ETS de Ingenieros Industriales y de Telecomunicación.
Universidad de Cantabria

cuieet 2010

Teoría de Autómatas y Lenguajes Formales: Una aproximación práctica desde dos paradigmas de programación

Milagros Fernández Gavilanes, Carlos Gómez Rodríguez, Jesús Vilares Ferro, Jorge Graña Gil, Antonio Blanco Ferro

Facultad de Informática, Universidade da Coruña, Campus de Elviña, 15071 A Coruña
Tel: 981 167 000, Fax: 981 167 160, Email: mfgavilanes@udc.es

Resumen
Debido a su carácter marcadamente algebraico, la materia de *Teoría de Autómatas y Lenguajes Formales* puede ser percibida como excesivamente teórica, con el consiguiente riesgo de rechazo por parte del alumno. Compartimos aquí nuestra experiencia dentro de esta asignatura, a la que hemos logrado dar un carácter mucho más práctico mediante una serie de ejercicios de programación empleando unas librerías de código, proporcionadas por el docente, que implementan las estructuras algebraicas básicas de la materia. Dichas prácticas y las librerías de código empleadas en su implementación, se abordan aquí.

Palabras Clave: Teoría de Autómatas y Lenguajes Formales; recursos para la docencia; librerías de código; ejercicios de laboratorio.

Abstract
Due to its strongly algebraic aspects, the Automata Theory and Formal Languages subject can be perceived as too theoretical, with the risk of rejection by the student. We share here our experience in this subject, to which we have managed to give a more practical character through a series of exercises using programming code libraries provided by the lecturer, which implement the basic algebraic structures of the matter. Such exercises, as well as the code libraries used for their implementation, are addressed here.

Keywords: Automata Theory and Formal Languages; teaching resources; programming code libraries; lab exercises.

1. Introducción

Independientemente del nombre concreto que reciba en las diversas titulaciones de Informática, la asignatura de *Teoría de Autómatas y Lenguajes Formales (TALF)* [1-3] supone para el alumno su primera aproximación al estudio formal de los fundamentos teóricos que subyacen en la ingeniería interna de los lenguajes de programación y en los propios fundamentos de la computación. No sólo será capaz de usar mejor los lenguajes de programación disponibles, sino que además será capaz de construir y

adecuar un lenguaje de programación en atención a los requerimientos. Asimismo, el alumno adquirirá un mayor conocimiento de los tipos de problemas que pueden ser resueltos con los lenguajes de programación. Dada la relevancia de sus contenidos, esta asignatura ha sido ampliamente implementada en los planes de estudios de las diversas facultades y escuelas de informática españolas con esquemas de contenidos bastante similares de unas universidades a otras.

En nuestro caso se trata de una asignatura troncal de tercer curso de las titulaciones de *Ingeniería Informática (II)* e *Ingeniería Técnica en Informática de Sistemas (ITIS)*, de 9 créditos (6 teóricos y 3 prácticos) y con las siguientes competencias [4,5], que se traducirán, a su vez, en una serie de contenidos concretos:

- Conocer en profundidad la estructura y función de los sistemas de descripción y reconocimiento de lenguajes formales.
- Estudiar los conceptos, modelos y técnicas relacionados con estas cuestiones. Conocer las estructuras de datos y los algoritmos usados para implementar los modelos de reconocimiento de lenguajes formales, así como sus posibles dominios de aplicación práctica.
- Realizar implementaciones de estos modelos en algún dominio.
- Sintetizar los conceptos estudiados en ideas concretas que permitan comprender mejor los fundamentos de la computación.
- Considerar la integración de las técnicas y estructuras estudiadas aquí en otros dominios de aplicación.

El resto de este trabajo analiza nuestra visión práctica de la materia y los recursos empleados. Primero, la Sección 2 introduce el programa práctico de la asignatura[1]. A continuación, la Sección 3 describe las prácticas-tipo planteadas en ella, así como las librerías de código proporcionadas para su realización. Finalmente, la Sección 4 recoge nuestras reflexiones finales.

1 El programa teórico, ampliamente descrito y disponible en [4,5], no será objeto de análisis en este artículo debido a la falta de espacio y por no ser, además, el objetivo de este trabajo, eso sí, ambos diseñados en base a las directrices del *Computing Curricula* [1,2].

2. Programa práctico

Como sabemos, esta asignatura tiene un marcado carácter algebraico. Esto hace que a la hora de planificar sus contenidos prácticos surja una disyuntiva entre dos posibles aproximaciones: dedicar las clases prácticas a plantear y resolver en la pizarra ejercicios que permitan ilustrar los conceptos de carácter algebraico expuestos en teoría[2], o bien programar los algoritmos más destacados de entre aquéllos vistos en clase. En nuestro caso hemos optado por ésta última, por varias razones:

- Siempre se puede reservar parte de las clases teóricas para ejercicios.
- Dotar a la materia de un carácter más práctico permitirá evitar que ésta sea percibida como excesivamente teórica, con el consiguiente riesgo de rechazo por parte del alumno, que suele rehuir tales asignaturas.
- Al implementar los algoritmos relacionados con esta materia surgirán ejercicios de programación muy interesantes que permitirán al alumno no sólo poner en práctica y reafirmar los conocimientos sobre esta asignatura, sino también sus conocimientos sobre programación.
- La segunda opción es más flexible que la primera, ya que permite diseñar un abanico más amplio de estilos de prácticas, a la vez que más acordes con la aplicación real de los conceptos aprendidos.

Una vez decidido que las prácticas consistirán en ejercicios de programación, debemos decidir cómo orientarlas y qué lenguaje usar. En este sentido, existen paquetes y librerías disponibles para la construcción y uso de autómatas y traductores de estado finito, tales como: **FSA** (*Finite-State Automata*) y **UTR** (*Utilities for Transducers*) [8], ambas escritas en C++, y usadas en tareas de análisis morfológico o corrección ortográfica; y **FSM** (*Finite-State Machines*) [9], un conjunto de herramientas *software* sobre Unix. No obstante, todas ellas están orientadas hacia usos industriales, no didácticos. En este sentido, existen las herramientas **Grail+** [10] y **JFLAP** (*Java Formal Languages and Automata Package*) [11]. **Grail+** es un entorno de computación simbólica implementado en C++ para máquinas de estado finito, expresiones regulares

y lenguajes finitos, con múltiples funcionalidades, pero que dejó de mantenerse en 2002. Por contra, **JFLAP** sigue mantenida en la actualidad. Implementada en Java, esta herramienta de simulación permite diseñar y analizar el comportamiento de autómatas finitos, de pila y máquinas de Turing [12]. Desafortunadamente, dichas herramientas no nos permiten un acceso suficientemente completo o documentado a su código, por lo que no se adecúan a nuestras necesidades, debiendo también desecharlas.

Sin embargo, implementar desde cero las estructuras y funciones necesarias para programar los algoritmos propuestos en las prácticas podría resultar excesivo para el alumno. Por otro lado, la falta de una homogeneización a la hora de su programación supondría además un obstáculo tanto a la hora del seguimiento del trabajo del alumno como a la hora de su corrección. Es por ello que hemos optado por desarrollar unas librerías de código diseñadas específicamente de cara a las prácticas de la asignatura, las cuales describiremos en la siguiente Sección.

Para su implementación, nos hemos decantado por lenguajes diferentes: Java[3], del paradigma orientado a objetos, y OCaml[4], del paradigma funcional. En el marco actual de transformación de los planes de estudios a Grado esta disponibilidad de la misma librería en dos lenguajes diferentes dota de mayor aplicabilidad a nuestra propuesta, permitiendo elegir entre uno u otro caso dependiendo de la orientación del plan de estudios de cada titulación.

3. Prácticas-tipo y librerías

Como respuesta a nuestras necesidades, hemos desarrollado una serie de prácticas-tipo cubriendo el temario práctico de la asignatura. En ellas el alumno no está obligado a implementar todos los algoritmos planteados, pudiendo elegir entre los distintos apartados propuestos y así realizar él mismo la configuración final de la práctica, lo que hace su trabajo más ameno y repercute positivamente en su motivación. No obstante,

2 Este era el caso de la asignatura en el plan de estudios de la antigua Licenciatura.
3 http://www.sun.com/java/
4 http://caml.inria.fr

el grado de libertad no es total, ya que se definen una serie de pautas para que se establezca un encadenamiento lógico entre los ejercicios y, sobre todo, para que se profundice en aquellos apartados que no le hayan quedado suficientemente claros en teoría. Los contenidos prácticos cubiertos responden a las estructuras y algoritmos básicos empleados directamente en las aplicaciones prácticas más comunes basadas en esta tecnología, tales como la búsqueda de correspondencias en los propios comandos de la consola (ej. `dir *.txt`) y herramientas como `grep`, o los analizadores sintácticos de los lenguajes de programación.

Asimismo, como se ha comentado, se proporcionarán sendas librerías de código diseñadas específicamente para estas prácticas[5]: $TALF_{OCaml}$ (en OCaml) y $TALF_{Java}$ (en Java). Éstas proporcionan tipos de datos correspondientes a los principales formalismos algebraicos vistos en teoría, así como las funciones básicas para su generación y manejo, e interfaces para su representación gráfica y almacenamiento. Se ha procurado siempre que las implementaciones sean lo más acordes posibles con los formalismos originales. El empleo de estas librerías reporta diversas ventajas:

- Sirven como apoyo a las clases teóricas, permitiendo ilustrar el funcionamiento de las estructuras algebraicas implicadas.
- Responden a las necesidades que impone nuestro estilo de prácticas. El código fuente y el esquema de integración de los módulos está documentado, lo que hace viable seleccionar diferentes partes, retirarlas del sistema, y proponer al alumno que introduzca su propia implementación, permitiendo que los demás módulos sigan funcionando y verificar el correcto funcionamiento de la nueva solución.

No obstante, las librerías siguen en fase de desarrollo y, curso tras curso, crecen mediante la incorporación de nuevas funcionalidades.

3.1. Operaciones con conjuntos

Esta práctica inicial propone implementar, mediante listas, el tipo de dato *conjunto* y sus operaciones asociadas. Su objetivo es reforzar algunos de los conceptos

matemáticos básicos que el alumno repasa en el primer tema y retomar el contacto con el lenguaje de programación empleado. Dada su simplicidad, esta primera práctica no precisa de nuestras librerías. Sin embargo, como se verá, ambas hacen amplio uso del tipo de dato *conjunto*. En el caso de Java, el propio lenguaje proporciona diversas implementaciones del mismo[6]. En el caso de OCaml la librería proporciona una implementación propia junto con sus operaciones asociadas.

```
type 'a conjunto=Conjunto of 'a list;;
```

3.2. Expresiones regulares y autómatas finitos

El objetivo de esta segunda práctica es el de reforzar los conceptos relacionados con las expresiones regulares, los autómatas finitos, y las operaciones realizables sobre ambos. Así, se podrían implementar las operaciones de transformación entre ambos, o algunas de las operaciones, como pueden ser el *reconocimiento de cadenas* de entrada, la *eliminación de ε-transiciones*, la *determinización* o la *minimización*.

Sin embargo, antes de pasar a definir las estructuras algebraicas mencionadas, es preciso definir previamente las estructuras más básicas que permiten denotar un lenguaje: los *símbolos*. Concretamente, se distinguen dos tipos de símbolos: *terminales*, que forman parte de la cadena de los lenguajes, y *no terminales*, que sirven para definirlos. Asimismo, definimos *alfabeto*, representado como Σ, como un conjunto no vacío y finito de símbolos terminales. De este modo nuestra librería $TALF_{Java}$ define la superclase `Simbolo` y sus subclases `Terminal` y `No_terminal`. Por su parte, la librería $TALF_{OCaml}$ usa un tipo concreto de dato:

```
type simbolo = Terminal of string | No_terminal of string;;
```

Los terminales, a su vez, pueden agruparse formando *palabras* o *cadenas*, secuencias finitas de cero o más símbolos. Éstas son implementadas como listas de terminales. En el caso de $TALF_{Java}$ se proporciona la clase `Cadena`. Mientras, en $TALF_{OCaml}$, y de forma similar, las *cadenas* son tratadas como listas de símbolos: `simbolo list`. Asimismo,

5 Disponibles en `http://www.grupolys.org/docencia/talf/lib/`
6 Véase el `Interface Set` de `java.util`.

las librerías proporcionan funciones que permitan convertir una cadena de entrada en modo texto a su correspondiente representación interna.

3.2.1 Expresiones regulares

Dicho esto, podemos introducir ya el concepto formal de *expresión regular* (ER) sobre un alfabeto Σ, definida recursivamente tal que así: □ (el *conjunto vacío*), ε (*épsilon*) y cualquier *símbolo terminal a* son ER's básicas; y si *r* y *s* son ER's, entonces *r* □ *s* (*unión*), *r* □ *s* (*concatenación*), y r^* (*repetición* o *cierre*) también lo son.

Para su implementación, $TALF_{Java}$ proporciona el paquete `ER`, que incluye la clase abstracta `ER`, que engloba las subclases `VacioER` y `TerminalER`, que implementan los respectivos casos base, y las subclases `UnionER`, `ConcatenacionER` y `RepeticionER`, que implementan los respectivos casos recursivos. De forma similar, la librería $TALF_{OCaml}$ define el tipo de dato:

```
type er =  Vacio
         | Constante of simbolo
         | Union of (er * er)
         | Concatenacion of (er * er)
         | Repeticion of er;;
```

donde el `Vacio` corresponde al caso base □; el caso base `Constante of simbolo` corresponde a ε cuando el símbolo es `Terminal""` y al conjunto de ER's, cuando el símbolo es otro *terminal*; y el resto de casos corresponden a los casos recursivos.

Asimismo, las librerías proporcionan métodos para generar automáticamente la representación interna de una expresión regular a partir de su descripción en formato texto mediante una notación derivada directamente de la notación matemática empleada en la literatura, simplificando así el trabajo del alumno.

3.2.2 Autómatas finitos

Por su parte, un *autómata finito* (AF) se define formalmente como una tupla $(Q, \Sigma, q_0, \Delta, F)$ donde: Q es un conjunto de *estados*; Σ es el *alfabeto* de símbolos terminales de entrada; q_0 es el *estado inicial*; Δ es la *función de transición*, definible como un

conjunto de *arcos* o *transiciones* que constan de un estado origen, un estado destino y un símbolo terminal de entrada; por último, *F* es el conjunto de *estados finales*.

Para su implementación $TALF_{Java}$ proporciona el paquete `AF`, con las clases:

`Estado`: *estado* de un AF.

`Arco`: *arco* o *transición* de un AF. Consta de los siguientes atributos:

- `Estado qi`: su *estado origen*.
- `Estado qj`: su *estado destino*.
- `Terminal t`: *terminal de entrada* que etiqueta el arco.

`AF`: clase que implementa un AF. Contiene los siguientes atributos, derivados directamente de la definición formal:

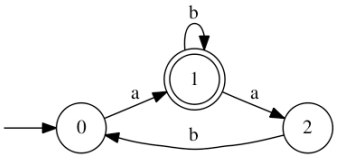
- `LinkedHashSet alfabeto`: almacena el *alfabeto* aceptado por el AF en forma de objetos de tipo `Terminal`.
- `Estado inicial`: *estado inicial* del AF.
- `LinkedHashSet arcos`: almacena el conjunto de objetos `Arco` que definen la *función de transición* del AF.
- `LinkedHashSet finales`: *estados finales* del AF, en este caso objetos `Estado`.

Por su parte, la librería $TALF_{Ocaml}$ proporciona los siguientes tipos de datos, también derivados de la definición formal:

```
type estado = Estado of string;;
type arco_af = Arco_af of (estado * estado * simbolo);;
type af = Af of (estado conjunto * simbolo conjunto *
            estado * arco_af conjunto * estado conjunto);;
```

Ambas librerías incluyen, de nuevo, mecanismos para generar la correspondiente implementación de un AF a partir de su especificación en modo texto de acuerdo a una sintaxis predefinida derivada de su definición matemática.

Asimismo, ambas librerías proporcionan igualmente funciones de salida que permiten al alumno visualizar por pantalla una representación del autómata, bien gráfica mediante un *diagrama de transiciones*, bien tabular mediante una *tabla de transiciones*, tal y como mostramos en la Figura 1.



	a	b	$
-> 0	1		
*1	2	1	
2		0	

Figura 1: *AF de ejemplo: diagrama de transiciones y tabla de transiciones*

3.3. Gramáticas independientes del contexto

Esta práctica pretende mostrar cómo funcionan dichas gramáticas, así como introducir la noción de *árbol sintáctico* como representación de la estructura que la gramática aplica sobre las cadenas del lenguaje. Queremos que el alumno sea capaz de implementar algoritmos de análisis sintáctico como el *CYK*, o comprobar si una gramática está en *Forma Normal de Chomsky*.

Una *gramática independiente del contexto* (GIC) se define como una tupla *(N, T, P, S)* donde: *N* es un conjunto de *no terminales*, *T* es el conjunto de *terminales*, *P* es el conjunto de *reglas de producción*, y *S* es el *axioma* de la gramática. Cada *regla de producción* reescribe un símbolo no terminal en una lista de símbolos que pueden ser tanto terminales como no terminales, en cualquier número y orden.

Para su implementación, $TALF_{Java}$ proporciona el paquete `Gramaticas` con las clases:

`ReglaGIC`: implementa una *regla de producción* en base a los atributos. De este modo, `ArrayList drcha` implementará la parte derecha de la regla como una lista objetos `Simbolo`; y `No_terminal izqda` hace lo mismo con su parte izquierda.

`GIC`: implementa una GIC empleando los siguientes atributos derivados de su definición formal:

- `LinkedHashSet noTerminales`: almacena el conjunto de *no terminales* en forma de objetos de tipo `No_terminal`.
- `LinkedHashSet terminales`: almacena el conjunto de *terminales* en forma de objetos de tipo `Terminal`.
- `LinkedHashSet reglas`: almacena el conjunto de *reglas de producción* en forma de objetos de tipo `ReglaGIC`.
- `No_terminal axioma`: su *axioma*.

De modo análogo, la librería $TALF_{Ocaml}$ define los siguientes tipos de datos:

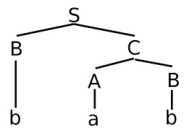
```
type regla_gic = Regla_gic of (simbolo * simbolo list);;
```

```
type gic = Gic of (simbolo conjunto * simbolo conjunto *
           regla_gic conjunto * simbolo);;
```

donde éste último está constituido –de forma paralela a la definición– por un primer conjunto de *símbolos no terminales*, un segundo conjunto de *símbolos terminales*, un conjunto de *reglas de producción* y, por último, el *axioma* de la gramática.

De nuevo las librerías proporcionan una serie de funciones que permiten generar una GIC a partir de una especificación en formato texto siguiendo una sintaxis determinada, derivada directamente de su notación formal y de la notación matemática empleada en la literatura.

Asimismo, de cara a las prácticas de análisis sintáctico, a la hora visualizar el árbol de análisis resultante, existen de nuevo dos formatos de salida, tal y como muestra la Figura 2. Por una parte una representación gráfica en forma arborescente, y por otra una representación textual mediante notación parentizada.




```
(S (B b) (C (A a) (B b)))
```

***Figura** 2: Ejemplo de árbol de análisis: representación gráfica y parentizada*

3.4. Autómatas de pila

Actualmente, esta última práctica sólo está disponible en el caso de emplear la librería $TALF_{Ocaml}$, y tiene por finalidad implementar el algoritmo de construcción de un autómata de pila a partir de la GIC que reconoce, o viceversa, así como simular su funcionamiento a la hora de intentar reconocer una cadena de entrada.

Formalmente, un autómata de pila (AP) se define como una tupla $(Q, \Sigma, \Gamma, q_0, \Delta, Z, F)$ donde: Q es un conjunto de estados; Σ es el *alfabeto de terminales de entrada*; Γ es el *alfabeto de la pila*; q_0 es el *estado inicial*; Δ es la *función de transición* entre estados, definible como un conjunto de *arcos* o *transiciones* que constan de estado origen, estado destino, símbolo terminal de entrada, símbolo de la cima de la pila, y cadena de

símbolos que reemplazan a dicho símbolo en la cima de la pila; *Z* es el *símbolo inicial de pila*; y *F* es el conjunto de *estados finales*. Para su implementación, la librería $TALF_{Ocaml}$ contempla los siguientes tipos de datos:

```
type arco_ap= Arco_ap of(estado * estado * simbolo *
              simbolo * simbolo list);;
type ap= Ap of(estado conjunto * simbolo conjunto *
         simbolo conjunto * estado * arco_ap conjunto *
         simbolo * estado conjunto);;
```

De nuevo se proporcionan las funciones pertinentes para la generación de un elemento de este tipo a partir de su especificación en modo texto, así como para su visualización mediante una representación gráfica del autómata siguiendo los convenios, tal y como muestra la Figura 3.

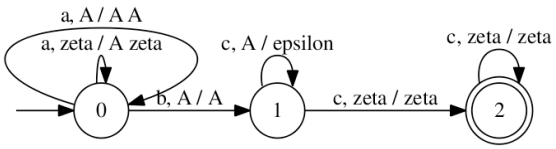


***Figura** 3: AP de ejemplo*

4. Reflexiones finales

Este trabajo presenta nuestra visión práctica de la asignatura de *Teoría de Autómatas y Lenguajes Formales*, la cual pretende evitar que dicha materia, por su carácter marcadamente algebraico, pueda ser percibida como excesivamente teórica, con el consiguiente riesgo de rechazo por parte del estudiante. Para ello proponemos una serie de prácticas-tipo que permiten darle un carácter más práctico, a la vez que reafirmar los conceptos de la materia. De este modo se plantean una serie de ejercicios de programación sobre las estructuras y algoritmos mas representativos del temario, empleando unas librerías de código proporcionadas por el docente que implementan las estructuras de la materia. Dichas prácticas son, además, configurables, lo que repercute muy positivamente en la motivación del alumno.

Consideramos que nuestro planteamiento ha resultado exitoso ya que: (1) los indicadores de rendimiento de la asignatura muestran que sus tasas de presentados y

aprobados superan ampliamente la media de la titulación, especialmente en los porcentajes de alumnos presentados al examen y aprobados respecto al total (en más de 10 puntos); (2) las encuestas de evaluación docente señalan un alto grado de satisfacción; y (3) el rendimiento académico del alumno en prácticas (que supone un 25% de la nota final) es satisfactoriamente alto.

6. Referencias

1. D. Kelley. *Teoría de autómatas y lenguajes formales*. Prentice Hall. (1995.)

2. J.E. Hopcroft, R. Motwani y J.D. Ullman. *Introducción a la teoría de autómatas, lenguajes y computación*. Addison Wesley. (2002)

3. P. Isasi, P. Martínez y D. Borrajo. *Lenguajes, gramáticas y autómatas: Un enfoque práctico*. Addison Wesley. (1997)

4. *Teoría de Autómatas y Lenguajes Formales (Ing. Informática), guía docente*. `https://campusvirtual.udc.es/guiadocente/guia_docent/index.php?centre=614&ensenyament=614111&assignatura=614111301`

5. *Teoría de Autómatas y Lenguajes Formales (Ing. Téc. en Informática de Sistemas), guía docente*. `https://campusvirtual.udc.es/guiadocente/guia_docente/index.php?centre=614&ensenyament=614311&assignatura=614311302`

6. *Computing Curricula 2001, Computer Science Final Report*, The Joint Task Force on Computing Curricula, IEEE ACM (2001).

7. Computer Science Curriculum 2008: An Interim Revision of CS 2001, Report from the Interim Review Task Force, IEEE Computer Society and Association for Computing Machinery (2008).

8. J. Daciuk, R.E. Watson y B.W. Watson. *Incremental Construction of Acyclic Finite-State Automata and Transducers.* Finite State Methods in Natural Language Processing. (1998). Disponible en: `http://www.eti.pg.gda.pl/~jandac/fsa.html`

9. M. Mohri, F.C.N. Pereira y M.D. Riley. *Weighted Finite-State Transducers in Speech Recognition. Computer Speech and Language.* (2002) Disponible en: `http://www.research.att.com/~fsmtools/fsm/`

10. T.O. Ellis, J.F. Heafner y W.L. Sibley. *The GRAIL project: An experiment in man-machine communication.* Proc. of the Society for Information Display Vol 11 No 3. (1969) `http://www.csd.uwo.ca/Research/grail/`

11. S.H. Rodger y T.W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package.* Editorial Jones & Bartlett, ISBN 0763738344 (2006). Disponible en: `http://www.jflap.org`

12. M. Mohri, F.C.N. Pereira y M.D. Riley. *Weighted Finite-State Transducers in Speech Recognition. Computer Speech and Language.* (2002) Disponible en: `http://www.research.att.com/~fsmtools/fsm/`