Documentación y Guía de Usuario de la Librería `ocaml-talf`

Índice General

Capítulo 1

Introducción

`ocaml-talf` es una librería OCaml diseñada específicamente para las clases prácticas sobre Teoría de Autómatas y Lenguajes Formales de la asignatura *Teoría de la Computación* que se imparte en la Facultad de Informática de la Universidad de La Coruña. El objetivo principal de la librería es el de reforzar la comprensión de los conceptos que conforman la materia de la asignatura, para lo cual se proporcionan implementaciones lo más acordes posible con las definiciones fomales que se presentan en las clases teóricas. La eficiencia de dichas implementaciones es también tenida en cuenta, pero constituye un objetivo secundario.

Actualmente, el núcleo principal de la librería `ocaml-talf` está formado por la definición de los tipos de datos de las estructuras algebraicas introducidas en la asignatura (expresiones regulares, gramáticas, autómatas finitos, autómatas de pila, máquinas de Turing, etc.), y por una serie de funciones de transformación que permiten construir cómodamente valores de esos tipos. No obstante, la librería está todavía en fase de desarrollo y se espera que crezca mediante la incorporación de nuevas funcionalidades y mejoras.

1.1 Descarga e instalación de la librería

La librería se encuentra en el fichero `ocaml-talf.tar.gz` que reside en el Moodle de la asignatura. Una vez descargardo, este fichero debe descomprimirse y expandirse mediante el comando

```
tar -zxvf ocaml-talf.tar.gz
```

Este proceso genera los siguientes directorios:

- El directorio `src` que contiene el código fuente de la librería.
- El directorio `doc` que contiene el presente manual.
- El directorio `data` que contiene algunos ejemplos de las estructuras algebraicas que maneja la librería.

1.2 Compilación y uso de la librería

Para compilar la librería es preciso entrar en el directorio `src` y ejecutar el comando

```
make all
```

La librería consta principalmente de cuatro módulos:

- El módulo `Conj` proporciona una implementación para realizar operaciones básicas sobre conjuntos.
- El módulo `Auto` proporciona los tipos correspondientes a las estructuras algebraicas anteriormente mencionadas y las operaciones básicas que se pueden realizar sobre ellos.
- El módulo `Ergo` proporciona funciones para crear cómodamente valores de dichos tipos desde cadenas de caracteres o ficheros, y viceversa.
- El módulo `Graf` proporciona funciones para visualizar las representaciones gráficas de los valores de dichos tipos. Para ello es necesario tener instalado el programa `dot` (un preprocesador para el dibujo de grafos dirigidos[1]).

La librería puede utilizarse para crear ejecutables, pero puede manejarse también directamente desde el lazo interactivo de OCaml, para lo cual se recomienda ejecutar las siguientes instrucciones:

```
$ ocaml
   # #load "talf.cma";;
   # open Conj;;
   # open Auto;;
   # open Ergo;;
   # open Graf;;
```

[1] Disponible en `www.graphviz.org`

Capítulo 2

Conjuntos

El módulo `Conj` proporciona una implementación basada en listas para realizar operaciones básicas sobre conjuntos de elementos de cualquier tipo. Esta implementación intenta resaltar la diferencia existente entre un conjunto y una lista, la cual se basa en los dos siguientes aspectos:

- Los conjuntos no tienen elementos repetidos, mientras que las listas sí pueden tenerlos.
- El orden de los elementos en un conjunto no es relevante, mientras que en la lista sí puede serlo.

Las siguientes secciones muestran la definición del tipo de dato para los conjuntos, así como las funciones de manejo más usuales.

2.1 Definición de conjunto

Para representar los conjuntos, el módulo `Conj` define el siguiente tipo de dato:

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

Para representar el conjunto vacío, el módulo `Conj` proporciona también el valor `conjunto_vacio : 'a Conj.conjunto`.

2.2 Operaciones básicas sobre conjuntos

Las operaciones sobre conjuntos que implementa el módulo `Conj` son las que se citan a continuación:

- `es_vacio : 'a Conj.conjunto -> bool`
  Comprueba si un conjunto es vacío o no.
- `pertenece : 'a -> 'a Conj.conjunto -> bool`
  `pertenece x c` comprueba si el elemento `x` pertenece o no al conjunto `c`.

- `agregar : 'a -> 'a Conj.conjunto -> 'a Conj.conjunto`
  `agregar x c` devuelve el conjunto resultante de añadir el elemento `x` al conjunto `c`. Si `x` ya está en `c`, el resultado es el propio `c`.

- `conjunto_of_list : 'a list -> 'a Conj.conjunto`
  A partir de una lista, esta función crea un conjunto sin elementos repetidos. Esto, unido al hecho de que el orden de los elementos dentro de un conjunto no es relevante, puede hacer que la lista soporte del conjunto difiera de la lista original.

- `suprimir : 'a -> 'a Conj.conjunto -> 'a Conj.conjunto`
  `suprimir x c` devuelve el conjunto resultante de eliminar el elemento `x` del conjunto `c`. Si `x` no está en `c`, el resultado es el propio `c`.

- `cardinal : 'a Conj.conjunto -> int`
  Devuelve el número de elementos de un conjunto.

- `union : 'a Conj.conjunto -> 'a Conj.conjunto -> 'a Conj.conjunto`
  `interseccion : 'a Conj.conjunto -> 'a Conj.conjunto -> 'a Conj.conjunto`
  `diferencia : 'a Conj.conjunto -> 'a Conj.conjunto -> 'a Conj.conjunto`
  Unión, intersección y diferencia de conjuntos.

- `incluido : 'a Conj.conjunto -> 'a Conj.conjunto -> bool`
  `incluido c1 c2` comprueba si el conjunto `c1` es o no un subconjunto del conjunto `c2`.

- `igual : 'a Conj.conjunto -> 'a Conj.conjunto -> bool`
  Comprueba si dos conjuntos son iguales o no.

- `list_of_conjunto : 'a Conj.conjunto -> 'a list`
  Dado un conjunto, esta función devuelve una lista con todos sus elementos. Es útil cuando se necesita procesar todos y cada uno de los elementos del conjunto.

- `cartesiano : 'a Conj.conjunto -> 'b Conj.conjunto -> ('a * 'b) Conj.conjunto`
  Calcula el producto cartesiano de dos conjuntos.

Para facilitar las tareas de programación, no se ha ocultado el constructor del tipo. Así pues, el usuario dispone de mecanismos para crear conjuntos con elementos repetidos. No obstante, si los conjuntos se construyen única y exclusivamente a través de las funciones descritas anteriormente, está garantizado que eso no ocurrirá.

Capítulo 3

Símbolos

Como ya se ha mencionado, el módulo `Auto` de la librería proporciona los tipos correspondientes a las estructuras algebraicas involucradas en la teoría de autómatas y lenguajes formales. La estructura más básica viene dada por la definición de los símbolos que permiten denotar los lenguajes.

3.1 Símbolos individuales

En general, existen dos tipos de *símbolos*:

- Los símbolos *terminales* son aquéllos que forman parte de las cadenas de los lenguajes.
- Los símbolos *no terminales* son aquéllos que no forman parte de las cadenas de los lenguajes, sino que sirven para ayudar a definirlos.

Para ello, el módulo `Auto` define el siguiente tipo de dato:

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

Como se puede observar, los nombres de los símbolos podrán constar de más de un caracter. El juego de caracteres válido para los símbolos está formado por todos los caracteres imprimibles, excepto por `#` (que se reserva para lo introducción de comentarios). No obstante, dicho caracter puede también incluirse en el nombre de un símbolo a través de la secuencia `\#`.

El símbolo especial *épsilon* lo representaremos mediante `Terminal ""`. El símbolo especial de inicio de pila *zeta* (necesario para la implementación de los autómatas de pila) y el símbolo especial *blanco* (necesario para la implementación de las máquinas de Turing) los representaremos mediante `No_terminal ""`.

3.2 Cadenas de símbolos

Las cadenas de símbolos se pueden especificar en modo texto mediante cadenas de caracteres donde los nombres de los símbolos aparecen separados por espacios. Por ejemplo, la cadena de caracteres

```
"a B ce"
```

es una posible representación de la lista de símbolos

```
[Terminal "a"; Terminal "B"; Terminal "ce"]
```

Además, existe la palabra reservada `epsilon`, aunque cualquier aparición de la misma será siempre ignorada. Así pues, la cadena vacía se puede representar mediante `"epsilon"` o mediante `""`. Ambas representaciones producen la lista de símbolos `[]`. Por último, existe también la palabra reservada `blanco`, la cual será siempre convertida en `No_terminal ""`.

El módulo `Ergo` incluye una serie de funciones para manejar la sintaxis anteriormente descrita. Dichas funciones son las que se citan a continuación:

- `cadena_of_string : string -> Auto.simbolo list`
  Función que dada una cadena de caracteres devuelve la lista de símbolos correspondiente.

- `cadena_of_file : string -> Auto.simbolo list`
  Función que dado un nombre de fichero que contenga una cadena de caracteres devuelve la lista de símbolos correspondiente. Cuando la especificación de una cadena de símbolos se escribe en un fichero, todo texto que aparezca desde un símbolo `#` hasta un final de línea es considerado como un comentario.

- `string_of_cadena : Auto.simbolo list -> string`
  Función que dada una lista de símbolos devuelve la cadena de caracteres correspondiente.

- `file_of_cadena : Auto.simbolo list -> string -> unit`
  Función que dada una lista de símbolos y un nombre de fichero escribe en ese fichero la cadena de caracteres correspondiente.

Una vez más, para facilitar las tareas de programación, no se han ocultado los constructores del tipo. Así pues, el usuario dispone de mecanismos para crear cadenas de símbolos inconsistentes, como por ejemplo `[Terminal ""]`. No obstante, si las cadenas de símbolos se construyen mediante las funciones descritas anteriormente, está garantizado que eso no ocurrirá.

Capítulo 4

Expresiones regulares

La siguiente estructura algebraica que implementa el módulo `Auto` es la correspondiente a las expresiones regulares, las cuales constituyen el mecanismo más cómodo para denotar los lenguajes regulares.

4.1 Definición de expresión regular

Las *expresiones regulares* se definen recursivamente como sigue:

- En primer lugar, $\emptyset$ (el conjunto vacío), ϵ (épsilon) y cualquier símbolo terminal son expresiones regulares básicas.
- Y además, si r y s son expresiones regulares, entonces también lo son: $r \cup s$ (unión), $r \cdot s$ (concatenación), y r^* (repetición).

Para ello, el módulo `Auto` define el siguiente tipo de dato:

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

Así pues, en este tipo de dato, la expresión regular épsilon debe representarse como `Constante (Terminal "")`.

4.2 Construcción de expresiones regulares

Las expresiones regulares se pueden construir a partir de una especificación en modo texto, cuya sintaxis es como sigue. Los operadores permitidos, en orden de mayor a menor prioridad son: `*` (para la repetición), `.` (para la concatenación) y `|` (para la unión). Por supuesto, se pueden utilizar paréntesis cuando sea necesario variar este orden. Por ejemplo, la cadena de caracteres

```
"a.be|ce*"
```

es una posible representación de la expresión regular

```
Union (Concatenacion (Constante (Terminal "a"),
                      Constante (Terminal "be")),
       Repeticion (Constante (Terminal "ce")))
```

mientras que

```
"a.(be|ce)*"
```

es una posible representación de

```
Concatenacion (Constante (Terminal "a"),
               Repeticion (Union (Constante (Terminal "be"),
                                  Constante (Terminal "ce"))))
```

Como se puede observar, los nombres de los símbolos pueden constar de más de un caracter. El juego de caracteres válido para los símbolos está formado por todos los caracteres imprimibles, excepto por `#` (reservado para comentarios), y por `( ) | . *` (reservados para las operaciones anteriormente descritas). No obstante, dichos caracteres pueden también incluirse en el nombre de un símbolo si aparecen precedidos de `\`. Existen también las palabras reservadas `vacio` y `epsilon` para denotar las expresiones regulares `Vacio` y `Constante (Terminal "")`, respectivamente.

El módulo `Ergo` incluye una serie de funciones para manejar la sintaxis anteriormente descrita. Dichas funciones son las que se citan a continuación:

- `er_of_string : string -> Auto.er`
  Función que dada una cadena de caracteres devuelve la expresión regular correspondiente.

- `er_of_file : string -> Auto.er`
  Función que dado un nombre de fichero que contenga una cadena de caracteres devuelve la expresión regular correspondiente. Cuando la especificación de una expresión regular se escribe en un fichero, todo texto que aparezca desde un símbolo `#` hasta un final de línea es considerado como un comentario.

- `string_of_er : Auto.er -> string`
  Función que dada una expresión regular devuelve la cadena de caracteres correspondiente.

- `file_of_er : Auto.er -> string -> unit`
  Función que dada una expresión regular y un nombre de fichero escribe en ese fichero la cadena de caracteres correspondiente.

Una vez más, para facilitar las tareas de programación, no se han ocultado los constructores del tipo. Así pues, el usuario dispone de mecanismos para crear expresiones regulares inconsistentes, como por ejemplo `Constante (No_terminal "A")`. No obstante, si las expresiones regulares se construyen mediante las funciones descritas anteriormente, está garantizado que eso no ocurrirá.

Capítulo 5

Autómatas finitos

La siguiente estructura algebraica que implementa el módulo `Auto` es la correspondiente a los autómatas finitos, los cuales constituyen el mecanismo reconocedor de los lenguajes regulares.

5.1 Definición de autómata finito

Un *autómata finito* está definido por la 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 (es decir, un conjunto de arcos o transiciones que constan de estado origen, estado destino y símbolo terminal de entrada), y F es el conjunto de estados finales. Para ello, el módulo `Auto` define los siguientes tipos de datos:

```
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);;
```

Como se puede observar, al igual que ocurre con los símbolos, los nombres de los estados de un autómata pueden constar de más de un caracter.

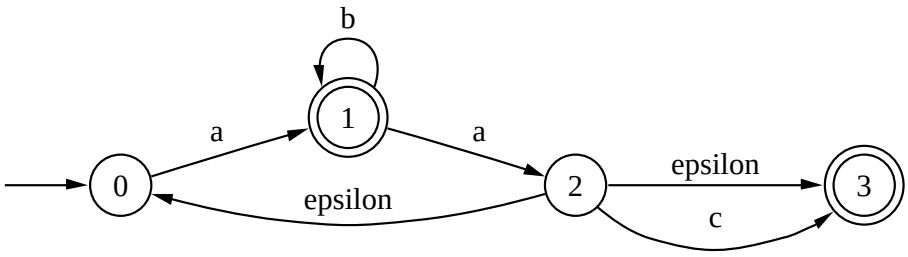
5.2 Construcción de autómatas finitos

Los autómatas finitos se pueden construir a partir de una especificación en modo texto, cuya sintaxis es como sigue. Primeramente, deberán aparecer los nombres de los estados, separados por espacios y finalizando la secuencia con un punto y coma. Seguidamente, deberá aparecer la secuencia de los nombres de los símbolos de entrada, finalizada también con un punto y coma. El juego de caracteres válido para los nombres de los estados y de los símbolos está formado por todos los caracteres imprimibles, excepto por `#` (reservado

para comentarios) y por `;` (reservado para la finalización de las secuencias de declaración, tal y como hemos visto anteriormente). No obstante, dichos caracteres pueden también incluirse en el nombre de un estado o de un símbolo si aparecen precedidos de `\`. Eso sí, el espacio de nombres de estados y el espacio de nombres de símbolos deben ser disjuntos. A continuación, deberá aparecer el nombre del estado inicial, seguido de un punto y coma. El estado inicial debe ser uno de los estados previamente declarados. Seguidamente, deberá aparecer la secuencia de los nombres de los estados finales, separados por espacios y seguida también de un punto y coma. Los estados finales deben ser también estados previamente declarados. Y por último, deberá aparecer la secuencia de arcos. Cada uno de estos arcos debe tener el formato `estado_origen estado_destino simbolo;` donde `estado_origen` y `estado_destino` deben ser estados previamente declarados, y `simbolo` puede ser un terminal previamente declarado o bien la palabra reservada `epsilon`. Así pues, por ejemplo, la cadena de caracteres

```
"0 1 2 3; a b c; 0; 1 3;
 0 1 a; 1 1 b; 1 2 a; 2 0 epsilon; 2 3 epsilon; 2 3 c;"
```

es una posible representación del autómata finito



cuya representación interna sería

```
Af (Conjunto [Estado "0"; Estado "1"; Estado "2"; Estado "3"],
    Conjunto [Terminal "a"; Terminal "b"; Terminal "c"],
    Estado "0",
    Conjunto [Arco_af (Estado "0", Estado "1", Terminal "a");
              Arco_af (Estado "1", Estado "1", Terminal "b");
              Arco_af (Estado "1", Estado "2", Terminal "a");
              Arco_af (Estado "2", Estado "0", Terminal "");
              Arco_af (Estado "2", Estado "3", Terminal "");
              Arco_af (Estado "2", Estado "3", Terminal "c")],
    Conjunto [Estado "1"; Estado "3"])
```

El módulo `Ergo` incluye una serie de funciones para manejar la sintaxis anteriormente descrita. Dichas funciones son las que se citan a continuación:

- `af_of_string : string -> Auto.af`
  Función que dada una cadena de caracteres devuelve el autómata finito correspondiente.

- `af_of_file : string -> Auto.af`
  Función que dado un nombre de fichero que contenga una cadena de caracteres devuelve el autómata finito correspondiente. Cuando la especificación de un autómata finito se escribe en un fichero, todo texto que aparezca desde un símbolo `#` hasta un final de línea es considerado como un comentario.

- `string_of_af : Auto.af -> string`
  Función que dado un autómata finito devuelve la cadena de caracteres correspondiente.

- `file_of_af : Auto.af -> string -> unit`
  Función que dado un autómata finito y un nombre de fichero escribe en ese fichero la cadena de caracteres correspondiente.

Una vez más, para facilitar las tareas de programación, no se han ocultado los constructores de los tipos. Así pues, el usuario dispone de mecanismos para crear autómatas finitos inconsistentes. En este caso, las inconsistencias no sólo se refieren a la aparición de símbolos no terminales en el alfabeto de entrada Σ o en las etiquetas de los arcos de Δ. Son también causas de inconsistencias la no equivalencia de los diferentes estados y símbolos que aparecen en los arcos de Δ con los conjuntos Q y Σ, respectivamente, o la no inclusión de los estados finales de F en Q. No obstante, si los autómatas finitos se construyen mediante las funciones descritas anteriormente, está garantizado que este tipo de fenómenos no se darán.

5.3 Autómatas finitos y expresiones regulares

Los autómatas finitos aceptan lenguajes regulares, los cuales, como hemos visto, se pueden denotar mediante expresiones regulares. En relación con esto, el módulo `Auto` implementa la siguiente función:

- `af_of_er : Auto.er -> Auto.af`
  Función que dada una expresión regular devuelve el autómata que acepta exactamente el lenguaje regular denotado por dicha expresión regular.

5.4 Funcionamiento de un autómata finito

Para simular el funcionamiento de un autómata finito en el ordenador, el módulo `Auto` implementa las siguientes funciones:

- `epsilon_cierre : Auto.estado Conj.conjunto -> Auto.af -> Auto.estado Conj.conjunto`
  Función que dado un conjunto de estados y un autómata calcula la unión de los épsilon-cierres de todos esos estados, a partir de las épsilon-transiciones del autómata.

- `escaner_af : Auto.simbolo list -> Auto.af -> bool`
  Función que dada una lista de símbolos terminales y un autómata finito indica si

dicha cadena de símbolos es aceptada o no por el autómata. Se trata de una versión de la función de reconocimiento más general posible, es decir, aquélla que es capaz de simular el funcionamiento de cualquier tipo de autómata finito (determinista, no determinista, e incluso no determinista con épsilon-transiciones).

5.5 Visualización de autómatas finitos

Para visualizar las representaciones gráficas de los autómatas finitos, el módulo `Graf` proporciona la siguiente función:

- `dibuja_af : ?titulo:string -> Auto.af -> unit`
  Función que dado un autómata finito, llama al comando `dot` para visualizar el grafo de dicho autómata en una ventana x11. Opcionalmente, se puede dar un título para el grafo.

Capítulo 6

Gramáticas independientes del contexto

La siguiente estructura algebraica que implementa el módulo `Auto` es la correspondiente a los gramáticas. Más concretamente, se trata del tipo de gramáticas que permiten denotar los lenguajes independientes del contexto.

6.1 Definición de gramática independiente del contexto

Una gramática independiente del contexto está definida por la tupla (N, T, P, S) donde: N es el conjunto de símbolos no terminales, T es el conjunto de símbolos terminales, P es el conjunto de reglas de producción, y S es un elemento destacado de N que denominamos símbolo inicial o 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 en cualquier orden. Para ello, el módulo `Auto` 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);;
```

En particular, el tipo `regla_gic` deja clara constancia de la diferencia entre una lista y un conjunto, y justifica la inclusión de los símbolos terminales y no terminales bajo un mismo tipo de dato, con el fin de que tanto unos como otros puedan aparecer juntos en la parte derecha de una regla de reescritura.

6.2 Construcción de gramáticas independientes del contexto

Las gramáticas se pueden construir a partir de una especificación en modo texto, cuya sintaxis es como sigue. Primeramente, deberán aparecer los nombres de los símbolos no terminales, separados por espacios y finalizando la secuencia con un punto y coma. Seguidamente, deberá aparecer la secuencia de los nombres de los símbolos terminales, finalizada también con un punto y coma. El juego de caracteres válido para los nombres de los estados y de los símbolos está formado por todos los caracteres imprimibles, excepto por `#` (reservado para comentarios), por `;` (reservado para la finalización de las secuencias de declaración, tal y como hemos visto anteriormente) y por `|` (reservado para que varias partes derecha de una regla compartan la misma parte izquierda, como veremos más adelante). No obstante, dichos caracteres pueden también incluirse en el nombre de un estado o de un símbolo si aparecen precedidos de `\`. Eso sí, el espacio de nombres de los no terminales y el espacio de nombres de los terminales deben ser disjuntos. A continuación, deberá aparecer el nombre del símbolo inicial, seguido de un punto y coma. El símbolo inicial debe ser uno de los no terminales previamente declarados. Y por último, deberá aparecer la secuencia de producciones o reglas de reescritura. Cada una de estas reglas debe tener el formato

```
no_terminal -> simbolo_1 simbolo_2 ... simbolo_n;
```

donde todos los símbolos que aparecen en la parte derecha de la flecha deben ser terminales o no terminales previamente declarados. Las reglas que compartan la misma parte izquierda se pueden escribir también de la forma

```
no_terminal -> parte_dcha_1 | parte_dcha_2 | ... | parte_dcha_n;
```

Existe la palabra reservada `epsilon` para denotar una parte derecha vacía. Así pues, por ejemplo, la cadena de caracteres

```
"S A B; a b c; S; S -> a A; A -> a b c A | b B; B -> b c B | epsilon;"
```

es una posible representación de la gramática

```
Gic (Conjunto [No_terminal "S"; No_terminal "A"; No_terminal "B"],
     Conjunto [Terminal "a"; Terminal "b"; Terminal "c"],
     Conjunto [
        Regla_gic (No_terminal "S", [Terminal "a"; No_terminal "A"]);
        Regla_gic (No_terminal "A",
                   [Terminal "a"; Terminal "b"; Terminal "c";
                    No_terminal "A"]);
        Regla_gic (No_terminal "A", [Terminal "b"; No_terminal "B"]);
        Regla_gic (No_terminal "B",
                   [Terminal "b"; Terminal "c"; No_terminal "B"]);
        Regla_gic (No_terminal "B", [])],
     No_terminal "S")
```

El módulo `Ergo` incluye una serie de funciones para manejar la sintaxis anteriormente descrita. Dichas funciones son las que se citan a continuación:

- `gic_of_string : string -> Auto.gic`
  Función que dada una cadena de caracteres devuelve la gramática independiente del contexto correspondiente.

- `gic_of_file : string -> Auto.gic`
  Función que dado un nombre de fichero que contenga una cadena de caracteres devuelve la gramática independiente del contexto correspondiente. Cuando la especificación de una gramática independiente del contexto se escribe en un fichero, todo texto que aparezca desde un símbolo `#` hasta un final de línea es considerado como un comentario.

- `string_of_gic : Auto.gic -> string`
  Función que dada una gramática independiente del contexto devuelve la cadena de caracteres correspondiente.

- `file_of_gic : Auto.gic -> string -> unit`
  Función que dada una gramática independiente del contexto y un nombre de fichero escribe en ese fichero la cadena de caracteres correspondiente.

Una vez más, para facilitar las tareas de programación, no se han ocultado los constructores de los tipos. Así pues, el usuario dispone de mecanismos para crear gramáticas inconsistentes. En este caso, las inconsistencias pueden referirse a la mezcla de terminales y no terminales en los conjuntos N y T, a la no equivalencia de los diferentes símbolos que aparecen en las producciones de P con los conjuntos N y T, a la presencia de un símbolo terminal como parte izquierda de una producción, a la no inclusión del axioma de la gramática en N, etc. No obstante, si las gramáticas se construyen mediante las funciones descritas anteriormente, está garantizado que no aparecerán este tipo de fenómenos.

6.3 Gramáticas regulares

Una gramática independiente del contexto es regular cuando todas las partes derechas de sus reglas de producción contienen como máximo un símbolo no terminal y, además, si dicho símbolo aparece, es siempre el último símbolo de la regla. En relación con esto, el módulo `Auto` implementa las siguientes funciones:

- `es_regular : Auto.gic -> bool`
  Función que dada una gramática independiente del contexto indica si es o no regular. La función comprueba únicamente el formato de las reglas. Por tanto, se recomienda usarla sobre gramáticas generadas con las funciones `gic_of_string` o `gic_of_file` del módulo `Ergo`, ya que las comprobaciones que no se realizan aquí las hacen automáticamente esas funciones. Así pues, en principio, si la gramática resulta no ser regular, sigue siendo una gramática correcta, lo que ocurre es que será una gramática independiente del contexto no regular.

- `af_of_gic : Auto.gic -> Auto.af`
  Función que dada una gramática regular devuelve el autómata finito que acepta exactamente el mismo lenguaje regular generado por dicha gramática.

- `gic_of_af : Auto.af -> Auto.gic`
  Función que dado un autómata finito devuelve la gramática regular que genera exactamente el mismo lenguaje regular aceptado por dicho autómata.

Capítulo 7

Autómatas de pila

La siguiente estructura algebraica que implementa el módulo `Auto` es la correspondiente a los autómatas de pila, los cuales constituyen un posible mecanismo reconocedor para los lenguajes independientes del contexto.

7.1 Definición de autómata de pila

Un *autómata de pila* está definido por la tupla $(Q, \Sigma, \Gamma, q_0, \Delta, Z, F)$ donde: Q es un conjunto de estados, Σ es el alfabeto de símbolos terminales de entrada, Γ es el alfabeto de la pila, q_0 es el estado inicial, Δ es la función de transición (es decir, 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 de inicio de pila, y F es el conjunto de estados finales. Para ello, el módulo `Auto` define 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);;
```

Como viene siendo habitual, tanto los nombres de los estados como los nombres de los símbolos pueden constar de más de un caracter. El símbolo especial de inicio de pila será representado mediante `No_terminal ""`, y existirá la palabra reservada `zeta` para denotarlo.

7.2 Construcción de autómatas de pila

Los autómatas de pila se pueden construir a partir de una especificación en modo texto, cuya sintaxis es como sigue. Primeramente, deberán aparecer los nombres de los estados, separados por espacios y finalizando la secuencia con un punto y coma. Seguidamente, deberá aparecer la secuencia de los nombres de los símbolos de entrada, finalizada también con un punto y coma. Seguidamente, deberá aparecer la secuencia de los nombres de los

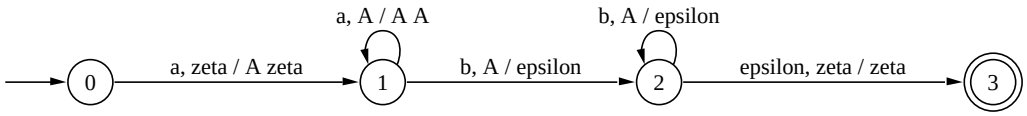
símbolos de la pila, finalizada también con un punto y coma. El juego de caracteres válido para los nombres de los estados y de los símbolos está formado por todos los caracteres imprimibles, excepto por `#` (reservado para comentarios) y por `;` (reservado para la finalización de las secuencias de declaración, tal y como hemos visto anteriormente). No obstante, dichos caracteres pueden también incluirse en el nombre de un estado o de un símbolo si aparecen precedidos de \. Eso sí, el espacio de nombres de estados, y el espacio de nombres de símbolos de entrada y de pila deben ser disjuntos entre sí. Por otra parte, existe la palabra reservada `zeta` para hacer referencia al símbolo de inicio de pila. A continuación, deberá aparecer el nombre del estado inicial, seguido de un punto y coma. El estado inicial debe ser uno de los estados previamente declarados. Seguidamente, deberá aparecer la secuencia de los nombres de los estados finales, separados por espacios y seguida también de un punto y coma. Los estados finales deben ser también estados previamente declarados. Y por último, deberá aparecer la secuencia de arcos. Cada uno de estos arcos debe tener el formato

```
estado_origen estado_destino simbolo_de_entrada
   simbolo_en_cima_de_pila nuevos_simbolos_en_cima_de_pila;
```

Así pues, por ejemplo, la cadena de caracteres

```
"0 1 2 3; a b; zeta A; 0; 3; 0 1 a zeta A zeta; 1 1 a A A A; 1 2 b A epsilon;
 2 2 b A epsilon; 2 3 epsilon zeta zeta;"
```

es una posible representación del autómata de pila



cuya representación interna sería

```
Ap (Conjunto [Estado "0"; Estado "1"; Estado "2"; Estado "3"],
    Conjunto [Terminal "a"; Terminal "b"],
    Conjunto [No_terminal ""; No_terminal "A"],
    Estado "0",
    Conjunto [Arco_ap (Estado "0", Estado "1", Terminal "a",
                       No_terminal "",
                       [No_terminal "A"; No_terminal ""]);
              Arco_ap (Estado "1", Estado "1", Terminal "a",
                       No_terminal "A",
                       [No_terminal "A"; No_terminal "A"]);
              Arco_ap (Estado "1", Estado "2", Terminal "b",
                       No_terminal "A",
                       []);
              Arco_ap (Estado "2", Estado "2", Terminal "b",
                       No_terminal "A",
                       []);
```

```
            Arco_ap (Estado "2", Estado "3", Terminal "",
                     No_terminal "",
                    [No_terminal ""])],
    No_terminal "",
    Conjunto [Estado "3"])
```

El módulo `Ergo` incluye una serie de funciones para manejar la sintaxis anteriormente descrita. Dichas funciones son las que se citan a continuación:

- `ap_of_string : string -> Auto.ap`
  Función que dada una cadena de caracteres devuelve el autómata de pila correspondiente.

- `ap_of_file : string -> Auto.ap`
  Función que dado un nombre de fichero que contenga una cadena de caracteres devuelve el autómata de pila correspondiente. Cuando la especificación de un autómata de pila se escribe en un fichero, todo texto que aparezca desde un símbolo `#` hasta un final de línea es considerado como un comentario.

- `string_of_ap : Auto.ap -> string`
  Función que dado un autómata de pila devuelve la cadena de caracteres correspondiente.

- `file_of_ap : Auto.ap -> string -> unit`
  Función que dado un autómata de pila y un nombre de fichero escribe en ese fichero la cadena de caracteres correspondiente.

Una vez más, para facilitar las tareas de programación, no se han ocultado los constructores de los tipos. Así pues, el usuario dispone de mecanismos para crear autómatas de pila inconsistentes. No obstante, si los autómatas de pila se construyen mediante las funciones descritas anteriormente, está garantizado que este fenómeno no se producirá.

7.3 Funcionamiento de un autómata de pila

Para simular el funcionamiento de un autómata de pila en el ordenador, el módulo `Auto` implementa la siguiente función:

- `escaner_ap : Auto.simbolo list -> Auto.ap -> bool`
  Función que dada una lista de símbolos terminales y un autómata de pila indica si dicha cadena de símbolos es aceptada o no por el autómata.

7.4 Visualización de autómatas de pila

Para visualizar las representaciones gráficas de los autómatas de pila, el módulo `Graf` proporciona la siguiente función:

- `dibuja_ap : ?titulo:string -> Auto.ap -> unit`
  Función que dado un autómata de pila, llama al comando `dot` para visualizar el grafo de dicho autómata en una ventana x11. Opcionalmente, se puede dar un título para el grafo.

Capítulo 8

Máquinas de Turing

La siguiente estructura algebraica que implementa el módulo `Auto` es la correspondiente a las máquinas de Turing, las cuales representan un posible mecanismo reconocedor para los lenguajes recursivos y recursivamente enumerables. Además de esto, la máquina de Turing constituye en sí misma un modelo abstracto de computación.

8.1 Definición de máquina de Turing

Una *máquina de Turing* está definida por la tupla $(Q, \Sigma, \Gamma, q_0, \Delta, B, F)$ donde: Q es un conjunto de estados, Σ es el alfabeto de símbolos terminales de entrada, Γ es el alfabeto de la cinta, q_0 es el estado inicial, Δ es la función de transición (es decir, un conjunto de arcos o transiciones que constan de estado origen, estado destino, símbolo que se lee en la cinta, símbolo que se escribe en la cinta, y movimiento que realiza la cabeza de lectura/escritura), B es el símbolo blanco, y F es el conjunto de estados finales. Para ello, el módulo `Auto` define los siguientes tipos de datos:

```
type movimiento_mt =
     Izquierda
   | Derecha;;

type arco_mt =
   Arco_mt of (estado * estado * simbolo * simbolo * movimiento_mt);;

type mt =
   Mt of (estado conjunto * simbolo conjunto * simbolo conjunto *
          estado * arco_mt conjunto * simbolo * estado conjunto);;
```

Como viene siendo habitual, tanto los nombres de los estados como los nombres de los símbolos pueden constar de más de un caracter. El símbolo blanco será representado mediante `No_terminal ""`, y existirá la palabra reservada `blanco` para denotarlo.

8.2 Construcción de máquinas de Turing

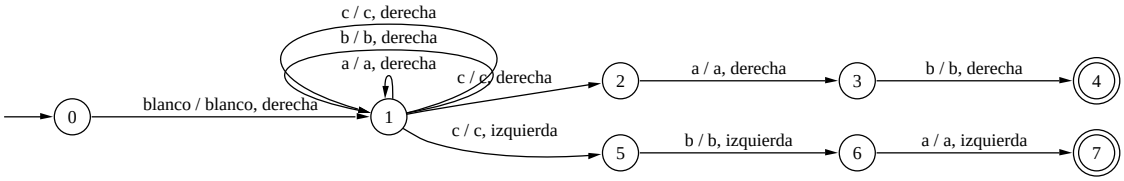
Las máquinas de Turing se pueden construir a partir de una especificación en modo texto, cuya sintaxis es como sigue. Primeramente, deberán aparecer los nombres de los estados, separados por espacios y finalizando la secuencia con un punto y coma. Seguidamente, deberá aparecer la secuencia de los nombres de los símbolos de entrada, finalizada también con un punto y coma. Seguidamente, deberá aparecer la secuencia de los nombres de los símbolos de la cinta, finalizada también con un punto y coma. El juego de caracteres válido para los nombres de los estados y de los símbolos está formado por todos los caracteres imprimibles, excepto por `#` (reservado para comentarios) y por `;` (reservado para la finalización de las secuencias de declaración, tal y como hemos visto anteriormente). No obstante, dichos caracteres pueden también incluirse en el nombre de un estado o de un símbolo si aparecen precedidos de `\`. Eso sí, el espacio de nombres de estados, y el espacio de nombres de símbolos de entrada y de la cinta deben ser disjuntos entre sí. Por otra parte, existe la palabra reservada `blanco` para hacer referencia al símbolo blanco. A continuación, deberá aparecer el nombre del estado inicial, seguido de un punto y coma. El estado inicial debe ser uno de los estados previamente declarados. Seguidamente, deberá aparecer la secuencia de los nombres de los estados finales, separados por espacios y seguida también de un punto y coma. Los estados finales deben ser también estados previamente declarados. Y por último, deberá aparecer la secuencia de arcos. Cada uno de estos arcos debe tener el formato

```
estado_origen estado_destino simbolo_de_lectura_en_la_cinta
   simbolo_de_escritura_en_la_cinta movimiento_de_la_cabeza;
```

Así pues, por ejemplo, la cadena de caracteres

```
"0 1 2 3 4 5 6 7; a b c; blanco a b c; 0; 4 7;
 0 1 blanco blanco derecha; 1 1 a a derecha; 1 1 b b derecha;
 1 1 c c derecha; 1 2 c c derecha; 2 3 a a derecha; 3 4 b b derecha;
 1 5 c c izquierda; 5 6 b b izquierda; 6 7 a a izquierda;"
```

es una posible representación de la máquina de Turing



cuya representación interna sería

```
Mt (Conjunto [Estado "0"; Estado "1"; Estado "2"; Estado "3";
              Estado "4"; Estado "5"; Estado "6"; Estado "7"],
    Conjunto [Terminal "a"; Terminal "b"; Terminal "c"],
    Conjunto [No_terminal ""; Terminal "a"; Terminal "b"; Terminal "c"],
```

```
Estado "0",
Conjunto [Arco_mt (Estado "0", Estado "1",
                   No_terminal "", No_terminal "", Derecha);
          Arco_mt (Estado "1", Estado "1",
                   Terminal "a", Terminal "a", Derecha);
          Arco_mt (Estado "1", Estado "1",
                   Terminal "b", Terminal "b", Derecha);
          Arco_mt (Estado "1", Estado "1",
                   Terminal "c", Terminal "c", Derecha);
          Arco_mt (Estado "1", Estado "2",
                   Terminal "c", Terminal "c", Derecha);
          Arco_mt (Estado "2", Estado "3",
                   Terminal "a", Terminal "a", Derecha);
          Arco_mt (Estado "3", Estado "4",
                   Terminal "b", Terminal "b", Derecha);
          Arco_mt (Estado "1", Estado "5",
                   Terminal "c", Terminal "c", Izquierda);
          Arco_mt (Estado "5", Estado "6",
                   Terminal "b", Terminal "b", Izquierda);
          Arco_mt (Estado "6", Estado "7",
                   Terminal "a", Terminal "a", Izquierda)],
No_terminal "",
Conjunto [Estado "4"; Estado "7"])
```

El módulo `Ergo` incluye una serie de funciones para manejar la sintaxis anteriormente descrita. Dichas funciones son las que se citan a continuación:

- `mt_of_string : string -> Auto.mt`
  Función que dada una cadena de caracteres devuelve la máquina de Turing correspondiente.

- `mt_of_file : string -> Auto.mt`
  Función que dado un nombre de fichero que contenga una cadena de caracteres devuelve la máquina de Turing correspondiente. Cuando la especificación de una máquina de Turing se escribe en un fichero, todo texto que aparezca desde un símbolo `#` hasta un final de línea es considerado como un comentario.

- `string_of_mt : Auto.mt -> string`
  Función que dada una máquina de Turing devuelve la cadena de caracteres correspondiente.

- `file_of_mt : Auto.mt -> string -> unit`
  Función que dada una máquina de Turing y un nombre de fichero escribe en ese fichero la cadena de caracteres correspondiente.

Una vez más, para facilitar las tareas de programación, no se han ocultado los constructores de los tipos. Así pues, el usuario dispone de mecanismos para crear máquinas de Turing inconsistentes. No obstante, si las máquinas de Turing se construyen mediante las funciones descritas anteriormente, está garantizado que este fenómeno no se producirá.

8.3 Funcionamiento de una máquina de Turing

Para simular el funcionamiento de una máquina de Turing en el ordenador, el módulo `Auto` implementa las siguientes funciones:

- `escaner_mt : Auto.simbolo list -> Auto.mt -> bool`
  Función que dada una lista de símbolos terminales y una máquina de Turing indica si dicha cadena de símbolos es aceptada o no por la máquina. Se trata de la versión más básica de la función, es decir, aquélla que simplemente indica si la máquina se detiene, y si lo hace en un estado final o de aceptación, pero no devuelve el contenido de la cinta después de la parada.

- `scpm : Auto.mt -> Auto.simbolo list -> (string * string) list`
  Función que dada una máquina de Turing y una cadena de entrada, devuelve el SCPM (Sistema de Correspondecia de Post Modificado) asociado.

8.4 Visualización de máquinas de Turing

Para visualizar las representaciones gráficas de las máquinas de Turing, el módulo `Graf` proporciona la siguiente función:

- `dibuja_mt : ?titulo:string -> Auto.mt -> unit`
  Función que dada una máquina de Turing, llama al comando `dot` para visualizar el grafo de dicha máquina en una ventana x11. Opcionalmente, se puede dar un título para el grafo.