecerá al lenguaje en caso contrario. Lo segundo es, ¿Qué pasa si dado un estado y un caracter no encontramos una transición que satisfaga en nuestro conjunto de transiciones? Bueno, esto sólo va a pasar cuando el autómata que estamos recorriendo no es completo, y para solucionarlo bastará con recorrer todas las transiciones y que no se satisfaga la condición, en cuyo caso retornaremos que la cadena no pertenece al lenguaje.

1.6. Interseccion de dos Autómatas:

El algoritmo que usamos fue, unir los estados de unos de los automatas con los del otro y agregar las transiciones que pasen en ambas.
Una explicación de lo anteriormente dicho es, como tenemos dos automatas (A1 y A2), tomamos cada tupla (e1,e2) donde e1 pertenece a A1 y e2 pertenece a A2, y las asignaremos como estado para el autómata intersección. Luego miramos, si tenemos en A1 una transición (e1,a,e3) y en A2 la transicion (e2,a,e4), entonces vamos a agregar la transición en el autómata intersección de la siguiente manera ((e1,e2),a,(e3,e4)), asi lo hacemos para todas las transiciones que cumplan con esta condición mencionada.
Para el estado inicial del nuevo autómata tomamos la tupla en donde se encuentren el estado inicial de A1 y el estado incial de A2. Para los estados finales hacemos un caso idéntico al explicado para el estado inicial, solo que en este caso pueden haber más de una tupla que sean estados finales.
Para finalizar y que quede como un autómata comun y no tenga estados con nombre en forma de tuplas, pasamos a renombrar los estados.
Cabe mencionar que este algoritmo genera muchos estados sobrantes, o sea estados que no son alcanzados desde el estado inicial. Para solucionar esto pasamos a determinizar el nuevo autómata, con lo cual borramos dichos estados y obtenemos el verdadero autómata intersección.

1.7. Complemento de un autómata:

Para esta función tuvimos un problema, ya que para hallar el complemento de un autómata necesitamos, para un estado y un alfabeto, saber su transición aunque ésta no exista. Para solucionar esto agregamos un estado *"trampa"*, y nuestro algoritmo se encargará de generar las transiciones ausentes de nuestro autómata y llevarlas a este nuevo estado. Como sabemos este estado *"trampa"* debe cumplir con su definición, no debe ser un estado final y tampoco se puede salir de él por medio de alguna transición.

1.8. Equivalencia de autómatas

Para poder obtener la equivalencia pensamos diferentes estrategias:

- La primera fue la de ir recorriendo todos los caminos de los autómatas y ver si pasaban por las transiciones en las cuales ambos tenían el mismo caracter. La descartamos porque era engorrosa y difícil de implementar.
- La segunda idea que surgió fue la usar el complemento de alguno de los dos autómatas e intersecarlo con el otro, de esta manera, si ambos eran equivalente, el resultado será vacío. Con esto parecía que teníamos resuelto el ejercicio, pero notamos casos bordes donde no se satisfacía muy bien este método. Por ejemplo: ¿Qué pasa si uno de los dos autómatas (A1) era inicialmente vacío y el otro no (A2), y además, tomamos el complemento del último mencionado (A2)? Bueno, al intersecar ambos automatas el resultado sería vacío por definición misma de los conjuntos, y concluiríamos que son equivalentes, lo cual es un error.
- La tercera y última opción fue la siguiente, llamemos a uno de los dos autómatas A1 y al otro A2. Si "tomamos el complemento de A2 y lo intersecamos con A1"(A3), y "tomamos el complemento de A1 y lo intersecamos con A2"(A4), entonces si tanto A3 y A4 dieron vacío podremos afirmar que son equivalentes. Se puede ver fácilmente que este caso cubre al mencionado en la segundo idea, puesto que si miramos la parte agregada recientemente nos comprobará que no son equivalente. Esto es, tomamos al que inicialmente era vacío (A1) su complemento, esto nos da un automata que acepta todas las cadenas, y lo intersecamos con uno no vacío (A2), el resultado es exactamente el autómata A2. Entonces, dado esta nueva propuesta, podemos concluir que no son equivalente, lo cual efectivamente es verdad. Además, para esta tercera opción, pudimos encontrar una demostración que indica que nuestra propuesta marca formalmente una equivalencia entre ambos autómatas.