Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 7

Gramática Regular a partir de un DFA

Factor de ponderación: 9

1. Objetivos

El objetivo de la práctica es consolidar los conocimientos adquiridos sobre Gramáticas [2] y sobre Autómatas Finitos [1] al mismo tiempo que se continúan desarrollando capacidades para diseñar y desarrollar programas orientados a objetos en C++. El programa leerá desde un fichero la especificación de un autómata finito determinista y a partir de dicha definición generará automáticamente una gramática regular que genere el lenguaje reconocido por el autómata.

Podemos entonces concretar que los objetivos principales de esta práctica son los siguientes:

- Consolidar los conocimientos adquiridos sobre Gramáticas.
- Consolidar los conocimientos adquiridos sobre Autómatas Finitos.
- Estudiar y practicar el algoritmo de obtención de una Gramática Regular a partir del DFA que reconoce el lenguaje.
- Refinar la implementación de la clase que representa Autómatas Finitos e implementar una clase para representar Gramáticas.
- Profundizar en las capacidades de diseñar y desarrollar programas orientados a objetos en C++.

2. Gramáticas independientes del contexto

Desde la antigüedad, los lingüistas han descrito las gramáticas de los idiomas en términos de su estructura de bloques, y han descrito cómo las oraciones se construyen recursivamente a partir de frases más pequeñas y, finalmente, de palabras o elementos de palabras individuales. Una propiedad esencial de estas estructuras de bloques es que las unidades lógicas nunca se solapan. Por ejemplo, la frase en inglés:

John, whose blue car was in the garage, walked to the grocery store

usando corchetes como meta-símbolos puede agruparse como:

[John [, [whose [blue car]] [was [in [the garage]]],]] [walked [to [the [grocery store]]]]

Una gramática independiente del contexto (*Context Free Grammar, CFG*) [3] proporciona un mecanismo simple y matemáticamente preciso para describir los métodos por los cuales las cadenas de algún lenguaje formal se construyen a partir de bloques más pequeños, capturando la "estructura de bloques" de las frases de manera natural. Su simplicidad hace que este formalismo sea adecuado para un estudio matemático riguroso. En cualquier caso, las características de la sintaxis de los lenguajes naturales no se pueden modelar mediante gramáticas independientes del contexto.

Este formalismo y también su clasificación como un tipo especial de gramática formal fue desarrollado a mediados de los años 50 del siglo pasado por Noam Chomsky [4]. Basándose en esta notación de gramáticas basada en reglas, Chomsky agrupó los lenguajes formales en una serie de cuatro subconjuntos anidados conocidos como la *Clasificación de Chomsky* [5]. Esta clasificación fue y sigue siendo fundamental para la teoría de lenguajes formales, especialmente para la teoría de los lenguajes de programación y el desarrollo de compiladores.

La estructura de bloques fue introducida en los lenguajes de programación por el proyecto Algol (1957-1960), el cual, como consecuencia, también incluía una gramática independiente del contexto para describir la sintaxis del lenguaje. Esto se convirtió en una característica estándar de los lenguajes de programación, aunque no solo de éstos. En [6], a modo de ejemplo, se utiliza una gramática independiente del contexto para especificar la sintaxis de JavaScript.

Las gramáticas independientes del contexto son lo suficientemente simples como para permitir la construcción de algoritmos de análisis eficientes que, para una cadena dada, determinan si - y cómo - se puede generar esa cadena a partir de la gramática. El *algoritmo de Cocke-Younger-Kasami* es un ejemplo de ello, mientras que los ampliamente usados analizadores LR y LL son algoritmos más simples que tratan sólo con subconjuntos más restrictivos de gramáticas independientes del contexto.

Formalmente una *gramática independiente del contexto* G (por simplicidad nos referiremos a "una gramática") viene definida por una tupla $G \equiv (\Sigma,\ V,\ S,\ P)$ cada uno de cuyos componentes se explican a continuación:

- Σ: Conjunto de símbolos terminales (o alfabeto de la gramática), $\Sigma \cap V = \emptyset$
- V: Conjunto de símbolos no terminales ($V \neq \emptyset$)
- S: Símbolo de arranque (***S**tart*) o axioma de la gramática ($S \in V$)
- P: Conjunto de reglas de producción:

  $P = \{A \to \alpha \mid A \in V,\ \alpha \in (V \cup \Sigma)^*\}$

  $P \subset V \times (V \cup \Sigma)^*,\ (P \neq \emptyset)$

Habitualmente, para especificar una gramática, se especifican todas sus reglas de producción (P), y se sigue un convenio de notación que permite determinar todos los elementos. El símbolo de arranque es aquel cuyas producciones aparecen en primer lugar en la lista de producciones. Por ejemplo, las siguientes producciones:

$S \to U \mid W$

$U \to TaU \mid TaT$

$W \to TbW \mid TbT$

$T \to aTbT \mid bTaT \mid \epsilon$

definen una gramática independiente del contexto para el lenguaje de cadenas de letras a y b en las que hay un número diferente de unas que de otras. Como se puede apreciar, a partir de las producciones anteriores podremos deducir la especificación completa de la gramática:

- $\Sigma = \{a,\ b\}$
- $V = \{S,\ U,\ W,\ T\}$
- El símbolo de arranque es S
- Las reglas de producción son las de la relación anterior

3. Gramáticas regulares

Analizando la forma que pueden tener las producciones de una gramática, se dice que una gramática $G \equiv (\Sigma, V, P, S)$ es lineal por la derecha si todas sus producciones tienen la forma:

$$A \to uB \mid v$$

Alternativamente, se dice que una gramática es lineal por la izquierda si todas las producciones son de la forma:

$$A \to Bu \mid v$$

En estas expresiones $A, B \in V$ son símbolos no terminales, mientras que $u, v \in \Sigma^*$ son secuencias de símbolos terminales.

Una gramática es regular si es lineal por la izquierda o lineal por la derecha. Nótese que la restricción consiste en que las partes derechas de las reglas de producción han de contener un único símbolo no terminal y éste debe ser el primero (lineal por la izquierda) o el último símbolo (lineal por la derecha) de la regla. Teniendo esto en cuenta, y analizando la anterior definición de gramáticas independientes del contexto, podemos decir que **las gramáticas regulares son un subconjunto de las gramáticas independientes del contexto**.

Se presentan a continuación algunos ejemplos de gramáticas regulares:

- $G_1 \equiv (\Sigma, V, P, S)$ definida por: $\Sigma = \{a, b\}$, $V = \{S, A\}$ y las producciones:

  $S \to bA$
  $A \to aaA \mid b \mid \epsilon$

  Es una gramática regular (lineal por la derecha) que genera el lenguaje $L(G_1) = L(\texttt{b(aa)*b?})$, formado por las cadenas que comienzan con b, seguidas de un número par de aes y que pueden acabar también en b.

- $G_2 \equiv (\Sigma, V, P, S)$ definida por: $\Sigma = \{0, 1\}$, $V = \{S, A\}$ y las producciones:

  $S \to 0A$
  $A \to 10A \mid \epsilon$

  Es una gramática regular (lineal por la derecha) que genera el lenguaje siguiente: $L(G_2) = L(\texttt{0(10)*})$

- Una gramática regular, lineal por la izquierda que genera el mismo lenguaje es:

  $G_3 \equiv (\Sigma, V, P, S)$ definida por: $\Sigma = \{0, 1\}$ $V = \{S\}$ y las producciones:

  $S \rightarrow S10 \mid 0$

  L(G_3)= L(G_2)

4. Ejercicio práctico

En esta práctica se desarrollará un programa `dfa2grammar` que lea un fichero de texto `.fa` en el que figure la especificación de un DFA M y genere un fichero `.gra` que contenga la especificación de una gramática regular G tal que $L(M) = L(G)$. Para obtener la gramática G, tal que $L(G) = L(M)$, aplicaremos el algoritmo visto en clase y disponible en las transparencias del tema 3 de la asignatura [2].

El comportamiento del programa al ejecutarse en línea de comandos debiera ser:

```
$ ./dfa2grammar
Modo de empleo: ./dfa2grammar input.fa output.gra
Pruebe ‘dfa2grammar --help’ para más información.
```

Donde `input.fa` es el fichero de entrada que contiene la especificación de un DFA M y `output.gra` es un fichero de salida que contiene la especificación de una gramática regular G tal que $L(M) = L(G)$. La opción `--help` en línea de comandos ha de producir que se imprima en pantalla un breve texto explicativo del funcionamiento del programa.

Los ficheros `.fa` contendrán la especificación de un DFA y deberán tener el siguiente formato:

- Línea 1: Símbolos del alfabeto separados por espacios.
- Línea 2: Número total de estados del autómata.
- Línea 3: Estado de arranque del autómata.
- A continuación figurará una línea para cada uno de los estados. Cada línea contendrá los siguientes números, separados entre sí por espacios en blanco:
  - Número identificador del estado. Los estados del autómata se representarán mediante números naturales. La numeración de los estados corresponderá a los primeros números comenzando en 0.
  - Un 1 si se trata de un estado de aceptación y un 0 en caso contrario.
  - Número de transiciones que posee el estado.

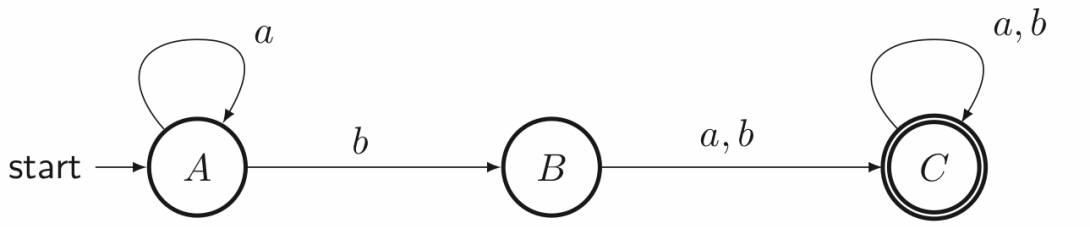

Figura 1: DFA que reconoce el lenguaje representado por la expresión `a*b(a|b)+`

- A continuación, para cada una de las transiciones, y utilizando espacios en blanco como separadores, se detallará la información siguiente:
  - Símbolo de entrada necesario para que se produzca la transición. Para representar la cadena vacía (el no consumir símbolo de la entrada) se utilizará el carácter `&`
  - Estado destino de la transición.

Cabe destacar que el formato de estos archivos es el mismo que el utilizado en la práctica anterior para la simulación de autómatas finitos.

Tal y como se ha visto en clase, dado un lenguaje regular siempre es posible hallar una gramática regular que lo genere a partir del DFA que reconoce ese lenguaje. Se explica a continuación cómo proceder para hallar esa gramática [2].

Sea L un lenguaje regular. Sea M el DFA que reconoce L, $L = L(M)$ con $M \equiv (\Sigma, Q, s, F, \delta)$. La gramática regular $G \equiv (\Sigma', N, S, P)$ que genera L se construye a partir del DFA M tomando:

- $V = Q$
- $\Sigma' = \Sigma$
- $S = s$
- $P = \{q \to ap \mid \delta(q, a) = p\} \cup \{q \to \varepsilon \mid q \in F\}$

Se observa que cada una de las transiciones del DFA da lugar a una regla de producción en la gramática. Los estados de aceptación del autómata introducen producciones adicionales en la gramática. Si se aplica este método al DFA que se muestra en la Figura 1 se obtiene la gramática siguiente:

$A \to aA \mid bB$

$B \to aC \mid bC$

$C \to aC \mid bC \mid \epsilon$

La gramática obtenida a partir del procedimiento anterior deberá ser modelada a través de una clase específica. Tras finalizar la creación de la gramática, ésta deberá ser volcada a un fichero de especificación de gramáticas. Los ficheros de especificación de gramáticas son ficheros de texto plano con extensión `.gra`. Estos ficheros contienen los elementos definitorios de la Gramática $G \equiv (\Sigma, V, S, P)$ en este orden: símbolos terminales, símbolos no terminales, símbolo de arranque y producciones. A continuación se muestra el fichero de especificación de la gramática G_2:

```
2
0
1
2
S
A
S
3
S -> 0A
A -> 10A
A -> &
```

El formato de cada uno de estos elementos en el fichero es el siguiente:

1. **Símbolos terminales (alfabeto)**: una línea que contiene N, el número de símbolos en el alfabeto seguida de N líneas, cada una de las cuales contiene un símbolo del alfabeto. Cada símbolo del alfabeto debe ser un único caracter imprimible.

2. **Conjunto de símbolos no terminales**: una línea que contiene V, el número de símbolos no terminales, seguida de V líneas, cada una de las cuales contiene una cadena alfanumérica sin espacios.

3. **Símbolo de arranque**: una única línea que contiene el símbolo de arranque, S, de la gramática. Ha de ser uno de los símbolos no terminales relacionados anteriormente.

4. **Producciones**: una línea que contiene P, el número de producciones de la gramática, seguida por P líneas cada una de las cuales contiene una producción en el formato:

   `A ->` α

   siendo $\alpha \in (\Sigma \cup V)^*$, es decir una secuencia de símbolos terminales y no terminales. La cadena vacía, ϵ se representa mediante el carácter

5. Detalles de implementación

Tenga en cuenta las siguientes consideraciones:

- No traslade a su programa la notación que se utiliza en este documento, ni en la teoría de Autómatas Finitos. Por ejemplo, el cardinal del alfabeto de su autómata no debiera almacenarse en una variable cuyo identificador sea `N`. Al menos por dos razones: porque no sigue lo especificado en la Guía de Estilo respecto a la elección de identificadores y más importante aún, porque no es significativo. No utilice identificadores de un único carácter, salvo para situaciones muy concretas.
- Favorezca el uso de las clases de la STL, particularmente `std::array`, `std::vector` o `std::string` frente al uso de estructuras dinámicas de memoria gestionadas a través de punteros.
- Construya su programa de forma incremental y depure las diferentes funcionalidades que vaya introduciendo.
- En el programa parece ineludible la necesidad de desarrollar una clase `Automata` y una clase `Grammar`. Estudie las componentes que definen a un autómata y a una gramática y vea cómo trasladar esas componentes a sus clases.
- Valore análogamente qué otras clases se identifican en el marco del problema que se considera en este ejercicio. Estudie esta referencia [9] para practicar la identificación de clases y objetos en su programa.
- La clase `Automata` ha de contemplar un constructor que, tomando como parámetro un nombre de fichero "construya" el autómata a partir de la especificación del fichero.
- Dicha clase también contemplará un método *ConvertToGrammar()* que devuelva un objeto de la clase `Grammar` siempre y cuando el autómata de partida sea un DFA.
- La clase `Grammar` deberá definir un método que permita imprimir la especificación de la gramática en un fichero.

6. Criterios de evaluación

Se señalan a continuación los aspectos más relevantes (la lista no es exhaustiva) que se tendrán en cuenta a la hora de evaluar esta práctica:

- Se valorará que el alumnado haya realizado, con anterioridad a la sesión de prácticas, y de forma efectiva, todas las tareas propuestas en este guión. Esto implicará que el programa compile y ejecute correctamente. Además, el comportamiento del programa deberá ajustarse a lo solicitado en este documento.
- También se valorará que, con anterioridad a la sesión de prácticas, el alumnado haya revisado los documentos que se enlazan desde este guión.
- El programa desarrollado deberá compilarse utilizando la herramienta make y un fichero Makefile, separando los ficheros de declaración (`*.h`) y definición (`*.cc`) de clases.
- Se requiere que todos los ficheros con códido fuente se alojen en un único directorio junto con el fichero Makefile.
- El código desarrollado ha de estar escrito de acuerdo al estándar de la guía de estilo de Google para C++ [12].
- El comportamiento del programa debe ajustarse a lo solicitado en este documento.
- Ha de acreditarse capacidad para introducir cambios en el programa desarrollado.
- El programa ha de ser fiel al paradigma de programación orientada a objetos. Se valorará que el alumnado haya identificado clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se plantea. Se valorará la coherencia e idoneidad del diseño realizado en las clases diseñadas para representar gramáticas y autómatas finitos.
- Se valorará que el programa se haya escrito de modo que las diferentes funcionalidades que se precisen hayan sido encapsuladas en métodos concretos cuya extensión textual se mantuviera acotada.
- Se requiere que todos los atributos de las clases definidas en el proyecto tengan un comentario descriptivo de la finalidad del atributo en cuestión. Además, se requiere que los comentarios del código fuente sigan el formato especificado por Doxygen [7]. Utilice asimismo esta [8] referencia para mejorar la calidad de la documentación de su código fuente.

Si el alumnado tiene dudas respecto a cualquiera de estos aspectos, debiera acudir al foro de discusiones de la asignatura para plantearlas allí. Se espera que, a través de ese foro, el alumnado intercambie experiencias y conocimientos, ayudándose mutuamente a resolver dichas dudas. También el profesorado de la asignatura intervendrá en las discusiones que pudieran suscitarse, si fuera necesario.

Referencias

[1] Transparencias del Tema 2 de la asignatura: Autómatas finitos y lenguajes regulares, https://campusingenieriaytecnologia2223.ull.es/mod/resource/view.php?id=5919

[2] Transparencias del Tema 3 de la asignatura: Lenguajes y Gramáticas Independientes del Contexto, https://campusingenieriaytecnologia2223.ull.es/mod/resource/view.php?id=5932

[3] Context Free Grammar, https://en.wikipedia.org/wiki/Context-free_grammar

[4] Noam Chomsky https://en.wikipedia.org/wiki/Noam_Chomsky

[5] Chomsky hierarchy https://en.wikipedia.org/wiki/Chomsky_hierarchy

[6] JavaScript 1.4 Grammar https://www-archive.mozilla.org/js/language/grammar14.html

[7] Doxygen http://www.doxygen.nl/index.html

[8] Diez consejos para mejorar tus comentarios de código fuente https://www.genbeta.com/desarrollo/diez-consejos-para-mejorar-tus-comentarios-de-codigo-fuente

[9] Cómo identificar clases y objetos http://www.comscigate.com/uml/DeitelUML/Deitel01/Deitel02/ch03.htm

[10] Makefile Tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor

[11] C++ Makefile Tutorial: https://makefiletutorial.com

[12] Google C++ Style Guide, https://google.github.io/styleguide/cppguide.html