Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación
1er cuatrimestre, 2013

Fecha de entrega: 23 de abril del 2013

Introducción

El objetivo de este trabajo es implementar una variante de los autómatas de estados infinitos, denominados "traductores", "transductores", o "Máquinas de Moore" (aunque estas últimas sólo consideran el caso finito); de ahora en adelante llamaremos a estos objetos *traductores*.

Además de implementar los traductores propiamente dichos, se implementarán ciertas operaciones sobre ellos, así como funciones que tomen traductores como entrada y determinen alguna propiedad de los mismos.

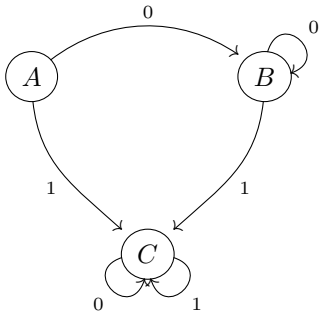
Un *traductor de un alfabeto* Σ_1 *a un alfabeto* Σ_2 consiste de un conjunto de estados Q, una función de transición $f : Q \times \Sigma_1 \to Q$, una función de traducción $g : Q \times \Sigma_1 \to \Sigma_2^*$, y un estado inicial $q_0 \in Q$. Σ_2^* es el conjunto de cadenas finitas de símbolos de Σ_2, incluyendo la vacía. Por ejemplo, si $\Sigma_2 = \{a, b\}$ entonces $\Sigma_2^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$, donde ϵ representa la cadena vacía.

El traductor siempre está en un estado $q \in Q$, inicialmente $q = q_0$. Si estando en el estado q se debe procesar un símbolo de entrada $c \in \Sigma_1$, el traductor pasa a estar en el estado $f(q, c)$ y da como salida en este paso la cadena $g(q, c)$.

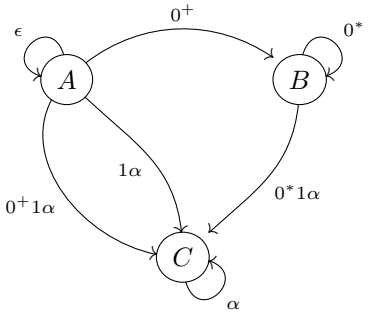
Clausura reflexo-transitiva

Dada una función de transición $f : Q \times \Sigma \to Q$, donde Q es el conjunto de estados y Σ el alfabeto, vamos a definir su *clausura reflexo-transitiva*, $f^* : Q \times \Sigma^* \to Q$, como aquella que realiza todas las transiciones dadas por f, comenzando desde el estado recibido como parámetro y procesando la cadena recibida, es decir, ejecuta el traductor y devuelve el estado en el que queda luego de procesar dicha cadena.

Por ejemplo, si se tiene que $Q = \{A, B, C\}$, $\Sigma = \{0, 1\}$, y f viene dada según:



se tiene que, entonces, f^* queda dada según:



donde ϵ representa a la cadena vacía, α representa a una cadena arbitraria, con 0^* denotamos a una cadena de longitud arbitraria (posiblemente vacía) de 0s, y con 0^+ a una de *al menos un* 0.

Este principio se puede extender y considerar también la función de traducción $g : Q \times \Sigma_1 \to \Sigma_2^*$, donde Q es el conjunto de estados (como antes), Σ_1 el alfabeto de entrada, y Σ_2 el de salida. La clausura reflexo-transitiva de g, (que notaremos $g^* : Q \times \Sigma_1^* \to \Sigma_2^*$) va a resultar de la concatenación de los resultados de aplicar g sucesivamente sobre la cadena recibida. Análogamente al caso anterior g^* ejecuta el traductor comenzando desde el estado recibido sobre la cadena recibida como parámetro y devuelve la concatenación de todas las salidas.

Notar que si bien f^* solo depende de f cuando f es una función de transición, si g es una función de traducción, g^* depende tanto de g como de la f correspondiente a las transiciones que realiza el traductor.

1. Implementación

Vamos a representar traductores de alfabeto `Char` en alfabeto `Char`. Esto es, en este trabajo práctico, $\Sigma_1 = \Sigma_2 = \texttt{Char}$ para los Σ_1 y Σ_2 que aparecen en la definición de los traductores. Notar que $\texttt{Char}^* = [\texttt{Char}] = \texttt{String}$, con lo cual $\Sigma_1^* = \Sigma_2^* = \texttt{String}$.

Supongamos que el traductor en cuestión tiene un conjunto de estados Q, una función de transición $f : Q \times \texttt{Char} \to Q$, una función de traducción $g : Q \times \texttt{Char} \to \texttt{String}$, y un estado inicial $q_0 \in Q$. Vamos a representar al traductor como una instancia del tipo `Traductor`,

```
type Traductor q = (q -> Char -> q, q -> Char -> String, q)    [1]
```

que se interpreta como sigue:

- El tipo `q` es el conjunto Q, es decir, los elementos de Q son exactamente las instancias del tipo `q`,
- la primera componente, de tipo `q -> Char -> q`, es la *función de transición*: aquélla que dado un estado actual de tipo `q`, y un carácter leído de la entrada de tipo `Char`, retorna un estado destino de tipo `q`,
- la segunda componente, de tipo `q -> Char -> String`, es la *función de traducción*: aquélla que dado un estado de tipo `q`, y un carácter leído de la entrada de tipo `Char`, retorna una cadena de salida de tipo `String` y
- la tercera componente, de tipo `q`, representa el estado inicial.

Notar que estamos representando un *conjunto* del mundo de la matemática (Q) con un *tipo* de Haskell (`q`).

1.1. Ejemplo

A modo de ejemplo, consideremos el problema de escribir un traductor que cambie las `a`es por `e`s y viceversa, con signatura `cambiarAE :: Traductor ()`[1]:

```
cambiarAE :: Traductor ()                                    [2]
cambiarAE = (const, g, ())                                   [3]
  where                                                      [4]
    g :: () -> Char -> String                                [5]
    g () 'a' = "e"                                           [6]
    g () 'e' = "a"                                           [7]
    g ()  x  = [x]                                           [8]
```

Como podemos ver, `cambiarAE` es un `Traductor ()` (ie. un traductor que tiene un único estado, a saber: `()`), su función de transición es la constante `()` para cualquier carácter leído desde el estado `()`, y por último, es justamente `()` su estado inicial (y único).

1.2. Ejercicio

Escribir un traductor que intercambie caracteres consecutivos, con signatura `intercambiarConsecutivos :: Traductor (Maybe Char)`. Se puede asumir que la cadena de entrada, en caso de ser finita, tendrá longitud par.

Por ejemplo, el traductor aplicado a la cadena `abcdbacc` debería devolver `badcabcc`.

[1] Recordemos que `()` es el nombre del tipo *Unit* de Haskell, además `()` es el único objeto que pertenece a ese tipo.

1.3. Ejemplo

Como ejemplo, escribamos un traductor que se comporte como la identidad, salvo que nunca genera salida de las `a`es que encuentra en la entrada, y, cuando aparece una `z`, muestra dicha `z` y luego todas las `a`es que se acumularon juntas; su signatura ha de ser `acumularAes :: Traductor Int`.

```
acumularAes :: Traductor Int                                    [9]
acumularAes = (f, g, 0)                                        [10]
  where                                                        [11]
    f :: Int -> Char -> Int                                    [12]
    f x 'a' = x + 1                                            [13]
    f x 'z' = 0                                                [14]
    f       = const                                            [15]
    --                                                         [16]
    g :: Int -> Char -> String                                 [17]
    g x 'z' = 'z':(replicate x 'a')                            [18]
    g x 'a' = [ ]                                              [19]
    g x  c  = [c]                                              [20]
```

Este traductor cuenta con un espacio de estados de tipo `Int`[2], entre los cuales `0` es el inicial.

Su función de transición `f` se queda en el mismo estado en el que estaba (`x`, en el listado) si el carácter `c` que se leyó *no* es ni `a` ni `z`; en caso de éste ser una `a`, se pasa al estado `x + 1` (ie. estamos utilizando los estados para "contar" ocurrencias de `a`es); por último, en caso de haber leído una `z`, se pasa al estado inicial nuevamente (ie. una vez que haya leído una `z` todas las `a`es acumuladas son escritas en la salida y no hay más acumuladas).

La línea 16 es sólo estética.

Por último, su función de traducción retorna el carácter leído siempre y cuando no sea éste una `a` ni una `z`; retorna la lista vacía en caso de leer una `a` (ignoramos las `a`es... ¡pero las contamos en los estados!); y al leer una `z` la copia a la salida, seguida de tantas `a`es como el estado en el que se encuentre especifique.

1.4. Ejercicio

Escribir un traductor que dé vuelta (ie. espeje) todo lo que en la entrada esté entre dos `a`es consecutivas o entre el comienzo y la primera `a`, con signatura `espejarEntreAes :: Traductor String`.

Por ejemplo, el traductor aplicado a la cadena `123a456aa789a` debería devolver `321a654aa987a`.

Precondición Se puede asumir que la cadena o bien termina con `a`, en caso de ser finita, o bien tiene infinitas `a`es.

[2] Los enteros positivos habrían alcanzado, pero Haskell no cuenta con ellos...

1.5. Ejercicio

Escribir una función que, dada una función de transición f, calcula la clausura reflexo-transitiva f^* de la misma. Damos a continuación el código de una tal función utilizando recursión explícita, reescribirla para que utilice esquemas de recursión.

```
fAst' :: (q -> Char -> q) -> q -> String -> q                [21]
fAst' _ q0    ""  = q0                                       [22]
fAst' d q0 (c:cs) = fAst' f (f q0 c) cs                      [23]
```

Su signatura ha de ser `fAst :: (q -> Char ->  q) -> q -> String -> q`.

Por ejemplo, en el ejemplo 1.3 dado mas arriba, $f^*(0, \texttt{cazaa}) = 2$, es decir `(let (f,g,0) = acumularAes in (fAst f) 0 "cazaa") = 2`.

1.6. Ejercicio

Escribir la función que, dada una función de transición y una función de traducción, calcula la clausura reflexo-transitiva de esta última. Damos a continuación el código de una tal función utilizando recursión explícita, reescribirla para que utilice esquemas de recursión.

```
gAst' :: (q -> Char -> q) -> (q -> Char -> String) ->        [24]
         q -> String -> String                               [25]
gAst' f g q0    ""  = ""                                     [26]
gAst' f g q0 (c:cs) = (o q0 c) ++ gAst' f g (f q0 c) cs      [27]
```

Su signatura ha de ser `gAst :: (q -> Char -> q) -> (q -> Char -> String) -> q -> String -> String`

Por ejemplo, en el ejemplo 1.3 dado mas arriba, $g^*(0, \texttt{cazaa}) = \texttt{cza}$, es decir `(let (f,g,0) = acumularAes in (gAst f g) 0 "cazaa") = "cza"`.

Nota Esta función puede presentar cierta dificultad. En caso de trabarse, es recomendable que continúen con el resto del trabajo, utilizando esta versión recursiva (ie. haciendo `gAst = gAst'`) para el resto y retomando este ejercicio más adelante.

1.7. Ejemplo

A título de ejemplo, y como facilidad adicional, presentamos la función `aplicando`, que dado un traductor, retorna una función `String -> String` que traduce su entrada a la salida que el traductor generaría:

```
aplicando :: Traductor q -> String -> String                 [28]
aplicando (f, g, q) = gAst f g q                             [29]
```

Es decir: alcanza con calcular la clausura de Kleene de la función de traducción `g` con la función de transición `f`, y aplicarla (parcialmente) al estado inicial `q`.

1.8. Ejercicio

Escribir la función que, dados dos traductores, devuelve un traductor tal que la función `String -> String` que denota el resultado, sea justo la composición de las funciones de cada uno de los parámetros; su signatura ha de ser `comp :: Traductor q1 -> Traductor q2 -> Traductor (q1, q2)`. Formalmente se requiere que `aplicando (comp t1 t2)` sea la misma función que `(aplicando t1) . (aplicando t2)`.

Ayuda Notar que ya se les dice cuál es el conjunto de estados del resultado. La idea es ir simulando los dos traductores simultáneamente, mientras al traductor 2 se lo alimenta con la entrada recibida, al traductor 1 se lo alimenta con la salida del traductor 2. La salida de la composición es la salida que produce el traductor 1.

1.9. Ejercicio

Escribir la función que, dado un traductor, retorna la cadena posiblemente infinita que resulta de procesar a través del mismo una lista infinita de `a`es, con signatura `salidaAes :: Traductor q -> String`.

Nota Para todo `n`, si se pide `take n (salidaAes t)`, la función debe eventualmente terminar y dar la salida correcta. Tengan especial cuidado en las recursiones utilizadas tanto en este ejercicio como en aquéllos que utilicen para resolver este.

1.10. Ejercicio

Escribir una función que decida si es posible que un traductor `t` dado genere la salida `w` dada como segundo parámetro como salida de una entrada numérica, formalmente, si existe una cadena `s` compuesta únicamente por números tal que `aplicando t s = w`. La función pedida debe tener signatura `salidaPosible :: Traductor q -> String -> Bool`.

Precondición En este ejercicio, el traductor tomado como parámetro *no contiene transiciones cuya traducción resulte en la cadena vacía*, es decir, si la función de traducción del traductor tomado como parametro es `g`, vale que, para todo estado `q` y todo carácter `c`, `length (g q c)` es mayor o igual a 1.

2. Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función debe contar con un comentario donde se explique su funcionamiento. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del nombre del grupo.

- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado. Los objetivos a evaluar en la implementación de las funciones son:

- corrección,
- declaratividad,
- reutilización de funciones previamente definidas (teniendo en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos, como así también las de `Prelude` u otros módulos de Haskell importados),
- uso de funciones de alto orden, currificación y esquemas de recursión.

No se permite utilizar recursión explícita, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar esquemas de recursión definidos en el preludio. Pueden escribirse todas las funciones auxiliares que se requieran, pero éstas no pueden ser recursivas (ni mutuamente recursivas).

Pueden utilizar cualquier función definida en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Function`, `Data.List`, `Data.Maybe`, y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas.

Importante Se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

3. Referencias

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en `http://www.haskell.org/onlinereport/haskell2010`.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en `http://learnyouahaskell.com/chapters`.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en `http://book.realworldhaskell.org/read`.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en `http://www.haskell.org/hoogle`.

- **Hayoo!**: buscador de módulos no estándard (ie. aquéllos no necesariamene incluídos con la plataforma Haskell, sino a través de **Hackage**), online en `http://holumbus.fh-wedel.de/hayoo/hayoo.html`.

El código proporcionado por la cátedra para este trabajo práctico se da, tanto a continuación como en forma de *attachment* a este mismo archivo:

```
import Prelude                                                          [1]
                                                                        [2]
-- El traductor se representa por un conjunto de estados "q",           [3]
--   - una funcion de transicion (primer parametro),                    [4]
--   - una funcion de output (segundo parametro) y                      [5]
--   - un estado inicial.                                               [6]
type Traductor q = (q -> Char -> q, q -> Char -> String, q)             [7]
                                                                        [8]
-- Traductor que cambia las "a"es por "e"s y viceversa.                 [9]
cambiarAE :: Traductor ()                                              [10]
cambiarAE = (const, g, ())                                             [11]
  where                                                                [12]
    g :: () -> Char -> String                                          [13]
    g () 'a' = "e"                                                     [14]
    g () 'e' = "a"                                                     [15]
    g ()  x  = [x]                                                     [16]
                                                                       [17]
-- Traductor que intercambia caracteres consecutivos.                  [18]
intercambiarConsecutivos :: Traductor (Maybe Char)                     [19]
-- EJERCICIO                                                           [20]
                                                                       [21]
-- Traductor que sea la identidad, salvo que nunca genera              [22]
-- salida output de las "a"es y, cuando aparece una "z",               [23]
-- muestra la "z" y luego todas las "a"es que se acumularon            [24]
-- juntas.                                                             [25]
acumularAes :: Traductor Int                                           [26]
acumularAes = (f, g, 0)                                                [27]
  where                                                                [28]
    f :: Int -> Char -> Int                                            [29]
    f x 'a' = x + 1                                                    [30]
    f x 'z' = 0                                                        [31]
    f x  c  = x                                                        [32]
    --                                                                 [33]
    g :: Int -> Char -> String                                         [34]
    g x 'z' = 'z':(replicate x 'a')                                    [35]
    g x 'a' = [ ]                                                      [36]
    g x  c  = [c]                                                      [37]
                                                                       [38]
-- Traductor que de vuelta (ie. espeje) todo lo que esta               [39]
-- entre dos "a"es consecutivas.                                       [40]
espejarEntreAes :: Traductor String                                    [41]
-- EJERCICIO                                                           [42]
                                                                       [43]
-- Calcular la clausura de Kleene de la funcion de                     [44]
```

```
-- transicion pasada como parametro                                [45]
-- (version recursiva explicita).                                  [46]
fAst' :: (q -> Char -> q) -> q -> String -> q                       [47]
fAst' d q0 ""     = q0                                             [48]
fAst' d q0 (c:cs) = fAst' d (d q0 c) cs                            [49]
                                                                   [50]
-- Calcular la clausura de Kleene de la funcion de                 [51]
-- transicion pasada como parametro                                [52]
-- (version con esquemas de recursion).                            [53]
fAst :: (q -> Char -> q) -> q -> String -> q                       [54]
-- EJERCICIO                                                       [55]
                                                                   [56]
-- Calcular la clausura de Kleene de la funcion de                 [57]
-- salida pasada como parametro junto con la funcion               [58]
-- de transicion pasada como parametro                             [59]
-- (version recursiva explicita).                                  [60]
gAst' :: (q -> Char -> q) -> (q -> Char -> String) ->              [61]
         q -> String -> String                                     [62]
gAst' d o q0 ""     = ""                                           [63]
gAst' d o q0 (c:cs) = (o q0 c) ++ gAst' d o (d q0 c) cs            [64]
                                                                   [65]
-- Calcular la clausura de Kleene de la funcion de salida          [66]
-- pasada como parametro junto con la funcion de                   [67]
-- transicion pasada como parametro                                [68]
-- (version con esquemas de recursion).                            [69]
gAst :: (q -> Char -> q) -> (q -> Char -> String) ->               [70]
        q -> String -> String                                      [71]
-- EJERCICIO                                                       [72]
                                                                   [73]
-- Dado un traductor, retornar la funcion String -> String         [74]
-- que resulta al aplicarlo a cualquier entrada                    [75]
aplicando :: Traductor q -> String -> String                       [76]
aplicando (f, g, q) = gAst f g q                                   [77]
                                                                   [78]
-- Dados dos traductores, dar un traductor tal que la              [79]
-- funcion String -> String que denota el resultado, sea           [80]
-- justo la composicion de las funciones de cada                   [81]
-- uno de los parametros.                                          [82]
comp :: Traductor qq1 -> Traductor qq2 ->                          [83]
        Traductor (qq1, qq2)                                       [84]
-- EJERCICIO                                                       [85]
                                                                   [86]
-- Dado un traductor, dar la cadena infinita que resulta de        [87]
-- procesar una lista infinita de "a"es (si se pide                [88]
-- "take n (salidaAes t)" no puede procesar infinitamente          [89]
-- para ningun "n")                                                [90]
salidaAes :: Traductor q -> String                                 [91]
-- EJERCICIO                                                       [92]
                                                                   [93]
-- Decidir si es posible que el traductor dado de la salida        [94]
```

```
-- dada como segundo parametro                                    [95]
salidaPosible :: Traductor q -> String -> Bool                    [96]
-- EJERCICIO                                                      [97]
```

Haciendo click acá, se extrae el archivo Haskell correspondiente.

Por último, acá está el código fuente de este archivo propiamente dicho.