CC3006: Tarea #2

Para entregar el Lunes, Febrero 15, 2010

Bidkar Pojoy

Carlos E. López Camey

Problema 1

Considere la gramática G

G:

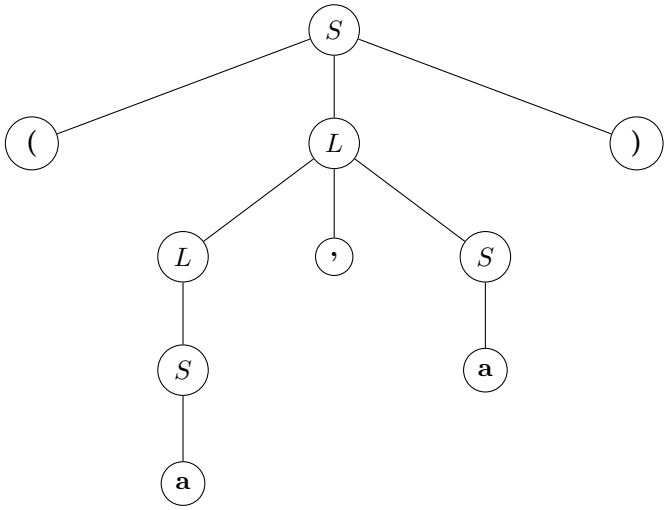
$S -> (L) \mid \mathbf{a}$
$L -> L\,\mathbf{,}\,S \mid S$

Cuáles son los terminales, los no terminales y el símbolo inicial?

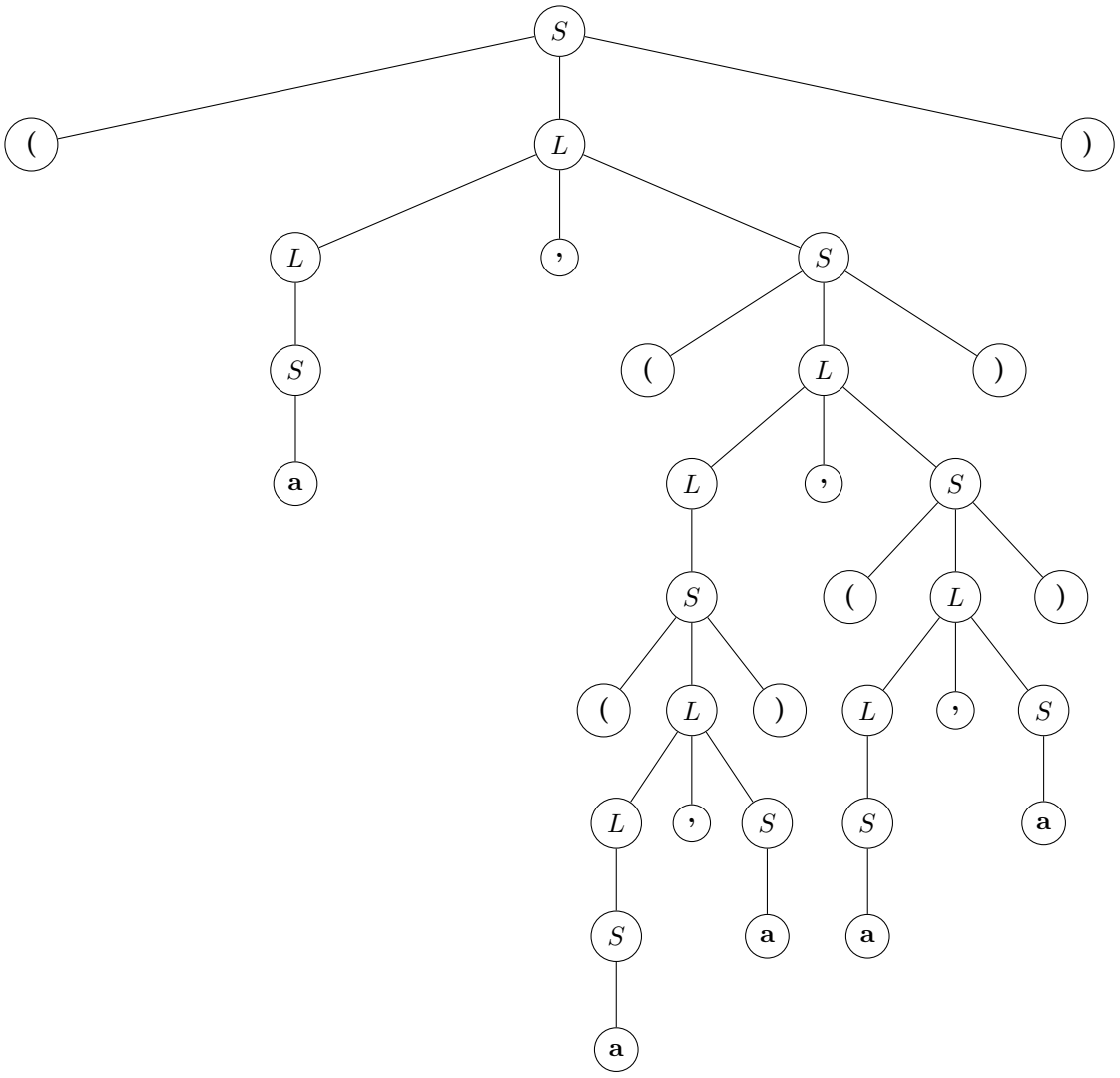
Conjunto de terminales $T = \{$'(', ',' , ')'$\}$
Conjunto de terminales $NT = \{L, S\}$
Símbolo inicial = S

Encuéntrense árboles de análisis sintáctico para las siguientes frases:

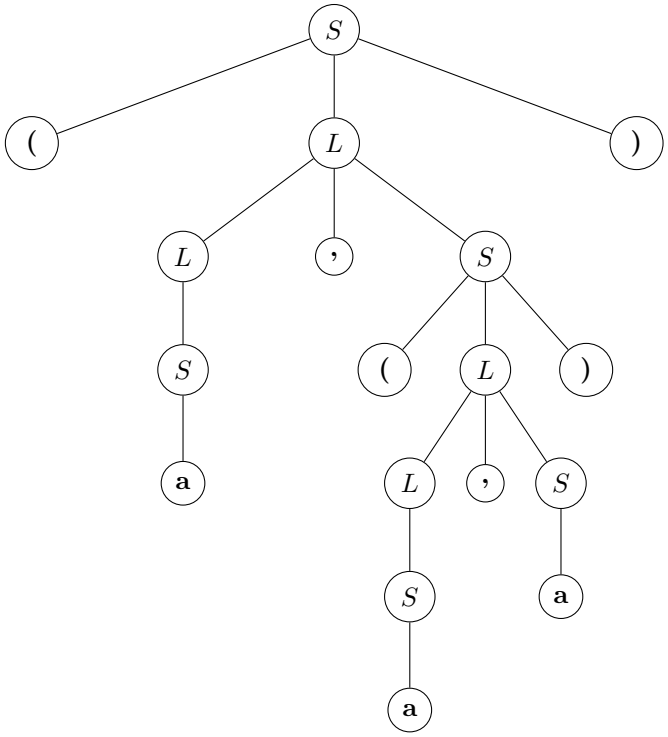
- (a,a)



- (a,((a,a),(a,a)))



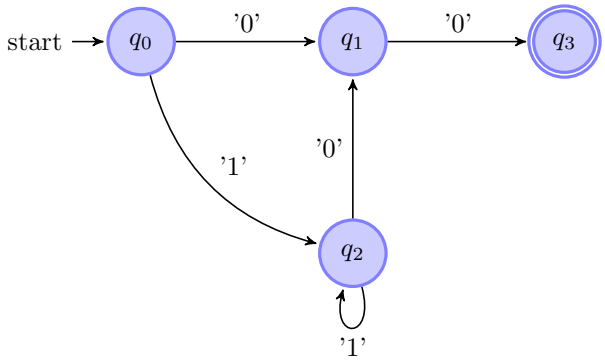
- (a,(a,a))



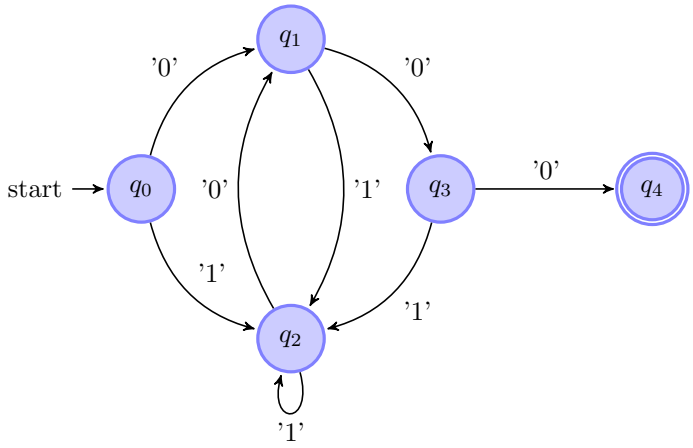
Problema 2

Construya un DFA que acepte los siguientes lenguajes sobre el alfabeto $\{0, 1\}$

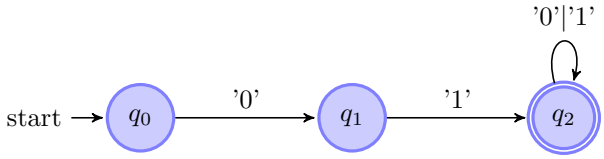
- Conjunto de cadenas que terminen en 00



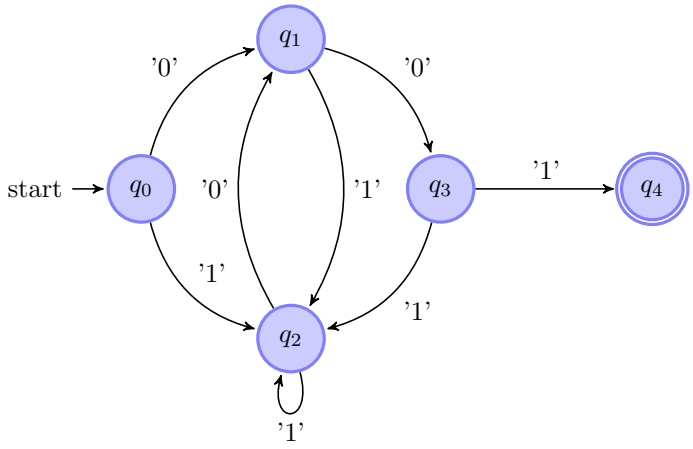
- Conjunto de cadenas con tres 0 consecutivos (no necesariamente al final)



- Conjunto de cadenas que empiecen con 01



- Conjunto de cadenas que contengan a 001 como sub-cadena



Problema 3

Escriba expresiones regulares para los lenguajes del ejercicio anterior

- Conjunto de cadenas que terminen en 00
  Respuesta:

$$(0|1)^*00$$

- Conjunto de cadenas con tres 0 consecutivos (no necesariamente al final)
  Respuesta:

$$(0|1)^*000(0|1)^*$$

- Conjunto de cadenas que empiecen con 01
  Respuesta:
  $$01(0|1)^*$$

- Conjunto de cadenas que contengan a 001 como sub-cadena
  Respuesta:
  $$(0|1)^*001(0|1)^*$$

Problema 4

De una descripción del lenguaje que generan las siguientes expresiones regulares

- $L = (0^*1^*)^*000(0|1)^*$

  Digamos que L está conformado por tres partes de manera que $L = L_1L_2L_3$ i.e. L_1, L_2 y L_3 son sub-cadenas de L las cuales se concatenan entre sí obligatoriamente en el orden establecido.

  Observemos que L_2 es en sí es, tres ceros 'obligatorios', es decir, que todas las cadenas $w \in L$ tendrán a tres ceros (L_2) como sub-cadena, en alguna parte entre el inicio y el final.

  Para definir la sub-cadena que representa L_1 al principio de cada cadena en L, observemos que la expresión 0^* genera a $\epsilon|0|00|000\ldots$ y 1^* denota al lenguaje cuyas cadenas son $\epsilon|1|11|111\ldots$. Notemos entonces que $L_1 = (0^*1^*)^*$ denota al lenguaje con las cadenas resultantes de concatenar alguna cadena generada por 0^* seguida por una generada por 1^* cero o más veces.

  Las sub-cadenas que representa L_3 al final de L son el conjunto de concatenar ninguna o más veces un 0 o un 1, e.g. $\epsilon|0|1|00|01|10|11|\ldots$, es decir, todos los números binarios posibles y ϵ.

  Ejemplos de cadenas $\in L$

  - 0011001100000110101 en donde $L_1 = 00110011$, $L_2 = 000$ como siempre y $L_3 = 00110101$
  - 0000101010111 en donde $L_1 = \epsilon$, $L_2 = 000$ como siempre y $L_3 = 0101010111$

- $(0|10)^*1^*$

  Análogamente, pensando en L como una concatenación de dos lenguajes L_1 y L_2, tenemos $L = L_1L_2$ en ese orden específico.

  L_1 contiene las sub-cadenas de L al principio que resultan de concatenar cero o más veces un 0 con un 10 o viceversa, es decir, un 10 con un 0. Por ejemplo: 010 ó 100. Esta generación de cadenas sería $\epsilon|0|10|010|100\ldots$.

  L_2 contiene las subcadenas de L al final, que es el resultado de concatenar 1 cero o más veces i.e. $\epsilon|1|11|111|1111$.

  Ejemplos de cadenas $\in L$

  - 101010101111 en donde $L_1 = 10101010$ y $L_2 = 1111$
  - 000010 en donde $L_1 = 000010$ y $L_2 = \epsilon$

Problema 5

Dadas dos expresiones regulares r_1 y r_2. Cómo demostraría (o refutaría) que $L(r_1) = L(r_2)$

Dado que las expresiones regulares se definen recursivamente, sabiendo que r_1 y r_2 son expresiones regulares entonces tenemos:

- $L(r_1|r_2) = L(r_1) \cup L(r_2)$
- $L(r_1r_2) = L(r_1)L(r_2)$
- $L(r_1^*) = (L(r_1))^*$
- $L((r_1)) = L(r_1)$

La respuesta sería hacer la serie de pasos recursivos hasta que lleven al lenguaje en una expresión denotada con las expresiones regulares anteriormente mencionadas.

Tomemos como ejemplo la siguiente demostración dadas dos expresiones regulares.

Dados $L = \{$ todas las cadenas sin dos 0 consecutivos$\}$, $r_1 = (1|01)^*(0|\epsilon)$ y $r_2 = (1^*011^*)(0|\epsilon)|1^*(0|\epsilon)$. Demostremos que $L(r_1) = L(r_2) = L$

Veamos que

$$L(r_1) = L((1|01)^*(0|\epsilon))$$

$$L((1|01)^*)L(0|\epsilon)$$

$$L((1|01))^*L(0) \cup L(\epsilon)$$

$$L(L(1) \cup L(01))^*(\{0\} \cup \{\epsilon\})$$

$$(\{1\} \cup \{01\})^*(\{0, \epsilon\})$$

$$\{1, 01, 11, 011, 101, 0101 \dots\}(\{0, \epsilon\})$$

$$\{\epsilon, 0, 10, 010, 110, 0110, 1010, 01010, \dots, 1, 01, 11, 011, 101, 0101\}$$

efectivamente vemos que $L(r_1) = L$

Ahora,

$$L(r_2) = L((1^*011^*)(\epsilon|0))|1^*(\epsilon|0))$$

$$L(L((1^*011^*))L(\epsilon|0)) \cup L(1^*(\epsilon|0))$$

$$L(L(1^*)L(011^*)(L(0) \cup L(\epsilon)) \cup L(L(1^*)(L(\epsilon) \cup L(0))))$$

$$\{1\}^*(\{011\}^*)(\{\epsilon, 0\}) \cup \{1\}^*(\{\epsilon, 0\})$$

Para mayor legibilidad del lector, notemos que 1^* actúa como lenguaje operando al principio en los dos conjuntos que serán unidos, entonces escribimos

$$\{1\}^*[(\{011\}^*)(\{\epsilon, 0\})\cup) \cup \{\epsilon, 0\}]$$

Observemos ahora que el segundo conjunto a ser unido, $\{\epsilon, 0\}$, ya está contenido en el primero. Entonces,

$$\{1\}^*(\{011\}^*\{\epsilon, 0\})$$

$$\{\epsilon, 1, 11, 111, 111\ldots\}^*(\{\epsilon, 0, 011, 0110, 011011, 0110110\ldots\})$$

$$\{\epsilon, 0, 011, 0110, 011011, 0110110, \ldots, 10, 1011, 10110, 1011011, 10110110\ldots\}$$

el cual también es el mismo conjunto que L y $L(r_1)$, implicando que $L(r_1) = L(r_2) = L$

Problema 6

Sea $w = xy$ donde x y y son cadenas sobre un conjunto de símbolos dados. Demuestre que $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, a), y)$, donde $\hat{\delta}$ es la función de transición extendida definida para un AFD.

Por inducción tenemos como caso base: Si $y = \epsilon$, entonces

$$\hat{\delta}(q, x) = \hat{\delta}(\hat{\delta}(q, x), \epsilon)$$

Que cumple para la definición de $\hat{\delta}$

Paso inductivo: Asumimos que la proposición se cumple para cadenas más pequeñas y partimos $y = wa$, en donde a es el último símbolo de y.

Partamos entonces de $\hat{\delta}(\hat{\delta}(q, x), y)$, que por la separación de y se convierte en

$$\hat{\delta}(\hat{\delta}(q, x), wa)$$

Por la definición de $\hat{\delta}$, sabemos que eso es lo mismo que

$$\delta(\hat{\delta}(\hat{\delta}(q, x), w), a)$$

Y por nuestra hipótesis inductiva,

$$\delta(\hat{\delta}(q, xw), a)$$

De nuevo, por la definición de $\hat{\delta}$ tenemos

$$\hat{\delta}(q, xwa)$$

Y por la separación que hicimos de y en wa, tenemos

$$\hat{\delta}(q, xy)$$

□

Problema 7

Demuestre que para cualquier estado q, cadena x y símbolo de entrada a tenemos que $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$, donde δ es la función de transición y $\hat{\delta}$ es la función de transición extendida para un AFD.

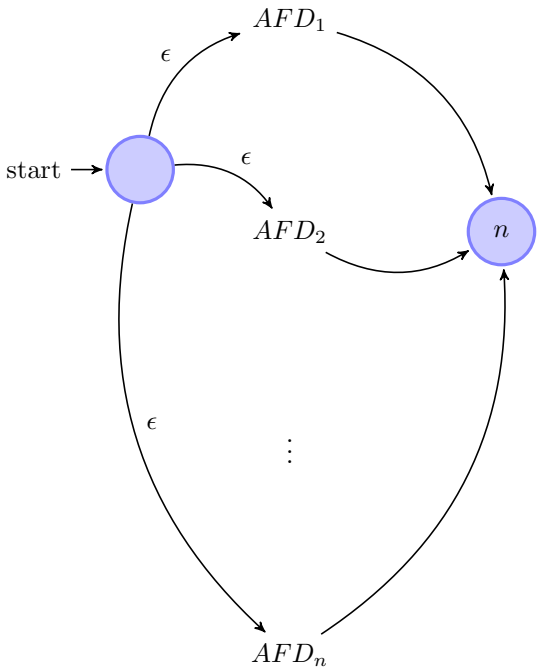
Problema 8

Dada una especificación de componentes léxicos, explique: Cómo a partir de dicha especificación se puede generar un analizador léxico que reconozca dicha especificación i.e. que reconozca cada uno de los componentes léxicos?

Tenemos una especifiifación de componentes léxicos en forma de expresiones regulares $r_1, r_2 \dots r_n$. Usando el algoritmo de Thompson generaría un Automata No-Determinista $\epsilon - AFN_1, \epsilon - AFN_2 \dots \epsilon - AFN_3$con transiciones ϵ para cada expresión regular.

Luego usando el algoritmo de cerradura-ϵ convertiría cada $\epsilon - AFN_i$ en una serie de Automatas No-Deterministas $AFN_1, AFN_2 \dots AFN_n$. Luego con el algoritmo de construccipn de subconjuntos convertiría cada AFN_i en un Automata Finito Determinista.

En este punto, tenemos n Automatas Finitos Deterministas, uno para cada expresion regular. Por último, compondría UN SOLO Automata No-Determinista con transiciones ϵ, eso es:



Teniendo este Automata grande, aplicaría de nuevo cerradura-ϵ y construcción de subconjuntos para llegar a un solo Automata Finito Determinista A.

Con A podemos rechazar o aceptar a cualquier cadena en la especificacion de componentes léxicos.

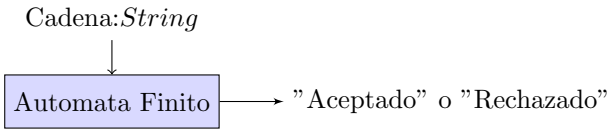
Investigue: Complejidad Computacional y Análisis de Algoritmos

Qué es complejidad computacional?

Complejidad computacional es una rama de la teoría de la computación que se enfoca en clasificar problemas computacionales acorde a su dificultad. En este contexto, se entiende que un problema está sujeto a ser resuelto por una computadora.

Automata Finito

Un automata finito toma una cadena de entrada y devuelve una respuesta de la forma "aceptado" o "rechazado", eso es:



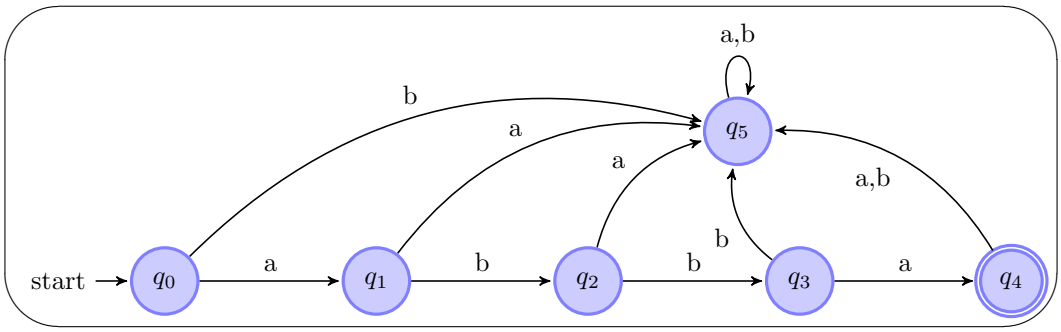
La definición formal de un Automa Finito M es $M = (Q, \sum, \delta, q_0 F)$ en donde:

- Q: conjunto de estados
- $\sum$: alfabeto de entrada
- δ: función de transición
- q_0: estado de transición
- F: conjunto de estados de aceptación

Lenguaje aceptados por Automatas Finitos

Definimos a $L(M)$ como el lenguaje que contiene todas las cadenas aceptadas por un automata finito M
Ejemplo: $L(M) = \{abba\}$

M



En donde

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
- $\sum = \{a, b\}$
- δ: función de transición, en donde aparecen $\delta(q_0, a) = q_1, \delta(q_0, b) = q_5 \ldots$
- q_0 es el estado inicial
- $F = \{q_4\}$

Lenguajes Regulares

Decimos que un lenguaje L es regular si existe un automata finito M tal que $L = L(M)$, eso es, que el automata finito pueda generar a L.

Observación: Todos los lenguajes aceptados por Automatas Finitos, forman la familia de los Lenguajes Regulares.

Ejemplos de lenguajes regulares:

- { abba }
- { ϵ,ab,abba }
- { todas las cadenas que empiezan con ab }

Gramática independiente del contexto

Decimos que G es una gramática independiente si es de la forma $G = (V, T, S, P)$ en donde:

1. T es un conjunto de **terminales**
2. V es un conjunto de **no terminales** (variables sinácticas)
3. P es el conjunto de producciones de la forma $A- > x$
4. S es una definición de un **no terminal** como símbolo inicial de G.

Ejemplo, G:
$L- > L$ '+' d
$L- > L$ '-' d
$L- > d$
$L- >' 0'||'1'||'2'...'9'$

En donde el símbolo inicial es el **no terminal** L.

$L- > L$ '+' d (Regla 1)
$L- > d$ '+' d (Regla 3)
$L- > 5$ '+' d (Regla 4)
$L- > 5$ '+' 1 (Regla 4)

Y decimos que $w \in L(G)$

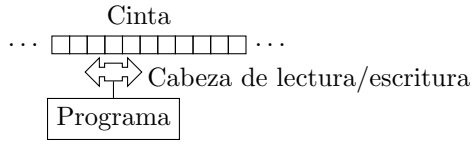
Lenguajes libres de contexto

Decimos que un lenguaje L es libre de contexto sí y solo si existe una gramática G con $L = L(G)$, eso es $L(G) = \{w|w$ es derivada a partir de $G\}$, que es el conjunto de todas las producciones posibles a partir de G

Máquina de Turing

Una máquina de Turing es un modelo matemático de una máquina computacional. Es un dispositivo que manipula símbolos contenidos en una cinta y se cree que si un problema puede ser resuelto por un algoritmo, entonces existe una máquina de Turing que puede resolver el problema. En la teoría computacional es el modelo de computadora más utilizado. Nos enfocaremos en este.

Conceptualmente, una máquina de Turing, como un Automata Finito, consiste de un control finito y una cinta. A cualquier hora está en uno de los estados finitos. La cinta se extiende infinitamente hacia la derecha y también está dividido en cuadrados en los cuales puede ser escrito un símbolo. Aunque, a diferencia de los automatas finitos, su cabeza puede leer y escribir y se puede mover hacia la izquierda, derecha o quedarse en el mismo cuadrado después de haber leído o escrito.



Dada una cadenea de símbolos en una cinta, una máquina de Turing empieza en el estado inicial. En cualquier estado, la cabeza lee el símbolo bajo ella y puede ya sea borrarlo o remplazarlo con otro símbolo (posiblemente el mismo). Después mueve la cabeza hacia la izquierda o derecha o no se mueve y procede al siguiente estado, que puede ser el mismo que el actual. Uno de sus estados es el estado en donde la computación para y cuando la máquina de Turing para.

Formalmente, una máquina de Turing es una 5-tupla $M = (Q, \sum, \Gamma, q_0, \delta)$ en donde:

- Q es un conjunto finito de estados, en donde se asume que ninguno contiene al símbolo h. El símbolo h se usa para denotar al estado de interrupción.
- $\sum$ es un conjunto finito de símbolos y el alfabeto de entrada.
- Γ es un conjunto finito de símbolos conteniendo $\sum$ como subconjunto y es el conjunto de símbolos en la cinta.
- q_0 es el estado inicial
- δ es la función de transición pero su valor puede no estar definido para ciertos puntos, es un mapeo desde $Q \times (\Gamma \cup \{V\})$ a $(Q \cup \{h\}) \times (\Gamma \cup \{\Delta\}) \times \{R, L, S\}$.

  Aquí Δ denota el vacío y R, L y S denotan una movida de la cabeza hacia la derecha, izquierda o la misma (sin moverla) respectivamente.

Aceptancia en una máquina de Turing

Una máquina de Turing M acepta a la entrada si la máquina se para o se interrumpe en un estado final.

La misma máquina de Turing rechaza la entrada si la máquina para o se interrumpe en un estado no-final, o si la máquina entra en un ciclo infinito.

Definición de un algoritmo

Un algoritmo para la función $f(w)$ es una máquina de Turing que computa $f(w)$. Esto es en efecto, la tésis de Alan Turing, los algoritmos son máquinas de Turing. Cuando existe 'existe un algoritmo' nos referimos a 'existe una máquina de Turing que ejecuta el algoritmo'.

Máquina de Turing determinista

En una máquina de Turing determinista, que es uno de los modelos más usados para definir clases de complejidad, no está permitido tener una función de transición que para un mismo estado dado y un símbolo, solo tiene un movimiento de respuesta i.e. derecha, izquierda o quedarse en el mismo cuadro.

Existen otros modelos de máquina de Turing, como la máquina de turing no-determinista, la máquina de Turing cuántica, la máquina de Turing simétrica, máquina de Turing probabilistica o una máquina de

Turing alternante. Todas ellas son igualmente capaces en principio de computar un algoritmo pero cuando los recursos (como el tiempo o el espacio) están limitados, algunas de estas pueden ser más poderosas que otras.

Lenguajes aceptados por máquinas de Turing

Una máquina de Turing acepta a los lenguajes independientes del contexto y a los lenguajes regulares, que están contenidos en los lenguajes independientes del contexto. También aceptan unos lenguajes que no definimos aquí que son de la forma $a^n b^n c^n$ y ww.

Medidas de complejidad

Para una definición más precisa de que significa resolver un problema dado una cantidad de recursos (tiempo o espacio), un modelo computacional como una máquina de Turing puede ser usada. El *tiempo* requerido por una máquina de Turing determinista M para una entrada x es el número total de transiciones de estado, o el número de pasos que esta hace antes de interrumpirse y dar salida a la respuesta ("si" o "no").

Se dice que una máquina de Turing M opera en un tiempo de $f(n)$ si el tiempo requerido por M para cada entrada de longitud n es a lo sumo $f(n)$. Un problema de decisión A puede ser resuelto en tiempo $f(n)$ si existe una máquina de Turing que opera en tiempo $f(n)$ que resuelve el problema.

La teoría de complejidad computacional se enfoca en separar y clasificar problemas basados en su dificultad, se definen conjuntos de problemas basados en cierto criterio. Por ejemplo, el conjunto de los problemas resolvibles en un tiempo $f(n)$ en una máquina de Turing determinista es denotado DTIME($f(n)$) y es una clase de complejidad.

El recurso DTIME(n)

Usando la notación $O(k)$, usaremos una máquina de Turing con multicintas y contaremos el número de pasos hasta que una cadena sea aceptada.

Ejemplo: $L = \{a^n b^n : n \geq 0\}$ Algoritmo que acepta una cadena w:

- Usamos una máquina de Turing de dos cintas
- Copiamos a en la segunda cinta
- Comparamos a y b

El tiempo necesitado para computar la salida de w es:

- Copiamos a en la segunda cinta $\implies O(|w|)$
- Comparamos a y b $\implies O(|w|)$

Tiempo total necesitado: $O(|w|)$

Eso es, para una cadena de longitud n, el tiempo necesitado para ser aceptado es de $O(n)$

Decimos entonces que la clase de complejidad del lenguaje es DTIME(n) dado que una máquina de Turing determinista acepta cada cadena de longitud n en tiempo $O(n)$

De una manera similar, definimos la clase DTIME($T(n)$) para cualquier función de tiempo $T(n)$, por ejemplo: DTIME(n^2), DTIME(n^3), ...

Teorema: DTIME(n^{k+1})contiene a DTIME(n^k)

Decimos que los algoritmos con tiempo polinomial son DTIME(n^k) y para un k pequeno podemos computar el resultado rápido

La clase P

La clase P está definida como

$$P = \cup \text{DTIME}(n^k) \text{ para todo } k$$

- Tiene tiempo polinomial
- Contiene todos los problemas con tiempo polinomial.

Otras clases

Existen otras clases de complejidad, pero las más importantes son generalmente definidas por limitar el tiempo o el espacio usado por el algoritmo.

Análisis de algoritmos

El analizar un algoritmo significa determinar la cantidad de recursos (como tiempo y espacio) necesarios para ejecutarlo. Es una parte de la amplia teoría de la complejidad computacional, explicada anteriormente.

En el análisis de algoritmos, es común estimar su complejidad en un **sentido asintótico** i.e. estimar la función de complejidad para una entrada grande. Aquí notaciones como la Big O (usada para definir P en el inciso anterior), Ω y Θ son usadas para este fin.

Por ejemplo, un algoritmo de búsqueda binaria corre en un número de pasos proporcionales al logaritmo de el tamano de la lista en donde se está buscando i.e. $O(\log(n))$, dicho comunmente "en tiempo logaritmico"

Notación Big-O

Algunas veces en análisis de el tiempo de corrida de los algoritmos, se puede ser muy dejado y esto podría llevar a un nivel de inexactitud en el resultado. Pero también el riesgo opuesto está presente: es posible ser *demasiado* preciso. Un análisis profundo se basa en las simplificaciones correctas.

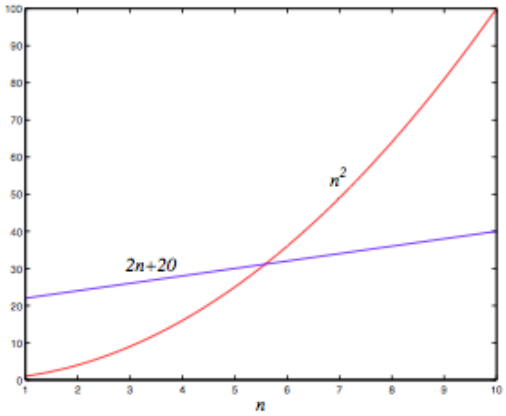
El expresar el tiempo en términos de *pasos básicos de computadora* es ya una simplificación. Después de todo, el tiempo tomado para cada una de esos pasos depende en el procesador y hasta en los detalles de como hacer *cache* y como resultado, el tiempo de corrida podría diferir entre una ejecución de un algoritmo con otra.

Es necesario entonces, expresar la eficiencia de un algoritmo en términos que sean independientes de la máquina en donde está corriendo. Y esto lleva a otra simplificación, en vez de decir que un algoritmo toma $8n^3 + 2n + 10$ pasos para una entrada de tamano n, es mucho más simple creer que los términos más bajos como $2n$ y 10 (que serán insignificantes mientras n crezca), y hasta el detalle del coeficiente 8 en el primer término, decimos que el algoritmo corre en tiempo $O(n^3)$.

Es en efecto el tiempo que define esta notación precisamente. Pensemos en $f(n)$ y $g(n)$ como tiempos de corrida de dos algoritmos con la misma entrada de tamano n.

Sean $f(n)$ y $g(n)$ funciones que van de los enteros positivos hacia los reales positivos. Decimos que $f = O(g)$, que significa que f no crece más rápido que g, si hay una constante $c > 0$ que cumpla con $f(n) \leq c \cdot g(n)$

Entonces, tenemos una herramienta para elegir digamos entre dos algoritmos que corren uno en $f_1(n) = n^2$ pasos y el otro $f_2(n) = 2n + 20$). Si graficamos las dos notemos que



para $n \leq 5$, f_1 es mas pequeno. En este caso, f_2 se escala mucho mejor mientras n crece y por eso es superior. Esto es capturado por la notación big-O: $f_2 = O(f_1)$, ya que:

$$\frac{f_2(n)}{f_1(n)} = \frac{2n + 20}{n^2} \leq 22$$

para todo n; de otro modo, vemos que $f_1 \neq O(f_2)$, ya que $f_1(n)/f_2(n) = n^2/(2n + 20)$ puede ser arbitrariamente largo y por lo tanto no hay ninguna constante c tal que cumpla la definición.

Investigue: Generadores de Analizadores Léxicos

Un generador de analizadores léxico, hace lo que su nombre lo dice, generar analizadores léxicos. El trabajo de un analizador léxico es tomar palabra por palabra a partir de un programa y dice si todas esas palabras están contenidas en el lenguaje y hacen *match* con algún patrón de un componente léxico, eso es, si pertenecen al lenguaje L generado o denotado por una expresión regular r.

Un generador de analizadores léxicos GAL toma básicamente como entrada una especificación de componentes léxicos.

Haciendo lo explicado en el **Problema 8**, GAL toma una serie de especificaciones de componentes léxicos y los representa con un Automata Finito Determinista.

Una lista de ejemplos de generadores léxicos:

- Lex
- Flex
- CoCoR

Bibliografía

- *Petros Drineas, Models of Computation, Computer Science, Rensselaer Polytechnic Institute, http://www.cs.rpi.edu/ dri*
- *S. Dasgupta, C.H. Papadimitriou y U.V. Vazirani, Algorithms, Julo 18 del 2006*