Universidad de La Laguna

ESIT – INFORMÁTICA
Departamento de Ingeniería Informática y de Sistemas

Computabilidad y Algoritmia
Curso 2019-2020

PRACTICA 9: Gramáticas Regulares y Autómatas Finitos. Gramática regular a partir de un DFA

9.1. Objetivos

- 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.
- Implementar en C++ una clase para representar Gramáticas.
- Profundizar en las capacidades de diseñar y desarrollar programas orientados a objetos en C++.

9.2. Rúbrica de evaluación del ejercicio

Se señalan a continuación los aspectos más relevantes (la lista no es exhaustiva) que el profesorado tendrá en cuenta a la hora de evaluar el trabajo que el alumnado presentará en la sesión de evaluación de la práctica:

- El alumnado ha de acreditar conocimientos para trabajar con la shell de GNU/Linux en su VM.
- El alumnado ha de acreditar capacidad para editar ficheros remotos en la VM de la asignatura usando VSC.
- El código ha de estar escrito de acuerdo al estándar de la guía de Estilo de Google para C++
- El programa desarrollado deberá compilarse utilizando la herramienta make y un fichero Makefile, separando los ficheros de declaración (`*.h`) y definición (`*.cc`) de clases.

- El comportamiento del programa debe ajustarse a lo solicitado en este documento.
- Ha de acreditarse capacidad para introducir cambios en el programa desarrollado.
- Modularidad: el programa ha de escribirse de modo que las diferentes funcionalidades que se precisen sean encapsuladas en funciones y/o métodos cuya extensión textual se mantenga acotada.
- El programa ha de ser fiel al paradigma de programación orientada a objetos.
- Se valorará la coherencia e idoneidad del diseño realizado en las clases diseñadas para representar gramáticas y autómatas finitos.
- Se requiere que, en la sesión de evaluación de la misma, todos los ficheros con códido fuente se alojen en un único directorio junto con el fichero Makefile de compilación.
- Se requiere que todos los atributos de las clases definidas en el proyecto tengan un comentario descriptivo de la finalidad del atributo en cuestión.
- Se requiere que los comentarios del código fuente sigan el formato especificado por Doxygen [1].
- Utilice asimismo esta [7] 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.

9.3. Introducción

Desde la antiguedad, 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 metasímbolos 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* [2] 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. Las características de la sintaxis de los lenguajes naturales no se pueden modelar mediante CFGs.

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 [3], 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 $G \equiv (V, \Sigma, S, P)$ cada uno de cuyos componentes se explican a continuación:

- V: Conjunto de símbolos no terminales ($V \neq \emptyset$)
- Σ: Conjunto de símbolos terminales (o alfabeto de la gramática), $\Sigma \cap V = \emptyset$
- 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.

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. En el ejemplo:

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

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, 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.

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 $L(G_1) = L(\texttt{b(aa)*b?})$: 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 $L(G_2) = L(\texttt{0(10)*})$.

- 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)

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.

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

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

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 9.1.

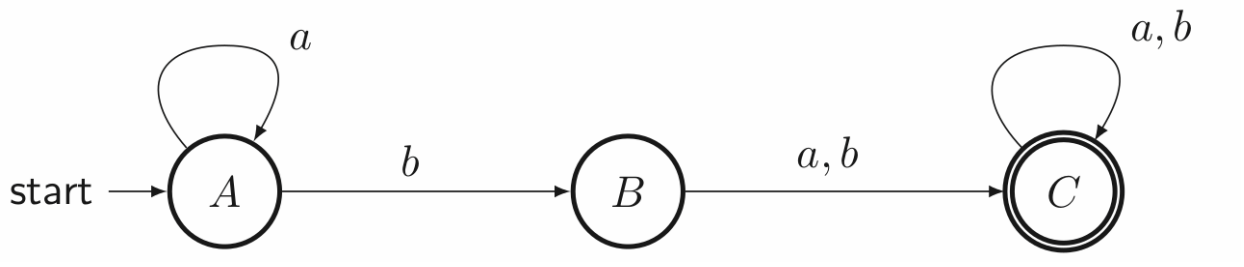

Figura 9.1: DFA que reconoce el lenguaje representado por la expresión `a*b(a|b)+`

se obtiene la gramática:

$A \rightarrow aA|bB$
$B \rightarrow aC|bC$
$C \rightarrow aC|bC|\epsilon$

En esta referencia [6] se encuentra otro ejemplo de aplicación del método expuesto.

9.4. Ejercicio práctico

Desarrollar un programa `dfa2g.cc` que lea un fichero de texto en el que figure la especificación de un DFA y genere un fichero `.gra` que contenga la especificación de una gramática regular tal que $L(M) = L(G)$.

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

```
$ ./dfa2g
Modo de empleo: ./dfa2g input.dfa output.gra
Pruebe 'dfa2g --help' para más información.
```

Donde `input.dfa` es el fichero conteniendo la especificación de un DFA y `output.gra` es un fichero que contiene la especificación de una gramática regular. 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 de especificación de gramáticas son ficheros de texto plano con extensión `.gra` El Listado 9.1 muestra el fichero de especificación de la gramática G_2. 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. 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 `~` (código ASCII 126).

Todas las líneas de un fichero `.gra` que comiencen con los caracteres `//` corresponden a comentarios, y deben ser ignorados por el programa a la hora de procesar el fichero.

```
1  // Universidad de La Laguna
2  // Grado en Ingenieria Informatica
3  // Computabilidad y Algoritmia
4  // Fichero de representacion de la Gramatica:
5  //
6  //       S -> 0A
7  //       A -> 10A | epsilon
8  //
9  // Lenguaje generado: 0(10)*
10 //
11 ////////////////////////////////////
12 2
13 0
14 1
15 2
16 S
17 A
18 S
19 3
20 S -> 0A
21 A -> 10A
22 A -> ~
```

Listado 9.1: Fichero de especificación de la gramática G_2

Los ficheros de especificación de DFAs definen en este orden el alfabeto, los estados, el estado inicial, los estados finales y las transiciones. El formato de cada uno de estos elementos en el fichero es el siguiente:

1. **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 caracter imprimible.

2. **Estados**: una línea que contiene M, el número de estados, seguida de M líneas, cada una de las cuales contiene el identificador (etiqueta) de un estado. Cada identificador de estado debe ser una cadena alfanumérica sin espacios.

3. **Estado inicial**: una línea que contiene el identificador del estado inicial. El estado inicial ha de ser uno de los estados relacionados anteriormente.

4. **Estados de aceptación**: una línea que contiene F, el número de estados finales, seguido por F líneas, cada una de las cuales contiene el identificador de un estado final. Cada estado final ha de ser uno de los relacionados anteriormente.

5. **Transiciones**: una línea que contiene T, el número de transiciones en la función δ, seguida de T líneas, cada una de las cuales contiene tres cadenas q_i, sim, q_f separadas por espacios. Cada una de estas líneas indica una transición del estado q_i al q_f con símbolo *sim*, es decir, $\delta(q_i, sim) = q_f$. q_i y q_f han de estar listados en la lista de estados y sim debe estar en la lista de símbolos del alfabeto.

Todas las líneas de un fichero `.dfa` que comiencen con los caracteres `//` corresponden a comentarios, y deben ser ignorados por el programa a la hora de procesar el fichero.

9.5. 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.
- Valore análogamente qué otras clases identifica Ud. en el marco del problema que se considera en este ejercicio.
- El programa `dfa2g.cc` ha de contener al menos las clases `Grammar` y `Dfa` para representar gramáticas y DFAs respectivamente.
- La clase `DFA` ha de contemplar un constructor que, tomando como parámetro un nombre de fichero "construya" el DFA a partir de la especificación del fichero.
- También contemplará un método *ConvertToGrammar()* que devuelva un objeto de la clase `Grammar`.
- La clase `Grammar` deberá definir un método que permita imprimir la especificación de la gramática en un fichero.

9.6. Formateo automático del código

El código fuente escrito por un programador debiera ser conforme a las reglas de estilo de la organización para la que trabaja. Ello facilita la legibilidad de los programas así como el trabajo en equipo: cualquier programador de la organización puede entender fácilmente el código escrito por otros desarrolladores puesto que todos siguen unas reglas comunes a la hora de escribir sus programas.

En el caso particular de esta asignatura, las reglas de estilo que se ha propuesto seguir son las de la Guía de Estilo de Código de Google para C++ [17]. Estas reglas de

estilo no son únicas. Algunas otras relevantes para el caso de C++ son [8, 9, 10, 11, 12, 13]. y cada organización que no define las suyas propias suele adoptar alguna de éstas.

Hay fundamentalmente dos formas de garantizar que el código sigue uno de estos estándares. La primera es adoptar la costumbre de escribir el código siempre atendiendo a las recomendaciones de estilo de la guía elegida. Esta es posiblemente la más adecuada porque dota a la programadora de buenos hábitos a la hora de escribir su código y porque, salvo cuestiones opinables, las diferencias entre unas guías de estilo u otras no atañen a los aspectos más relevantes. Se podría establecer un paralelismo con lo que ocurre al escribir en nuestro idioma natural: se puede utilizar un corrector ortográfico pero es muy conveniente ser capaz de escribir sin faltas de ortografía. La otra opción es utilizar una herramienta que analice el código y lo reescriba de acuerdo a unas reglas de estilo. Los *linters* [14] son herramientas que cumplen con este cometido. `clang-tidy` [15] o `cpplint` [16] son dos linters adecuados para usar con C++ y su función va más allá de corregir los errores de estilo porque pueden ser útiles para detectar y corregir errores comunes a la hora de programar.

La extensión para código C++ [21] de Microsoft Visual Studio Code (VSC) suministra soporte para dar formato al códifo fuente [22] usando el formato *clang* [18] incluído con la extensión. Una vez configurado el estilo que se desea para los proyectos, las combinaciones de teclas `Ctrl+Shift+I` y `Ctrl+K Ctrl+F` permiten formatear un fichero completo o parte del mismo respectivamente. Si en el espacio de trabajo se encuentra un fichero con nombre `.clang-format` el formato se aplica de acuerdo a las especificaciones de ese fichero. En este tutorial [23] se explica cómo utilizar la herramienta `clang-format` [25] para crear un fichero `.clang-format` que siga alguno de los estilos soportados por la herramienta, uno de los cuales (*style=google*) es el que se usa en la asignatura. En este artículo [24] se explica el significado de algunos de los ajustes definidos por el estilo.

Bibliografía

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

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

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

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

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

[6] Converting DFA to Regular Grammar http://www.jflap.org/modules/ConvertedFiles/DFA%20to%20Regular%20Grammar%20Conversion%20Module.pdf

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

[8] GeoSoft's C++ Programming Style Guidelines https://geosoft.no/development/cppstyle.html

[9] LLVM coding standards https://llvm.org/docs/CodingStandards.html

[10] Chromium's style guide https://www.chromium.org/developers/coding-style

[11] Mozilla's style guide https://firefox-source-docs.mozilla.org/code-quality/coding-style/index.html

[12] WebKit's style guide https://webkit.org/code-style-guidelines/

[13] Microsoft's style guide https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/code-style-rule-options?view=vs-2017

[14] Linter https://en.wikipedia.org/wiki/Lint_(software)

[15] https://clang.llvm.org/extra/clang-tidy/

[16] https://github.com/google/styleguide/tree/gh-pages/cpplint

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

[18] Clang Format https://clang.llvm.org/docs/ClangFormat.html

[19] Format C/C++ Code Using Clang-Format https://leimao.github.io/blog/Clang-Format-Quick-Tutorial/

[20] How to customize C++'s coding style in VSCode https://medium.com/@zamhuang/vscode-how-to-customize-c-s-coding-style-in-vscode-ad16d87e93bf

[21] extensión para código C++ https://code.visualstudio.com/docs/languages/cpp

[22] soporte para dar formato al códifo fuente https://code.visualstudio.com/docs/cpp/cpp-ide#_code-formatting

[23] Clang-Format Quick Tutorial https://leimao.github.io/blog/Clang-Format-Quick-Tutorial/

[24] How to customize coding style in VSC https://medium.com/@zamhuang/vscode-how-to-customize-c-s-coding-style-in-vscode-ad16d87e93bf

[25] clang-format https://clang.llvm.org/docs/ClangFormatStyleOptions.html