Universidad Nacional Autónoma de México

Facultad de Ciencias

Una implementación funcional de máquinas de Turing

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
JOSÉ RAMOS RAMOS

DIRECTOR DE TESIS:
DR. FAVIO EZEQUIEL MIRANDA PEREA

2012

1. Datos del alumno	1. Datos del alumno
Apellido paterno	Ramos
Apellido materno	Ramos
Nombre(s)	José
Teléfono	56 17 45 24
Universidad	Universidad Nacional Autónoma de México
Facultad	Facultad de Ciencias
Carrera	Ciencias de la Computación
Número de cuenta	300110607
2. Datos del tutor	2. Datos del tutor
Grado	Dr.
Nombre(s)	Favio Ezequiel
Apellido paterno	Miranda
Apellido materno	Perea
3. Datos del sinodal 1	3. Datos del sinodal 1
Grado	Dra.
Nombre(s)	Elisa
Apellido paterno	Viso
Apellido materno	Gurovich
4. Datos del sinodal 2	4. Datos del sinodal 2
Grado	Dr.
Nombre(s)	Carlos
Apellido paterno	Torres
Apellido materno	Alcaraz
5. Datos del sinodal 3	5. Datos del sinodal 3
Grado	Dr.
Nombre(s)	Francisco
Apellido paterno	Hernández
Apellido materno	Quiroz
6. Datos del sinodal 4	6. Datos del sinodal 4
Grado	M. en C.
Nombre(s)	Araceli Liliana
Apellido paterno	Reyes
Apellido materno	Cabello
7. Datos del trabajo escrito	7. Datos del trabajo escrito
Título	Una implementación funcional de máquinas de Turing
Número de páginas	107 p
Año	2012

Índice general

1 Introducción

El objetivo de este trabajo consiste en presentar una implementación de máquinas de Turing utilizando el lenguaje de programación Haskell. Primero se enuncian las definiciones clásicas como: definición de una máquina de Turing estándar y sus variantes, lenguaje aceptado por una máquina de Turing, lenguaje traducido por una máquina de Turing, lenguaje generado por una máquina de Turing y por último se aborda el concepto de la máquina universal de Turing, siendo este último quizás el tema más importe de este trabajo. Además, se ilustran mediante ejemplos algunos elementos del lenguaje de programación que se van a utilizar. Esto se hace en el siguiente capítulo, el cual se recomienda sea leído con cuidado a aquellas personas que no estén familiarizadas con las máquinas de Turing mientras que aquellas que conocen el tema analicen la estructura de las definiciones.

Posteriormente presentamos nuestras definiciones, las cuales han sido ligeramente modificadas con respecto a las definiciones clásicas procurando que el paso de nuestras definiciones a una implementación en Haskell sea prácticamente inmediato, tan sólo aprovechando de forma adecuada los elementos del lenguaje. Cabe aclarar que no sólo las implementaciones de las definiciones son inmediatas sino que también las funciones que determinan el comportamiento de la máquina de Turing. A continuación se explica el motivo por el cuál hemos decidido utilizar Haskell.

Una máquina de Turing, como cualquier máquina, toma una entrada, la procesa y produce una salida. En este caso, cualquier objeto de entrada lo podemos representar como una cadena. El estudio elemental de una máquina de Turing se hace sobre un lenguaje, mismo que se puede ver como una lista de cadenas y es aquí en donde surge la importancia de utilizar Haskell, ya que nos permite definir listas infinitas gracias al mecanismo de evaluación perezosa mediante una estructura llamada *lista por comprensión*, ventaja fundamental ya que los números naturales, los números enteros, los palíndromos sobre un alfabeto, etc., son conjuntos infinitos, los cuales en Haskell se pueden implementar y manipular por medio de listas sin mayor complicación.

Para realizar la implementación de los módulos que se exponen en este trabajo, se ha explotado la recursividad, ya que por su simplicidad de comprensión y su gran potencia, favorece a la resolución de problemas de manera natural, sencilla y elegante. Sin embargo, en caso de que nos preocupe la eficiencia, esta técnica no es la mejor, ya que implica un

crecimiento considerable en tiempo y memoria, dado que para permitir su uso es necesario transformar el programa recursivo en otro iterativo que utiliza ciclos y pilas para almacenar variables, y así garantizar que cualquier caso del paso recursivo siempre converja al caso base de la definición.

Haskell es un lenguaje funcional, lo que significa que todo se puede ver como una función, de hecho, se puede pasar una función como parámetro para que alimente la defición de otras funciones. Esto es esencial ya que entre los elementos que definen una máquina de Turing existe una función que es la encargada de procesar la cadena de entrada. La máquina se recibe como un todo y posteriormente es dividida en dos partes: las transiciones (las cuales definen a nuestra función) y el resto de los elementos.

Una máquina de Turing no sólo se utiliza para determinar si una cadena (objeto de estudio) pertenece o no a un lenguaje (respuesta verdadero o falso), sino que también produce salidas que pudieran ser conjuntos o listas infinitas; en este caso, por ejemplo, sabemos que una máquina de Turing sirve para generar lenguajes en donde cada elemento tiene que cumplir ciertas características y también se utiliza para computar lenguajes transformando una cadena en otra, pero en ambos casos no sabemos el tamaño del conjunto (lista) de salida, lo cual deja de inquietarnos el momento de utilizar las *listas por comprensión* ya que éstas serán las encargadas de manejar estos detalles.

A continuación, se mencionan brevemente los contenidos de los capítulos posteriores:

Capítulo 2. Preliminares. Se divide en dos secciones, la primera cubre los aspectos necesarios de la máquina de Turing vitales para entender este trabajo, mientras que en la segunda se presentan características y generalidades de Haskell.

En el apartado de la máquina de Turing se indica qué es, para qué se utiliza y se enuncian las definiciones clásicas como: configuración de la cinta, ejecución de un paso de cómputo, lenguaje aceptado por una máquina de Turing estándar, lenguaje generado por una máquina de Turing estándar y cómputo de una función por una máquina de Turing estándar. De manera análoga se presentan las variantes de una máquina de Turing estándar.

En cuanto a la segunda sección se da una breve introducción a Haskell, qué son las listas por comprensión y qué es la evaluación perezosa.

Capítulo 3. Máquina de Turing estándar. Primero se hacen algunas modificaciones a las definiciones clásicas de la máquina de Turing para después dar una implementación del modelo estándar y sus utilidades, con un listado previo de las funciones auxiliares básicas haciendo énfasis en lo inmediato que resulta la implementación basada en nuestra teoría. Desde aquí se hace presente la ventaja que tiene Haskell sobre otros lenguajes cuando se tratan temas matemáticos.

Capítulo 4. Variantes de la máquina de Turing estándar. Trabajamos sobre algunas variantes de la máquina de Turing estándar, tales como máquinas multipista, máquinas multicinta y máquinas no deterministas. También se define el lenguaje aceptado por cada una de estas variantes y la forma en la que cada una de ellas funciona. Se realizan las implementaciones correspondientes a estos conceptos.

Capítulo 5. Máquina universal de Turing. En este capítulo se trabaja sobre un tema fundamental como es la *máquina universal de Turing*. Se presenta la teoría general y se desarrolla una implementación de la misma. Además, se trabajan dos secciones que complementan a la máquina universal de Turing como son la codificación y la decodificación de una máquina estándar.

Capítulo 6. Aplicaciones de la máquina de Turing. Se proporcionan dos aplicaciones de las máquinas de Turing como son: simulación de autómatas finitos deterministas y simulación de autómatas de pila sin transiciones ε por medio de máquinas de Turing. En la primera aplicación se utiliza el modelo estándar mientras que en la segunda se utiliza el modelo multicinta. Para cada una se muestra el código y se ilustra con un ejemplo.

2 Preliminares

La máquina de Turing surgió ante la propuesta de demostrar la existencia de inteligencia en una máquina bajo una prueba conocida como *prueba de Turing* la cual es expuesta en el artículo "Computing machinery and intelligence"[7]. Esta prueba sigue siendo uno de los mejores métodos para los defensores de la Inteligencia Artificial. Se fundamenta en la hipótesis positivista de que, si una máquina se comporta en todos los aspectos como inteligente, entonces debe ser inteligente.

La prueba consiste en el siguiente desafío. Se supone un juez situado en una habitación, una máquina y un ser humano en otras. El juez debe descubrir cuál es el ser humano y cuál es la máquina, estándoles a los dos permitido mentir al contestar por escrito las preguntas que el juez les hiciera. La tesis de Turing es que si ambos jugadores son suficientemente hábiles, el juez no podrá distinguir quién es el ser humano y quién la máquina. Todavía ninguna máquina puede pasar este examen en una experiencia con método científico. De hecho, para incentivar el desarrollo de una máquina que pase este examen se inició un concurso en 1990 en el cual se otorga el *Premio Loebner*; esta competencia es de carácter anual entre programas de computadora que siguen los estándares establecidos en la prueba de Turing. Un juez humano se enfrenta a dos pantallas, una de ellas que se encuentra bajo el control de una computadora, y la otra bajo el control de un humano. El juez plantea preguntas a las dos pantallas y recibe respuestas. El premio está dotado con 100,000 dólares para el programa que pase la prueba, y un premio de consolación para el mejor programa anual. Todavía no ha sido otorgado el premio principal.

Esta prueba tiene diversas aplicaciones, entre otras se utiliza para detectar si una solicitud de inscripción fue contestada por un humano o por una máquina bajo una idea simple que consiste en que el usuario introduzca un conjunto de caracteres que se muestran en una imagen distorsionada que aparece en pantalla. Se supone que una máquina no es capaz de comprender e introducir la secuencia de forma correcta, por lo que solamente el humano podría hacerlo.

2.1 Máquina de Turing

Una *máquina de Turing* es un modelo matemático abstracto que formaliza el concepto de algoritmo. Como toda máquina trabaja en tres pasos básicos: tomar el objeto de entrada, procesarlo y generar una salida. En general, son utilizadas para estudiar lenguajes de forma "automática", esto es, alimentamos la máquina con algún lenguaje y manipulamos la salida dependiendo del problema que estamos estudiando. Se presentan tres aplicaciones de la máquina en cuanto a lenguajes se refiere, las cuales son: lenguaje aceptado, lenguaje traducido o cómputo de una función, y lenguaje generado por una máquina de Turing.

La idea es automatizar los pasos primitivos que hacen las computadoras humanas al momento de realizar ciertas operaciones. Se trabaja con dos conjuntos de símbolos o *alfabetos* uno de entrada y otro de salida sobre los cuales se define el lenguaje que se quiere estudiar. Cuando trabajamos sobre una cadena, vamos pasando de un estado a otro hasta concluir la tarea, por lo que se usa un *conjunto de estados* entre los cuales se encuentran dos tipos especiales: el *estado inicial* y el *conjunto de estados finales*. En lugar de papel, se utiliza una cinta lineal potencialmente infinita en ambas direcciones, dividida en sectores, cada uno de los cuales tiene espacio para un símbolo de la cadena que está siendo procesada. Para saber qué símbolo es el que se está analizando se utiliza un elemento de control llamado *cabeza de la cinta*, que en todo momento apunta a un sector de la cinta (sector actual). Esta cabeza se encarga de llevar a cabo una transición ejecutando los siguientes pasos:

1. Reemplazar el símbolo en el sector actual con otro, posiblemente distinto.
2. Mover la cabeza de la cinta al sector de la izquierda o al sector de la derecha. Pero también puede ser que la cabeza no se mueva y permanezca en el mismo sector.
3. Cambiar el estado actual de la máquina a otro, posiblemente distinto.

2.1.1. Máquina de Turing estándar (MTE)

Pasamos a las definiciones formales de la máquina de Turing y su funcionamiento, suponiendo que el lector tiene conocimientos sobre los conceptos básicos de conjuntos, funciones y lenguajes.

Definición 2.1.1 (Máquina de Turing estándar)**.** Una máquina de Turing estándar (MTE) es una 6-tupla $T = (Q, F, \Sigma, \Gamma, q_0, \delta)$ en donde:

- Q es un conjunto finito de estados, $Q \neq \varnothing$.
- F es un conjunto de estados finales con $F \subseteq Q$.
- Σ es un conjunto finito de símbolos de entrada, $\Sigma \neq \varnothing$, conocido como el alfabeto de entrada.
- Γ es un conjunto finito de símbolos de cinta con $\Sigma \subseteq \Gamma$, conocido como el alfabeto de la cinta; se supone que Γ no contiene a $\flat$, el símbolo de espacio en blanco, $\Gamma \neq \varnothing$.

- q_0 estado inicial con $q_0 \in Q$.
- δ es la función parcial con la cual se rige la MTE con:

$$\delta : Q \times (\Gamma \cup \{\flat\}) \longrightarrow Q \times (\Gamma \cup \{\flat\}) \times \{I, N, D\},$$

la terna $\{I, N, D\}$ indica las tres posibles direcciones a las que se moverá la cabeza de la cinta: izquierda, no se mueve o derecha, respectivamente.

Definición 2.1.2 (Configuración de la cinta de una MTE)**.** Una configuración de la cinta de una MTE es un par:

$$(q, x\underline{a}y)$$

donde $q \in Q$, $x, y \in (\Gamma \cup \{\flat\})^*$ y $a \in \Gamma \cup \{\flat\}$, siendo a el símbolo al que está apuntando la cabeza de la cinta. La *configuración inicial* para la entrada $w = ay$ es:

$$(q_0, \underline{a}y)$$

donde q_0 es el estado inicial de la MTE, $y \in (\Gamma \cup \{\flat\})^*$ y $a \in \Gamma \cup \{\flat\}$.

Definición 2.1.3 (Paso computacional $\vdash$)**.** El paso de una configuración a otra cuando se aplica una transición definida en δ se conoce como paso computacional y se denota por el símbolo $\vdash$.

Para indicar que hemos realizado cero o más pasos computacionales utilizamos el símbolo $\vdash^*$.

Lenguaje aceptado por una MTE

Definición 2.1.4 (Lenguaje aceptado por una MTE)**.** Sean $T = (Q, F, \Sigma, \Gamma, q_0, \delta)$ una MTE y $x \in \Sigma^*$, decimos que x es aceptada por T si y sólo si existen $y, z \in (\Gamma \cup \{\flat\})^*$ y $a \in \Gamma \cup \{\flat\}$ tales que:

$$(q_0, x) \vdash^* (p, y\underline{a}z)$$

en donde $p \in F$ y q_0 está apuntando al primer símbolo de x en caso de haberlo. Entonces, el lenguaje aceptado por T $L(T)$, es el conjunto de cadenas de entrada que acepta T.

Para definir una MTE se puede utilizar un diagrama de estados o se podría definir de manera directa definiendo la función de transición δ. En este capítulo recurriremos a la primera técnica.

Definición 2.1.5 (Diagrama de estados)**.** Un diagrama de estados define gráficamente los estados por los que pasa un objeto como respuesta a la aplicación de diferentes eventos. En el caso particular de este trabajo, un diagrama de estados consta de los siguientes elementos:

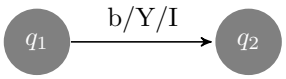
1. start: señala el estado inicial de la máquina.
2. q_{fn}: n - ésimo estado final de la máquina.

3. se, ss, M: terna que define el símbolo de entrada, el símbolo de salida y el movimiento de la máquina. Por ejemplo, el siguiente diagrama indica que cuando la máquina esté en el estado q_1 y lea el símbolo b se tienen que ejecutar los siguientes pasos:

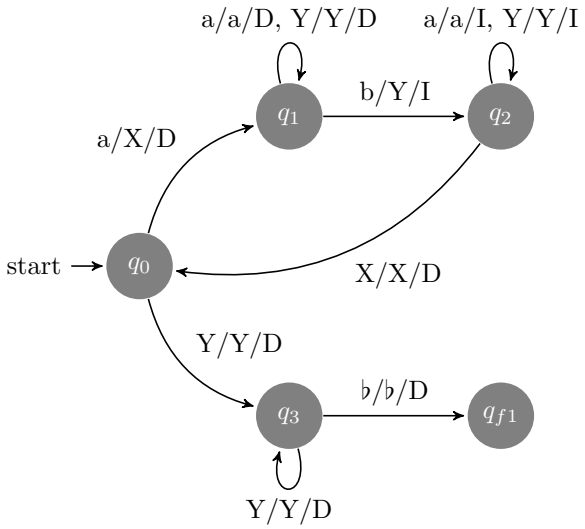
   a) Reemplazar el símbolo b por el símbolo Y del sector que está siendo leído en la cinta.

   b) Pasar del estado q_1 al estado q_2.

   c) Mover la cabeza al sector que está a la izquierda del sector actual de la cinta.



Ejemplo 1. Consideremos T_1 una MTE que acepta el lenguaje $\{a^n b^n$ donde $n \geq 0\}$ y cuyo diagrama de estados es:



Tomemos dos cadenas $w_1 = aabb$ y $w_2 = aabba$ tales que $w_1 \in L(T_1)$, $w_2 \notin L(T_1)$. El comportamiento de T_1 con cada cadena se muestra a continuación:

$$
\begin{array}{lllllllll}
(q_0, \underline{a}abb) & \vdash & & & & & & & \\
 & \vdash & (q_1, X\underline{a}bb) & \vdash & (q_1, Xa\underline{b}b) & \vdash & (q_2, X\underline{a}Yb) & \vdash & \\
 & \vdash & (q_2, \underline{X}aYb) & \vdash & (q_0, X\underline{a}Yb) & \vdash & (q_1, XX\underline{Y}b) & \vdash & \\
 & \vdash & (q_1, XXY\underline{b}) & \vdash & (q_2, XX\underline{Y}Y) & \vdash & (q_2, X\underline{X}YY) & \vdash & \\
 & \vdash & (q_0, XX\underline{Y}Y) & \vdash & (q_3, XXY\underline{Y}) & \vdash & (q_3, XXYY\underline{\flat}) & \vdash & \\
 & \vdash & (q_{f1}, XXYY\flat\underline{\flat}) & & & & & &
\end{array}
$$

Por lo tanto la cadena *aabb* es aceptada por T_1.

Mientras que:

$$\begin{array}{lllllllll}
(q_0, \underline{a}abba) & \vdash & & & & & & & \\
& \vdash & (q_1, X\underline{a}bba) & \vdash & (q_1, Xa\underline{b}ba) & \vdash & (q_2, X\underline{a}Yba) & \vdash & \\
& \vdash & (q_2, \underline{X}aYba) & \vdash & (q_0, X\underline{a}Yba) & \vdash & (q_1, XX\underline{Y}ba) & \vdash & \\
& \vdash & (q_1, XXY\underline{b}a) & \vdash & (q_2, XX\underline{Y}Ya) & \vdash & (q_2, X\underline{X}YYa) & \vdash & \\
& \vdash & (q_0, XX\underline{Y}Ya) & \vdash & (q_3, XXY\underline{Y}a) & \vdash & (q_3, XXYY\underline{a}) & \vdash & \\
& \nvdash & & & & & & &
\end{array}$$

Por lo tanto la cadena *aabba* no es aceptada por T_1.

En el caso de una MTE como aceptadora de lenguajes nos olvidamos de la cinta una vez que obtenemos una respuesta que nos indique si la cadena fue o no aceptada, pero en la funcionalidad que se presenta en la siguiente sección nos interesará el contenido de la cinta una vez terminado el proceso.

MTE que computa una función (MTE como traductora)

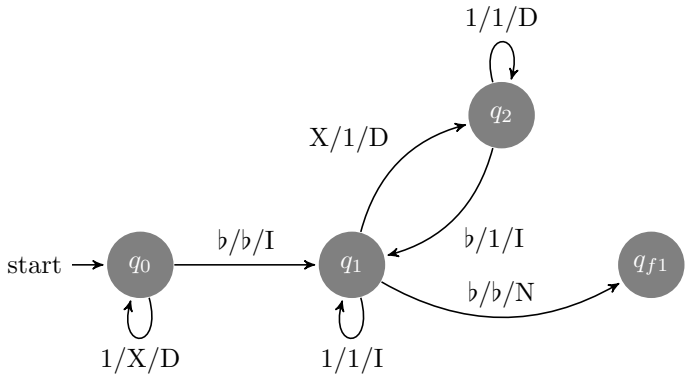
Una máquina de Turing T con alfabeto de entrada Σ puede computar una función f cuyo dominio es un subconjunto de Σ^*. Para cada cadena x en el dominio de f, siempre que T empiece en la configuración inicial correspondiente a x, T se detiene eventualmente con la cadena de salida $f(x)$ en la cinta. También se suelen referir a estas máquinas como *traductoras*.

Definición 2.1.6 (Cómputo de una función por una MTE)**.** Sean $T = (Q, F, \Sigma, \Gamma, q_0, \delta)$ una MTE y f una función parcial en Σ^* con valores en Γ^*, $f : \Sigma^* \longrightarrow \Gamma^*$. Se afirma que T computa a f si y sólo si para cada $x \in \Sigma^*$ con la cual f está definida se tiene que:

$$(q_0, x) \vdash^* (p, f(x))$$

en donde $p \in F$ y tanto q_0 como p están apuntando al primer símbolo de x y $f(x)$ respectivamente.

Ejemplo 2. Consideremos T_1 una MTE que computa la función $f(1^n) = 1^{2n}$ y cuyo diagrama de estados es:



Dado que el dominio de esta función es 1^* entonces tomemos $w_1 = 11$ para ilustrar el efecto de T_1 sobre w_1.

$$
\begin{array}{lllllllllll}
(q_0, \underline{1}1) & \vdash & & & & & & & & & \\
& \vdash & (q_0, x\underline{1}) & \vdash & (q_0, xx\underline{\flat}) & \vdash & (q_1, x\underline{x}\flat) & \vdash & (q_2, x1\underline{\flat}) & \vdash & \\
& \vdash & (q_1, x\underline{1}1) & \vdash & (q_1, \underline{x}11) & \vdash & (q_2, 1\underline{1}1) & \vdash & (q_2, 11\underline{1}) & \vdash & \\
& \vdash & (q_2, 111\underline{\flat}) & \vdash & (q_1, 11\underline{1}1) & \vdash & (q_1, 1\underline{1}11) & \vdash & (q_1, \underline{1}111) & \vdash & \\
& \vdash & (q_1, \underline{\flat}1111) & \vdash & (q_{f1}, \underline{\flat}1111) & & & & & &
\end{array}
$$

Por lo tanto $(q_0, \underline{1}1) \vdash^* (q_{f1}, \underline{\flat}1111)$.

Para cada cadena x de la forma 1^n al final del proceso tendremos sobre la cinta $f(x) = 1^{2n}$. Cuando T_1 reciba como entrada una cadena que no pertenezca al dominio, donde el dominio es Σ^*, la secuencia de pasos de cómputo se bloqueará, ya que no existe algún caso en la función que se pueda aplicar a la configuración que presenta la cinta.

MTE como generadora de lenguajes

La MTE que se utiliza para generar lenguajes, funciona como un proceso inverso al de aceptar un lenguaje. Anteriormente, el objetivo era decidir si una cadena pertenecía o no a un lenguaje. Ahora el objetivo será generar cadenas verificando que cumplan con ciertas reglas, las cuales definen al lenguaje generado.

Para este propósito, la definición de la MTE sufre una ligera modificación ya que el conjunto de estados finales es vacío. El proceso comienza con la configuración inicial en q_0, un conjunto $L = \varnothing$ y la cinta en blanco. Comienza la ejecución y cada vez que la máquina se encuentra en q_0 emite un símbolo para separar las cadenas generadas, es decir, se agrega a L la cadena que se encuentra sobre la cinta y continúa el proceso.

Definición 2.1.7 (Lenguaje generado por una MTE)**.** Sean $T = (Q, F = \varnothing, \Sigma, \Gamma, q_0, \delta)$ una MTE y $L \subseteq \Sigma^*$ un lenguaje. Se afirma que T genera a L si para cada $x = yaz \in L$ en donde $y, z \in (\Gamma \cup \{\flat\})^*$ y $a \in \Gamma \cup \{\flat\}$ se tiene que:

$$(q_0, \underline{\flat}) \vdash^* (q_0, y\underline{a}z)$$

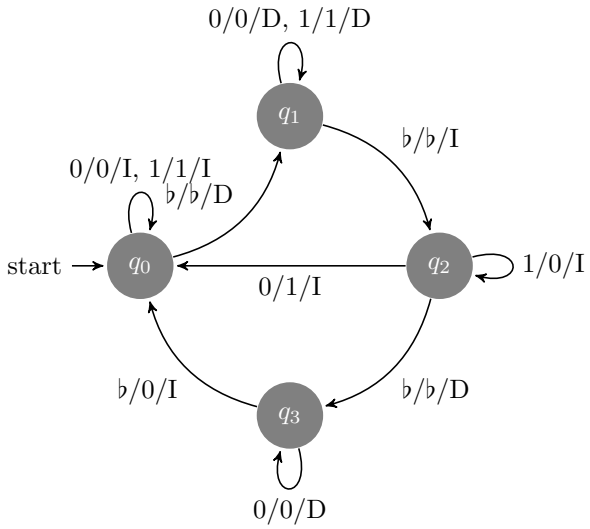
Durante el proceso de generación del lenguaje L, se deben de tomar en cuenta las siguientes consideraciones:

1. Cada vez que regresamos a q_0 tenemos que verificar que la cadena sobre la cinta no esté ya en L para poder agregarla.

2. Probablemente la cadena que generemos contenga *blancos*, entonces tenemos que depurarla quitándole estos blancos antes de agregarla a L.

3. Dado que eventualmente L podría ser infinito según la máquina que se utilice, tenemos que indicarle a la máquina cuándo debe detenerse.

Ejemplo 3. Consideremos T_1 una MTE que genera cadenas sobre el alfabeto $\{0, 1\}$ en orden lexicográfico y cuyo diagrama de estados es:



Se muestra el funcionamiento de T_1 para la generación de las primeras siete cadenas:

$$
\begin{array}{lllllllllll}
(q_0, \underline{\flat}) & \vdash & & & & & & & & & \\
 & \vdash & (q_1, \flat\underline{\flat}) & \vdash & (q_2, \underline{\flat}\flat) & \vdash & (q_3, \flat\underline{\flat}) & \vdash & (q_0, \underline{\mathbf{\flat 0}}) & \vdash & \\
 & \vdash & (q_1, \flat\underline{0}, 1) & \vdash & (q_1, \flat 0\underline{\flat}) & \vdash & (q_2, \flat\underline{0}\flat) & \vdash & (q_0, \underline{\mathbf{\flat 1\flat}}) & \vdash & \\
 & \vdash & (q_1, \flat\underline{1}\flat) & \vdash & (q_1, \flat 1\underline{\flat}) & \vdash & (q_2, \flat\underline{1}\flat) & \vdash & (q_2, \underline{\flat}0\flat) & \vdash & \\
 & \vdash & (q_3, \flat\underline{0}\flat) & \vdash & (q_3, \flat 0\underline{\flat}) & \vdash & (q_0, \underline{\mathbf{\flat 00}}) & \vdash & (q_0, \underline{\flat}00) & \vdash & \\
 & \vdash & (q_1, \flat\underline{0}0) & \vdash & (q_1, \flat 0\underline{0}) & \vdash & (q_1, \flat 00\underline{\flat}) & \vdash & (q_2, \flat 0\underline{0}\flat) & \vdash & \\
 & \vdash & (q_0, \underline{\mathbf{\flat 01\flat}}) & \vdash & (q_0, \underline{\flat}01\flat) & \vdash & (q_1, \flat\underline{0}1\flat) & \vdash & (q_1, \flat 0\underline{1}\flat) & \vdash & \\
 & \vdash & (q_1, \flat 01\underline{\flat}) & \vdash & (q_2, \flat 0\underline{1}\flat) & \vdash & (q_2, \flat\underline{0}0\flat) & \vdash & (q_0, \underline{\mathbf{\flat\flat 10\flat}}) & \vdash & \\
 & \vdash & (q_1, \flat\underline{1}0\flat) & \vdash & (q_1, \flat 1\underline{0}\flat) & \vdash & (q_1, \flat 10\underline{\flat}) & \vdash & (q_2, \flat 1\underline{0}\flat) & \vdash & \\
 & \vdash & (q_0, \underline{\mathbf{\flat 11\flat}}) & \vdash & (q_0, \underline{\flat}11\flat) & \vdash & (q_1, \flat\underline{1}1\flat) & \vdash & (q_1, \flat 1\underline{1}\flat) & \vdash & \\
 & \vdash & (q_1, \flat 11\underline{\flat}) & \vdash & (q_2, \flat 1\underline{1}\flat) & \vdash & (q_2, \flat\underline{1}0\flat) & \vdash & (q_2, \underline{\flat}00\flat) & \vdash & \\
 & \vdash & (q_3, \flat\underline{0}0\flat) & \vdash & (q_3, \flat 0\underline{0}\flat) & \vdash & (q_3, \flat 00\underline{\flat}) & \vdash & (q_0, \underline{\mathbf{\flat 00\flat}}) & \vdash & \\
 & \vdash & (q_0, \flat\underline{0}00) & \vdash & (q_0, \underline{\flat}000) & \vdash & ... & & & &
\end{array}
$$

Por lo que $L = \{0, 1, 00, 01, 10, 11, 000, ...\}$. Nosotros sólo ejecutamos el proceso para generar las primeras siete cadenas de L, ya que es claro que este lenguaje es infinito.

En la siguiente sección se hace mención de algunas variantes del modelo estándar como son: máquina de Turing *multipista*, máquina de Turing *multicinta* y máquina de Turing *no determinista*.

2.1.2. Variantes de la MTE

Existen numerosas variantes en las cuales se cumplen convenciones levemente distintas de configuración inicial, movimientos permisibles, protocolos seguidos para aceptar cadenas, etc. Además el modelo básico o estándar puede mejorarse de diversas maneras. Una máquina de Turing de manejo más fácil podría ser una con cintas adicionales o simplemente dividir la cinta en múltiples pistas. Por ejemplo, el manejo del *no determinismo* también es de gran ayuda ya que permite explorar varios caminos a la vez y es un modelo de gran importancia en la Teoría de Complejidad. Sin embargo, no se obtiene cambio alguno en el poder del cómputo final, ya que todo lo que se puede hacer con estas variantes, también se puede hacer con el modelo estándar.

Máquina de Turing multipista (MTMP)

En el modelo *MTMP* la cinta está dividida en un número finito de pistas. De nueva cuenta la cabeza apunta a una posición en la cinta, con la diferencia de que ahora en esa posición se están leyendo tantos símbolos como pistas tengamos en la cinta, de modo que en un sólo paso podemos reemplazar más de un símbolo. En general, la definición de la máquina queda igual excepto por la función de transición δ y las configuraciones de la cinta:

- La función de transición δ se define como sigue:

$$\delta : Q \times (\Gamma \cup \{\flat\})^n \longrightarrow Q \times (\Gamma \cup \{\flat\})^n \times \{I, N, D\} \; n \geq 1$$

De modo que:

$$\delta(q,(a_0,a_1,...,a_n)) = (p,(b_0,b_1,...,b_n),M)$$

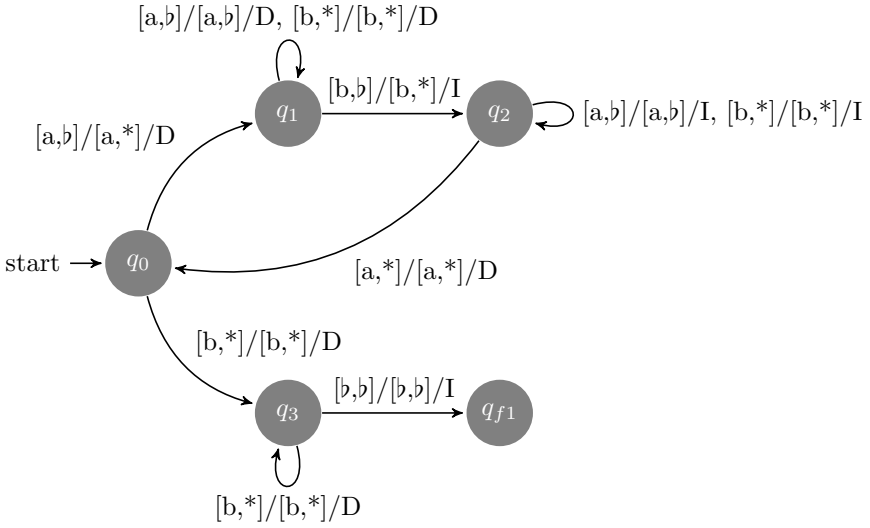
en donde a_i es el símbolo que se lee en la $i-$ ésima pista, b_i es el símbolo por el cual será reemplazado, mientras que M indica la dirección en la que se moverá la cabeza.

- La configuración de la cinta sigue siendo un par, sólo que ahora el segundo elemento es una lista en la cual cada elemento es el contenido de una pista, es decir, la lista tiene tantas cadenas como pistas tengamos. Entonces:

$$(q,[x_0\underline{a_0}y_0,x_1\underline{a_1}y_1,...,x_n\underline{a_n}y_n])$$

en donde $x_i, y_i \in (\Gamma \cup \{\flat\})^*$ y $a_i \in (\Gamma \cup \{\flat\})$, dado que se tiene una sola cabeza, estos símbolos están "alineados". Los elementos de la cinta aparecen ordenados, siendo el primer elemento de la lista el contenido de la primera pista y así sucesivamente.

Ejemplo 4. Consideremos T_1 una $MTMP$ con dos pistas que acepta cadenas de la forma $a^n b^n$ con $n \geq 1$ y cuyo diagrama de estados es:



Veamos cómo funciona esta variante simulando la ejecución del proceso de aceptación de T_1 sobre la cadena $w_1 = aabb$, la cual será colocada en la primera pista. Dado que T_1 trabaja con dos pistas, nuestra configuración inicial es $(q_0, [\underline{a}abb, \underline{\flat}\flat\flat\flat])$:

$$
\begin{array}{lllll}
(q_0, [\underline{a}abb, \underline{\flat}\flat\flat\flat]) & \vdash & & & \\
& \vdash & (q_1, [a\underline{a}bb, *\underline{\flat}\flat\flat]) & \vdash & (q_1, [aa\underline{b}b, *\flat\underline{\flat}\flat]) \quad \vdash \\
& \vdash & (q_2, [a\underline{a}bb, *\underline{\flat} * \flat]) & \vdash & (q_2, [\underline{a}abb, \underline{*}\flat * \flat]) \quad \vdash \\
& \vdash & (q_0, [a\underline{a}bb, *\underline{\flat} * \flat]) & \vdash & (q_1, [aa\underline{b}b, * * \underline{*}\flat]) \quad \vdash \\
& \vdash & (q_1, [aab\underline{b}, * * *\underline{\flat}]) & \vdash & (q_2, [aa\underline{b}b, * * \underline{*}*]) \quad \vdash \\
& \vdash & (q_2, [a\underline{a}bb, *\underline{*} * *]) & \vdash & (q_0, [aa\underline{b}b, * * \underline{*}*]) \quad \vdash \\
& \vdash & (q_3, [aab\underline{b}, * * *\underline{*}]) & \vdash & (q_3, [aabb\underline{\flat}, * * * * \underline{\flat}]) \quad \vdash \\
& \vdash & (q_{f1}, [aab\underline{b}\flat, * * *\underline{*}\flat]) & &
\end{array}
$$

Como podemos ver, la segunda pista nos sirve como *memoria auxiliar* y cada vez que apareamos una a con una b en la primera pista, escribimos el símbolo $*$ en la segunda pista. Dado que las pistas se encuentran alineadas, el símbolo $*$ será escrito en la segunda pista, en las posiciones correspondientes de los símbolos a y b que fueron apareados, mientras que la cadena original colocada en la primera pista no sufre cambio alguno.

Máquina de Turing multicinta (MTMC)

En esta variante la máquina cuenta con n cintas diferentes y completamente independientes, en cada una de las cintas la unidad de control cambia el contenido de la casilla revisada y luego realiza el desplazamiento correspondiente, todo en un sólo paso computacional. Nuevamente, el cambio significativo lo sufre la función de transición, la cual ahora se define como sigue:

$$\delta : Q \times (\Gamma \cup \{\flat\})^n \longrightarrow Q \times ((\Gamma \cup \{\flat\}) \times \{I, N, D\})^n$$

La sintáxis de las transiciones se muestra a continuación:

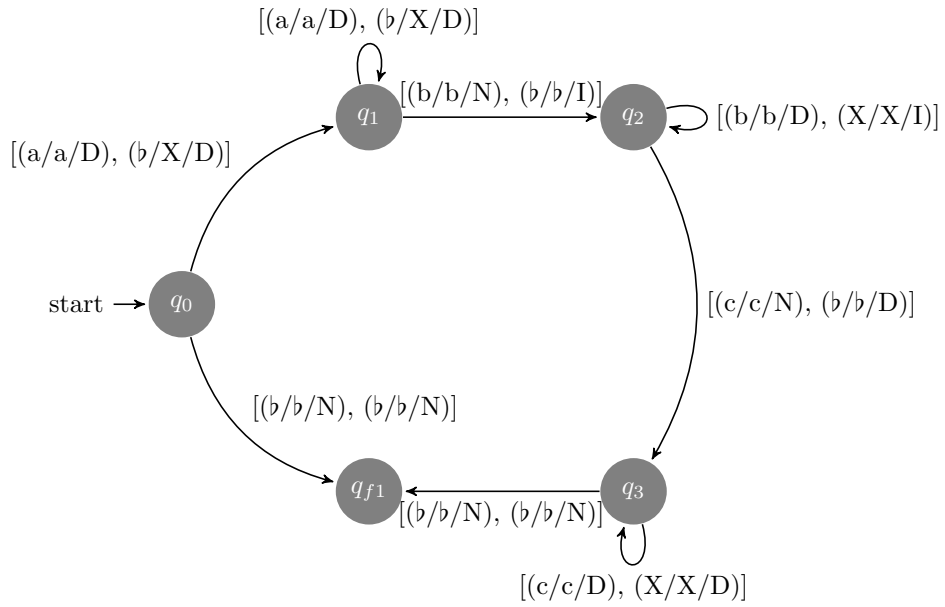
$$\delta(q, (a_0, a_1, ..., a_{n-1})) = (p, [(b_0, M_0), (b_1, M_1), ..., (b_n, M_{n-1})])$$

mientras que las configuraciones adquieren un aspecto similar al del modelo $MTMP$:

$$(q, [w_0, w_1, ..., w_{n-1}])$$

en donde w_i se encuentra alojada en la i - ésima cinta y en cada cadena hay una marca indicando el símbolo que está siendo leído por la unidad de control. De nueva cuenta la primera cinta se usa para la cadena de entrada, las demás se usan como espacio de trabajo auxiliar y se hace caso omiso de su contenido cuando la máquina se detiene.

Ejemplo 5. Veamos el siguiente diagrama de T_1 una $MTMC$ que acepta el lenguaje $L = \{a^n b^n c^n : n \geq 0\}$ y que trabaja con dos cintas:



Veamos cómo funciona esta variante simulando la ejecución del proceso de aceptación de T_1 sobre la cadena $w_1 = aabbcc$. Dado que T_1 trabaja con dos cintas partimos de la configuración $(q_0, [\underline{a}abbcc, \underline{\flat}\flat\flat\flat\flat])$:

$$
\begin{array}{lcccccc}
(q_0, [\underline{a}abbcc, \underline{\flat}\flat\flat\flat\flat]) & \vdash & & & & & \\
& \vdash & (q_1, [a\underline{a}bbcc, X\underline{\flat}\flat\flat\flat]) & \vdash & (q_1, [aa\underline{b}bcc, XX\underline{\flat}\flat\flat]) & \vdash & \\
& \vdash & (q_2, [aa\underline{b}bcc, X\underline{X}\flat\flat\flat]) & \vdash & (q_2, [aab\underline{b}cc, \underline{X}X\flat\flat\flat]) & \vdash & \\
& \vdash & (q_2, [aabb\underline{c}c, \underline{\flat}XX\flat\flat\flat]) & \vdash & (q_3, [aabb\underline{c}c, \flat\underline{X}X\flat\flat\flat]) & \vdash & \\
& \vdash & (q_3, [aabbc\underline{c}, \flat X\underline{X}\flat\flat\flat]) & \vdash & (q_3, [aabbcc\underline{\flat}, \flat XX\underline{\flat}\flat\flat]) & \vdash & \\
& \vdash & (q_{f1}, [aabbcc\underline{\flat}, \flat XX\underline{\flat}\flat\flat]) & & & &
\end{array}
$$

Esta variante es importante ya que es la base para la siguiente y última variante del modelo estándar, que es la introducción del no determinismo.

Máquina de Turing no determinista (MTND)

Una $MTND$ se define exactamente igual que una MTE y como en las variantes anteriores sólo se hace distinción en la función de transición δ, ya que ahora los valores de la imagen son conjuntos de ternas de la forma (edo, sim, M):

$$\delta : Q \times (\Gamma \cup \{\flat\})^n \longrightarrow \wp(Q \times (\Gamma \cup \{\flat\}) \times \{I, N, D\})$$

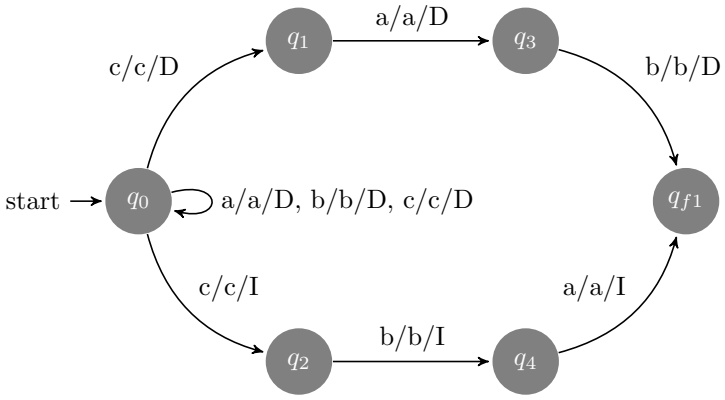
De modo que las transiciones toman el siguiente aspecto:

$$\delta(q, a) = \{(p_0, b_0, M_0), (p_1, b_1, M_1), ..., (p_{n-1}, b_{n-1}, M_{n-1})\}$$

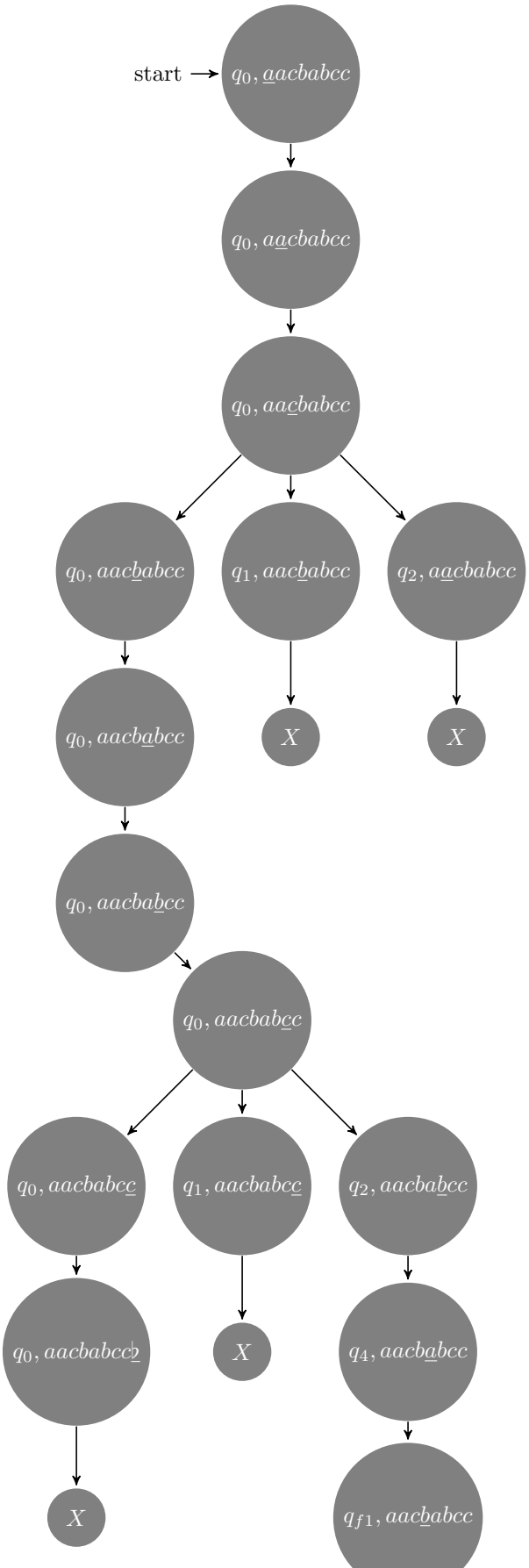
El proceso empieza con la configuración inicial (q_0, w) en donde $w \in \Sigma^*$. Cuando se presenta *no determinismo* se abren nuevas ramas (una por cada posible movimiento). Para poder inspeccionar todos los caminos el proceso deja de ser lineal y en cada paso se realiza sólo un movimiento por rama. Cuando una rama presenta una configuración en la cual ya no es posible continuar, esta se cierra y sólo se observarán aquellas que queden abiertas. Por lo tanto w es *aceptada* si existe por lo menos un cómputo en alguna de las ramas que presente en su configuración un *estado final*.

Ejemplo 6. Veamos el siguiente diagrama de T_1 una $MTND$ que acepta el lenguaje $L = \{w : w$ contiene la subcadena cab o la subcadena $abc\}$. Esta máquina contiene un no determinismo cuando el estado en la configuración de la cinta es q_0 y el símbolo que se está leyendo es c. Este escenario provoca la apertura de tres ramas y en cada una hay que ejecutar el movimiento correspondiente:

- Reemplazamos el símbolo que está siendo leído por el símbolo c, movemos la cabeza de la cinta al sector derecho y permanecemos en el estado q_0.
- Reemplazamos el símbolo que está siendo leído por el símbolo c, movemos la cabeza de la cinta al sector derecho y nos cambiamos al estado q_1.
- Reemplazamos el símbolo que está síendo leido por el símbolo c, movemos la cabeza de la cinta al sector izquierdo y nos cambiamos al estado q_2.



Veamos cómo funciona T_1 con la cadena $w = aacbabcc$ mediante un diagrama de secuencia. Numerando los niveles desde cero, vemos que en el tercer nivel tenemos tres ramas ya que el nivel dos presentó el caso $\delta(q_0, c)$. Cuando seguimos con el proceso a partir del tercer nivel las ramas uno, dos y tres presentan los casos $\delta(q_0, b)$, $\delta(q_1, b)$ y $\delta(q_2, a)$ respectivamente. Sólo el caso que presenta la primera rama se encuentra definido en δ y es por eso que en el cuarto nivel la segunda y la tercera rama se cierran. En el sexto nivel se vuelve a presentar no deternimismo y de nueva cuenta se abren tres ramas. Al final del proceso, en el noveno nivel, la única rama abierta y que presenta el estado final es la tercera y por lo tanto la cadena w es aceptada por T_1:


start
$q_0, \underline{a}acbabcc$
$q_0, a\underline{a}cbabcc$
$q_0, aa\underline{c}babcc$
$q_0, aac\underline{b}abcc$
$q_1, aac\underline{b}abcc$
$q_2, a\underline{a}cbabcc$
$q_0, aacb\underline{a}bcc$
X
X
$q_0, aacba\underline{b}cc$
$q_0, aacbab\underline{c}c$
$q_0, aacbabc\underline{c}$
$q_1, aacbabc\underline{c}$
$q_2, aacba\underline{b}cc$
$q_0, aacbabcc\underline{\flat}$
X
$q_4, aacb\underline{a}bcc$
X
$q_{f1}, aac\underline{b}abcc$

Con esto concluimos los fundamentos de la máquina de Turing y pasamos a los temas de Haskell necesarios para comprender las implementaciones que se realizan en el transcurso de este trabajo.

2.2 Haskell

Haskell es un lenguaje funcional puro, de propósito general, que incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional, como son las funciones de orden superior, evaluación perezosa, tipos polimórficos estáticos, tipos definidos por el usuario, casa de patrones, y listas por comprensión. Además, incorpora características interesantes como el tratamiento sistemático de la sobrecarga, la facilidad en la definición de tipos abstractos de datos, el sistema de entrada/salida puramente funcional y la posibilidad de utilizar módulos.

En la implementación de los módulos explotaremos estas características, de modo que creemos conveniente incluir un repaso de Haskell orientado hacia las necesidades de este trabajo. Para ampliar el panorama de este lenguaje se recomiendan los libros "Programming in Haskell"[4] y "Haskell 98 Languaje and Libraries: The Revised Report"[5], en los cuales está basada esta sección.

2.2.1. Tipos en Haskell

Haskell es un lenguaje fuertemente tipado basado en la regla de aplicación:

$$f : A \longrightarrow B,\ a : A \ / \therefore f\ a : B.$$

Esta regla está emparentada con el modus ponens de la lógica matemática.

A grandes rasgos, los temas que hay que saber de Haskell para entender las implementaciones desarrolladas en este trabajo, son:

1. Tipos elementales: Char (caracteres), Bool (booleanos), Int (enteros), Float (reales), etc.
2. Tipos estructurados: tuplas, listas, funciones, etc.
3. Tipos definidos por el programador.

Tipos elementales

Los valores de tipo *Bool* representan expresiones lógicas cuyo resultado puede ser $True$ ó $False$. Las principales funciones y los principales operadores para este tipo son:

```
&& :: Bool -> Bool -> Bool conjunción lógica.
```

```
|  :: Bool -> Bool -> Bool disyunción lógica.

not :: Bool -> Bool negación lógica.

otherwise :: Bool función constante que devuelve el valor True.
```

Entre los tipos numéricos se encuentran *Int*, *Integer*, *Float* y *Double*. Podemos distiguir algunas funciones y algunos operadores para estos valores:

```
+ :: Int -> Int -> Int operación suma entre enteros.

- :: Int -> Int -> Int operación resta entre enteros.

* :: Int -> Int -> Int operación producto entre enteros.

+ :: Float -> Float -> Float operación suma entre reales.

- :: Float -> Float -> Float operación resta entre reales.

* :: Float -> Float -> Float operación producto entre reales.

div :: Int -> Int -> Int operación cociente.

mod :: Int -> Int -> Int operación residuo.

even :: Int -> Bool comprueba la naturaleza par de un entero.

odd :: Int -> Bool comprueba la naturaleza impar de un entero.
```

Los operadores y funciones disponibles para *Int*, también lo están para *Integer*. En realidad la principal diferencia radica en que la capacidad de representación de *Int* es más limitada que la de *Integer*. Veamos la ejecución de las siguientes instrucciones en HASKELL:

```
Hugs> 2^31::Int
-2147483648

Hugs> 2^32::Integer
4294967296

Hugs> 2^31::Integer
2147483648
```

Existe gran variedad de operaciones predefinidas como son *abs*, *log*, *exp*, *sqrt*, *sin*, *cos*, *tan*, etc., que pueden encontrar en el API de HASKELL.

El tipo *Char* representa un carácter, ya sea un dígito, una letra, un signo, etc., y se escribe entre comillas simples, por ejemplo 'a' representa el carácter con la letra a. Algunas funciones que trabajan con caracteres son:

```
ord :: Char -> Int devuelve el código ASCII.

chr :: Int -> Char inversa de ord.

isUpper :: Char -> Bool indica si el carácter es mayúscula.

isLower :: Char -> Bool indica si el carácter es minúscula.

isDigit :: Char -> Bool indica si el carácter es dígito.
```

Los aperadores de igualdad y orden de HASKELL son:

```
> mayor que.

< menor que.

== igual a.

>= mayor o igual que.

<= menor o igual que.

/= distinto de.
```

Para poder utilizar estos operadores los argumentos deben tener el mismo tipo. Pero se pueden utilizar con números, símbolos, booleanos, etc.

Definición 2.2.1 (Currificación)**.** La currificación consiste en simular funciones de varios argumentos mediante funciones de orden superior de un argumento.

Ejemplo 7. La función f_c toma un entero x y devuelve una función que cuando toma un entero devuelve $2 * x + y$

```
fc :: Int -> (Int -> Int)
fc = \x -> (\y -> 2 * x + y)

Entonces:

fc 8 3 = 2 * 8 + 3 = 19
```

En realidad, la precedencia de las declaraciones de tipos en HASKELL hace innecesarios los paréntesis de la declaración anterior y la notación *lambda*[1] puede simplificarse mediante encaje de patrones. Dicha definición puede reescribirse como:

```
fc' :: Int -> Int -> Int
fc' x y = 2 * x + y
```

[1]Para mayor información sobre expresiones lambda se recomienda el libro "Programming in Haskell"[4]

```
Entonces:

fc' 8 3 = 2 * 8 + 3 = 19
```

Las definiciones mediante currificación son muy habituales en HASKELL ya que además de evitar paréntesis innecesarios, facilitan la aplicación parcial de funciones. De esa forma, la expresión $f_c\ 3$ tiene significado por sí misma:

```
(fc 8) 3 = (2 * 8) + 3 = 19
```

Ya podemos definir una función sencilla que calcule el factorial de un número entero cuya firma es:

```
factorial :: Int -> Int
```

el cuerpo de la función $factorial$ se puede implementar de varias forma, la primera es usar la instrucción de control if cuya sintaxis es:

$$if\ bandera :: Bool\ then\ inst_1 :: Int\ else\ inst_2 :: Int$$

La implementación completa para esta opción es:

```
factorial :: Int -> Int
factorial n = if (n == 0) then 1 else n*(factorial (n -  1))
```

Una segunda opción es utilizar lo que se conoce como *comparación de patrones* con lo cual es posible definir una función dando más de una ecuación para ésta. Cada ecuación define el comportamiento de la función para distintas formas del argumento, y los patrones permiten capturar dicha forma. Al aplicar la función a un parámetro concreto, la comparación de patrones determina la ecuación a utilizar a partir de sus argumentos. El orden de las ecuaciones es importante ya que la casa de patrones se hace de manera secuencial. Entonces, la implementación utilizando esta técnica es la siguiente:

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n*(factorial (n - 1))
```

También es posible realizar una comprobación de patrones en cualquier punto de una expresión utilizando la instrucción *case*, cuya sintaxis es:

```
case expresion of
     patron1 -> resultado1
     patron2 -> resultado2
     ... -> ...
     patronn -> resultadon
```

La definición de la función factorial con la instrucción *case* se muestra a continuación:

```
factorial :: Int -> Int
factorial n = case n of
     0 -> 1
     n -> n * (factorial (n - 1))
```

Tipos estructurados

Estos son constructores de tipos predefinidos que permiten representar colecciones de objetos. En esta sección veremos tuplas y listas.

Una tupla es un dato compuesto en donde el tipo de cada componente puede ser distinto. De modo que si v_1, v_2, ... , v_n son valores con tipos t_1, t_2, ... , t_n entonces (v_1, v_2, ... , v_n) es una tupla con tipo (t_1, t_2, ... , t_n). Algunos ejemplos son:

```
(Integer, Bool)
(Double, Double)
(Integer, Integer, Integer)
```

Una lista es una colección de cero o más elementos del mismo tipo. Hay dos constructores:

1. [ ] representa la lista vacía.
2. : permite añadir un elemento al principio de la lista.

De modo que si v_1, v_2, ... , v_n son valores con tipo t, entonces $v_1 : (v_2 : (...(v_{n-1} : (v_n : [])))$ es una lista con tipo $[t]$.

En HASKELL las cadenas de caracteres en realidad son listas de caracteres, y pueden declararse por medio de $[Char]$ o $String$. Existe gran variedad de funciones predefinidas que nos permiten manipular listas; aquí mencionamos algunas que adquieren importancia especial, ya que son las que utilizamos en nuestras implementaciones:

1. (!!) :: $[a] \longrightarrow Int \longrightarrow a$, regresa el elemento de la lista que se encuentra en la posición indicada, comenzando desde la posición cero. Por ejemplo:

   ```
   ['a', 'b', 'c'] !! 2 = 'c'
   ```

2. $reverse$ :: $[a] \longrightarrow [a]$, invierte una lista. Por ejemplo:

   ```
   reverse [1, 2, 3] = [3, 2, 1]
   ```

3. $take$ :: $Int \longrightarrow [a] \longrightarrow [a]$, regresa los primeros n elementos de la lista. Por ejemplo:

   ```
   take 4 "hola a todos" = "hola"
   ```

4. $length$ :: $[a] \longrightarrow Int$, regresa la longitud de una lista. Por ejemplo:

   ```
   length [1, 2, 3] = 3
   ```

5. $elem$ :: $a \longrightarrow [a] \longrightarrow Bool$, devuelve $True$ en caso de que el elemento pertenezca a la lista y $False$ en caso de que no. Por ejemplo:

```
elem 't' ['a', 'f', 'r', 'h', 't'] = True
```

6. *replicate* :: $Int \longrightarrow a \longrightarrow [a]$, crea una lista con n copias del elemento indicado. Por ejemplo:

```
replicate 5 'c' = "ccccc"
```

7. *drop* :: $Int \longrightarrow [a] \longrightarrow [a]$, devuelve la lista quitándole los primeros n elementos. Por ejemplo:

```
drop 2 [11, 1, 1985, 22, 8, 2007] = [1985, 22, 8, 2007]
```

8. ++ :: $[a] \longrightarrow [a] \longrightarrow [a]$, toma dos listas y devuelve una lista que resulta de la concatenación de estas listas. Por ejemplo:

```
['1', '2', '3'] ++ ['a', 'b', 'c', 'd'] = "123abcd"
```

9. *concat* :: $[[a]] \longrightarrow [a]$, toma una lista de listas y devuelve una lista que resulta de la concatenación de estas listas. Por ejemplo:

```
concat [[1,3,5],[7,9,11],[13,15,17]] = [1,3,5,7,9,11,13,15,17]
```

10. *map* :: $(a \longrightarrow b) \longrightarrow [a] \longrightarrow [b]$, toma una función y una lista, devuelve una lista que resulta de aplicarle la función a cada uno de los elementos de la lista original. Por ejemplo:

```
map (++"con") ["yo ", "tu "] = ["yo con", "tu con"]
```

11. *head* :: $[a] \longrightarrow a$, regresa la cabeza o primer elemento de la lista y en caso de que ésta esté vacía, nos devolverá un mensaje de error. Por ejemplo:

```
head [11, 1, 1985, 22, 8, 2007] = 11
```

12. *tail* :: $[a] \longrightarrow [a]$, regresa la cola de la lista. Por ejemplo:

```
tail [11, 1, 1985, 22, 8, 2007] = [1, 1985, 22, 8, 2007]
```

Las enumeraciones también juegan un papel importante en este trabajo y se utilizan para definir listas en HASKELL. Éstas poseen un conjunto de operaciones que subyacen bajo la sintaxis de las secuencias aritméticas. Por ejemplo:

```
take 10 [1,3..] = [1,3,5,7,9,11,13,15,17,19]

[1..15] = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Como podemos ver es simplemente una sintaxis más cómoda para expresar secuencias. Esto incluye no sólo la mayoría de los tipos numéricos, sino que también al tipo *Char*. Por ejemplo:

```
['a'..'z'] = "abcdefghijklmnopqrstuvwxyz"

['A'..'M'] = "ABCDEFGHIJKLM"
```

Como ya se había mencionado, las listas juegan un papel de suma importancia en las implementaciones de todos los módulos de este trabajo, en particular nos referimos a lo fácil que es manipular listas en HASKELL y al uso de *listas por comprensión*.

Las listas por comprensión nos permiten definir listas mediante una notación similar a la utilizada para escribir conjuntos por comprensión en matemáticas. Por ejemplo, $P = \{x \in \mathbb{N} \mid x \text{ es par } \}$ denota el conjunto de número naturales que son pares. Entonces, en HASKELL podemos escribir:

$$P = [x \mid x \longleftarrow [0..], even\ x]$$

siendo [0..] la lista de números naturales. La notación de listas por comprensión permite declarar de forma concisa una gran cantidad de iteraciones sobre listas. Esta notación está adaptada de la teoría de conjuntos de Zermelo - Fraenkel[2]. El formato básico de la definición de una lista por comprensión es:

$$[< expr > \mid < calif_1 >, < calif_2 >, ..., < calif_n >]$$

en donde $< cualif_i >$ es el i - ésimo cualificador, es decir, es la i - ésima condición con la cual tienen que cumplir los elementos que pertenecen al conjunto (lista) que queremos generar. Lo que hace posible este tipo de descripciones, es la evaluación que utiliza HASKELL conocida como evaluación perezosa, la cual sólo calcula el valor de las expresiones cuando necesita hacer uso de éste valor. Por ejemplo, en HASKELL tiene sentido la siguiente instrucción:

$$take\ 5\ P$$

obteniendo como resultado $[0, 2, 4, 6, 8]$, mientras que en otros lenguajes esta operación sería inútil ya que primero se intentaría evaluar P por completo y una vez que termine entonces sí tomaría los primeros 5 elementos, pero es claro que esto nunca va a suceder ya que P es un conjunto infinito y ni siquiera lo podríamos generar. Nuevamente reiteramos la importancia de esta característica, porque más adelante vamos a proporcionar funciones que nos devuelven el lenguajes, los cuales eventualmente podrían ser infinitos, pero que gracias a las listas por comprensión y a la evaluación perezosa, vamos a poder observarlos e incluso manipularlos.

Por último, para cerrar el tema de listas por comprensión vamos a definir una función

[2]Matemáticos creadores de una axiomática de la teoría de conjuntos

klns que nos genere la *estrella de Kleene*[3]. Sólo para recordar, la estrella de Kleene L^* para un lenguaje L se define de manera recursiva como se muestra a continuación:

$L_0 = \{\varepsilon\}$ donde ε es la cadena vacía.

$L_{i+1} = \{vw : v \in L \text{ and } w \in L_i \text{ donde } i \geq 0\}$.

$L^* = \bigcup_{i \in \mathbb{N}} L_i = \{\varepsilon\} \cup L_1 \cup L_2 \cup L_3 \cup \ldots$.

Como es de suponerse, necesitamos implementar tres casos, contemplando cada una de las posibilidades de la definición recursiva. Los primeros dos se agrupan en una función llamada *kln*, mientras que el tercero es la generalización en la cual hacemos uso de las listas por comprensión. Y aunque parece algo complicado, seguramente la implementación al final terminará siendo bastante intuitiva e incluso un poco obvia, ya que si recorremos la definición recursiva a la par junto con la implementación, veremos que sólo basta respetar el orden en el cuál se definieron los casos. En esta implementación *kln s n* construye a L_n. La implementación queda como se ve a continuación:

```
kln :: Alfabeto -> Int -> [Cadena]
kln s 0 = [" "]
kln s (n+1) = [a:w | a <- s , w <- kln s n]

klns :: Alfabeto -> [Cadena]
klns s = concat [kln s n | n <- [0..] ]
```

En el último caso, estamos generando las listas por longitud tomando $n = 0, 1, 2, \ldots$. y posteriormente utilizamos la función *concat* para concatenar todas las listas de las diferentes longitudes.

Tipos definidos por el programador

Haskell permite renombrar y definir tipos a través de las instrucciones *type* y *data*. La sintaxis de cada instrucción se muestra a continuación:

- *type* $Nombre\ e_1$ ... $e_n = expresion_T$, donde:
  - $Nombre$ es el nombre de un nuevo constructor de *tipo* de aridad $n \geq 0$.
  - e_1, ..., e_n son variables de *tipo* diferentes que representan los argumentos de $Nombre$.
  - $expresion_T$ es una expresión de *tipo* que sólo utiliza como variables de *tipo* las variables e_1, ..., e_n.

  type permite renombrar tipos (no crea nuevos tipos). Como ejemplo para este tipo de declaraciones tenemos:

[3]Los conceptos de la Teoría de la Computación no definidos pueden ser consultados en los libros "An Introduction to Formal Languages and Automata"[1], "Introduction to the Theory of Computation"[2] y "Lenguajes Formales y Teoría de la Computación"[3]

```
type Nombre = String
type Edad = Integer
type Persona = (Nombre, Edad)
type Punto = (Float,Float)
```

- *data* $Nombre = Nom$ { $e_1 :: T_1$, ..., $e_n :: T_n$ }, donde:
  - $Nombre$ es el nombre de un nuevo tipo de dato.
  - Nom es el nombre con el cual nos referiremos al nuevo tipo de dato.
  - e_i es una expresión de tipo T_i para $i = 1, ..., n$. La expresión e_i constituye el i - ésimo elemento de $Nombre$.

  data: crea nuevos tipos de datos mediante el uso de constructores. En esta instrucción tenemos la libertad de nombrar cada uno de los campos siempre y cuando los tipos que les asociemos ya existan:

  ```
  data Persona = Pers {nombre::Nombre, edad::Edad}
  data Punto = Punto {x::Float, y::Float}
  data Color = Rojo | Verde | Azul
  ```

  En el último ejemplo se utiliza el operador |, que funciona como *or*, es decir, estamos indicando que *Color* puede tomar sólo los valores *Rojo*, $Verde$ ó *Azul*.

Expresiones de tipo $IO()$

El tipo IO permite hacer uso de las operaciones de entrada y salida en HASKELL. Una expresión de tipo $IO()$ denota una acción. Las operaciones de entrada y salida que se utilizan en este trabajo son las siguientes:

```
putStrLn :: String -> IO()  imprime una cadena y un salto de línea.

getLine :: IO() -> String   lee una cadena de caracteres hasta que
                            encuentra un salto de línea.

writeFile :: String -> String -> IO()  toma como argumentos el nombre
                                       de un archivo y una cadena; es-
                                       cribe dicha cadena en el archi-
                                       vo correspondiente. En caso de
                                       que el archivo no exista, este
                                       se crea autómaticamente, si e-
                                       xiste, se borra el contenido
                                       del mismo antes de escribir la
                                       cadena en él.

appendFile :: String -> String -> IO()  toma como argumentos el nombre
                                        de un archivo y una cadena; a-
                                        ñade dicha cadena al final del
                                        archivo.
```

3 Modelo estándar de la máquina de Turing

En este capítulo vamos a implementar el funcionamiento del modelo estándar de la máquina de Turing en HASKELL. En especial nos interesan tres funcionalidades: lenguaje aceptado, lenguaje traducido o cómputo de una función y lenguaje generado por una máquina de Turing estándar. Para llevar a cabo esta tarea primero modificamos ligeramente las definiciones sobre la máquina de Turing estándar dadas en el capítulo anterior, con el fin de resaltar las bondades de HASKELL cuando trabajamos con modelos matemáticos a partir de una teoría adecuada.

Consideremos entonces la siguiente definición:

Definición 3.0.2 (Máquina de Turing Estándar (MTE))**.** Una máquina de Turing estándar es un par (mte, δ) en donde mte contempla todos los elementos básicos de la máquina mientras que δ es la función que determina su comportamiento. Ahora bien:

I. mte = (Q, q_0, q_f, Σ, Γ), en donde cada componente se define como sigue:

   1) Q es un conjunto *finito* de estados tal que $Q \neq \varnothing$.
   2) q_0 es el estado inicial tal que $q_0 \in Q$.
   3) q_f es el estado final *de aceptación* tal que $q_f \in Q$.
   4) Σ es el alfabeto de entrada.
   5) Γ es el alfabeto de la cinta, que incluye a Σ, es decir, $\Sigma \subseteq \Gamma$

II. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{I, N, D\}$ en donde:

   1) El símbolo *blanco* es parte de Γ y se denota mediante el símbolo $\flat$.
   2) $\{I, N, D\}$ es el conjunto de direcciones para las cuales se puede mover la cabeza de la cinta.

A continuación vamos construir en HASKELL el tipo MTE asociado con la definición anterior. Los tipos básicos de la máquina son:

```
type Simbolo = Char
type Estado = [Char]
type Alfabeto = [Simbolo]
```

Tenemos que definir un tipo para el movimiento que pueda tomar los valores I, N ó D, entonces:

```
data Movimiento = I | D | N deriving (Show, Eq)
```

La instrucción *deriving* (*Show*, *Eq*) se incluye para que los datos de este tipo puedan mostrarse en pantalla y pueda decidirse si dos elementos de este tipo son iguales o no. Ahora sí, definimos la función de transición δ de la siguiente manera:

```
type Delta = Estado -> Simbolo -> (Estado, Simbolo, Movimiento)
```

cubriendo todos los elementos de la máquina de Turing para el modelo estándar. Para encapsular todos estos componentes nos apoyamos en un nuevo tipo llamado MTE cuya definición es:

```
type MTE = (MaqT, Delta)
```

en donde:

```
data MaqT = MT {q::[Estado],
                qi::Estado,
                f::Estado,
                s::Alfabeto,
                g::Alfabeto
               } deriving Show
```

para tener acceso a cada uno de los elemento de $MaqT$ se utiliza la notación $Comp\ T$, en donde, $Comp = q$, qi, f, s ó g, y T es de tipo $MaqT$. Por ejemplo, al estado final de T se accede con la instrucción $f\ T$. Al incluir la instrucción *deriving Show* en la definición, HASKELL asociará a $MaqT$ un formato para poder presentar los elementos de la máquina.

Ejemplo 8. La máquina T acepta cadenas de ceros y unos con un número par de ceros. La definición en HASKELL de T (considerando $T = mte$) se muestra a continuación:

```
maqt = MT ["q0","q1","q2"]
      "q0"
      "q2"
      ['0','1']
      ['0','1','_']

mte = (maqt, del) where
             del "q0" '1' = ("q0",'1',D)
             del "q0" '0' = ("q1",'0',D)
             del "q1" '1' = ("q1",'1',D)
             del "q1" '0' = ("q0",'0',D)
             del "q0" '_' = ("q2",'_',N)
```

El formato que HASKELL le asocia a *maqt* es el siguiente:

```
Main> maqt
MT {q = ["q0","q1","q2"], qi = "q0", f = "q2", s = "01", g = "01_"}
```

Para mostrar T de manera conjunta, resulta conveniente definir nuestro propio formato. Una opción puede ser por medio de las siguientes funciones:

```
pintaEstados :: [Estado] -> String
pintaEstados [] = ""
pintaEstados (l:le) = l ++ "  " ++ pintaEstados le

pintaEstado :: Estado -> String
pintaEstado e = e ++ ""

pintaAlfabeto :: Alfabeto -> String
pintaAlfabeto [] = ""
pintaAlfabeto (l:la) = [l] ++ " " ++ pintaAlfabeto la

generaPares :: [Estado] -> [Simbolo] -> [(Estado,Simbolo)]
generaPares le ls = [(e,s)| e <- le, s <- ls]


pintaDeltaAux :: Delta -> [(Estado,Simbolo)] -> String
pintaDeltaAux _ [] = "\n"
pintaDeltaAux t ((e, s):xs) = case t e s of
              ("qr",s',m) -> pintaDeltaAux t xs
              (e',s',m) -> "                          d "
                        ++ e ++ " " ++ [s] ++ " = " ++ " " ++e'
                        ++ " " ++ [s'] ++ " " ++ show m ++ "\n"
                        ++ pintaDeltaAux t xs

pintaDelta :: [Estado] -> [Simbolo] -> Delta -> String
pintaDelta le ls t = pintaDeltaAux t (generaPares le ls)

instance Show (MaqT, Delta) where
  show (m, d) = "\nEstados:: "
                    ++ pintaEstados (q (m)) ++ "\n" ++
                "\nEstado Inicial:: "
                    ++ pintaEstado (qi (m)) ++ "\n" ++
                "\nEstado Final:: "
                    ++ pintaEstado (f (m)) ++ "\n" ++
                "\nAlfabeto de Entrada:: "
                    ++ pintaAlfabeto (s (m)) ++ "\n" ++
                "\nAlfabeto de la Cinta:: "
                    ++ pintaAlfabeto (g (m)) ++ "\n" ++
                "\nFuncion de Transicion::\n"
                    ++ pintaDelta (q (m)) (g (m)) d
```

Con esto la presentación de T en pantalla es:

```
Main> mte
```

```
Estados:: q0  q1  q2

Estado Inicial:: q0

Estado Final:: q2

Alfabeto de Entrada:: 0 1

Alfabeto de la Cinta:: 0 1 _

Funcion de Transicion::
                        d q0 0 =  q1 0 D
                        d q0 1 =  q0 1 D
                        d q0 _ =  q2 _ N
                        d q1 0 =  q0 0 D
                        d q1 1 =  q1 1 D
```

Usamos listas por comprensión para generar todas la posibles combinaciones de la forma (*Estado*, *Simbolo*) de *mte* para explorar todos los casos de la función *del*. Además, utilizamos la instrucción *instance* para definir el ejemplar de la clase *Show* para la MTE.

Ya podemos definir en HASKELL la máquina que acepta el lenguaje $L(mte) = \{w \in \{a,b\}^* \mid w = a^n b^n$ en donde $n \geq 0\}$ expuesta en el ejemplo 1 [Capítulo 2, pág. 11]:

```
mt = MT ["q0","q1","q2","q3","qf"]
         "q0"
         "qf"
        ['a','b']
        ['a','b','X','Y','_']

mte = (mt,d) where
       d "q0" 'a' = ("q1",'X',D)
       d "q1" 'a' = ("q1",'a',D)
       d "q1" 'Y' = ("q1",'Y',D)
       d "q1" 'b' = ("q2",'Y',I)
       d "q2" 'Y' = ("q2",'Y',I)
       d "q2" 'a' = ("q2",'a',I)
       d "q2" 'X' = ("q0",'X',D)
       d "q0" 'Y' = ("q3",'Y',D)
       d "q3" 'Y' = ("q3",'Y',D)
       d "q3" '_' = ("qf",'_',D)
       d _ _ = ("qr",'E',N)
```

Hay que recordar que el carácter que representa al guión bajo es el equivalente al símbolo $\flat$ y observar también que se ha sustituido la etiqueta del estado final de la máquina, reemplazando la etiquta original q_4 por la *única etiqueta* que nosostros reconoceremos como estado final

"qf". Como podemos ver, la función d se definió por patrones y se complementó con la transición:

```
d _ _ = ("qr",'E',N)
```

En este caso se utiliza para convertir a δ en una función total, con la cual podamos decidir en todo momento qué hacer sin importar qué argumentos se reciban. Dado que la comparación de patrones se hace de manera secuencial y esta transición ha sido colocada al final de la definición, no estamos afectando el funcionamiento de δ. Sin embargo, agregar este caso es necesario, ya que finalmente nos encontraremos con parámetros para los cuales no tenemos asociado un patrón en la definición original de la función. Por ejemplo, si recibieramos ("q3", a), dado que no tenemos un caso en δ explícito que corresponda a este par, automáticamente entraríamos a la transición complemento en d obteniendo una configuración con el estado "qr", lo cual indica que la cadena de entrada *no es aceptada por mte*. En realidad, "qr" es tratado como un estado de rechazo mientras que 'E' indica que se ha producido un "error" al intentar ejecutar otro cómputo. Por último debemos de tener claro que no se pierde generalidad ya que la única forma de llegar al estado final de *mte* es a través de las transiciones definidas en δ.

Definición 3.0.3 (Configuración de la cinta de una MTE)**.** Una configuración es una expresión de la forma:

$$(q, w, n)$$

en donde q es estado actual, w es la cadena que se encuentra sobre la cinta tal que $w \in \Gamma^*$ y n indica la posición de la cabeza sobre w, es decir, la cabeza esta apuntando al n-ésimo símbolo de w, numerando w de izquierda a derecha y asociándole al primer símbolo la posición cero, igual que lo hace HASKELL.

En relación a la definición 2.1.2 [Capítulo 2, pág. 9], la expresión (q, w, n) corresponde al par $(q, a_0a_1...\underline{a_n}a_{n+1}...a_k)$ en donde $w = a_0a_1...a_na_{n+1}...a_k$.

Si consideramos $\Gamma = \{\flat, a, b, c\}$, entonces la configuración $(q, w, 5)$ con $w = abba\flat cab$ significa que estamos en el estado q y la cabeza de la cinta apunta al sexto símbolo de w, en este caso c. De modo que si quisiéramos realizar un cómputo desde esta configuración, tendríamos que buscar en el dominio de nuestra función δ la pareja (q, c) y posteriormente utilizar la imagen para poder hacer las actualizaciones correspondientes, obteniendo una nueva configuración. Para definir el tipo de una configuración en HASKELL, basta con definir los siguientes tipos:

```
type Cadena = [Simbolo]
type Configuracion = (Estado,Cadena,Int)
```

Los movimientos en la cinta se pueden contemplar en nueve casos los cuales quedan determinados por la función de transición.

Definición 3.0.4 (Movimientos en la cinta de una MTE)**.** Los movimientos en la cinta son pasos computacionales que se ejecutan desde alguna configuración. Sea cf_a la configuración actual y supongamos que existe una definición para ella en el dominio de δ; entonces, la nueva configuración cf_s es:

I. Configuración actual $cf_a = (q, b_0b_1b_2...b_{n-1}b_n, n)$

1) Si $\delta(q, b_n) = (p, s, I)$ entonces $cf_s = (p, b_0b_1b_2...b_{n-1}s, n-1)$
2) Si $\delta(q, b_n) = (p, s, N)$ entonces $cf_s = (p, b_0b_1b_2...b_{n-1}s, n)$
3) Si $\delta(q, b_n) = (p, s, D)$ entonces $cf_s = (p, b_0b_1b_2...b_{n-1}s\flat, n+1)$

II. Configuración actual $cf_a = (q, b_0b_1b_2...b_{n-1}b_n, 0)$

1) Si $\delta(q, b_0) = (p, s, I)$ entonces $cf_s = (p, \flat sb_1b_2...b_{n-1}b_n, 0)$
2) Si $\delta(q, b_0) = (p, s, N)$ entonces $cf_s = (p, sb_1b_2...b_{n-1}b_n, 0)$
3) Si $\delta(q, b_0) = (p, s, D)$ entonces $cf_s = (p, sb_1b_2...b_{n-1}b_n, 1)$

III. Configuración actual $cf_a = (q, b_0b_1b_2...b_{i-1}b_ib_{i+1}...b_{n-1}b_n, i)$, con $0 < i < n$:

1) Si $\delta(q, b_i) = (p, s, I)$ entonces $cf_s = (p, b_0b_1b_2...b_{i-1}sb_{i+1}...b_{n-1}b_n, i-1)$
2) Si $\delta(q, b_i) = (p, s, N)$ entonces $cf_s = (p, b_0b_1b_2...b_{i-1}sb_{i+1}...b_{n-1}b_n, i)$
3) Si $\delta(q, b_i) = (p, s, D)$ entonces $cf_s = (p, b_0b_1b_2...b_{i-1}sb_{i+1}...b_{n-1}b_n, i+1)$

La siguiente función toma un símbolo y lo sustituye por el símbolo de la cadena que se encuentra en la posición indicada por el entero y regresamos la cadena modificada:

```
sustituye :: Cadena -> Int -> Simbolo -> Cadena
sustituye [] _ _ = []
sustituye (w:ws) 0 a = a:ws
sustituye (w:ws) (n+1) a = w: sustituye ws n a
```

La implementación de la función δ se hace de manera directa e incluso se respeta el orden de los nueve casos o patrones mostrados en la definición anterior. Entonces:

```
delta :: Delta -> Configuracion -> Configuracion
delta d (e,w,n)
      |(n==v && n/=0) = case d e (w!!n) of
              (e',a,I) -> (e',sustituye w n a,n-1)
              (e',a,N) -> (e',sustituye w n a,n)
              (e',a,D) -> (e',(sustituye w n a)++"_",n+1)

      |(n==0) = case d e (w!!n) of
              (e',a,I) -> (e','_':sustituye w n a,0)
              (e',a,N) -> (e',sustituye w n a,0)
              (e',a,D) -> if n==v
                          then (e', (sustituye w n a)++"_",1)
                          else (e',sustituye w n a,n+1)
```

```
|otherwise = case d e (w!!n) of
        (e',a,I) -> (e',sustituye w n a,n-1)
        (e',a,N) -> (e',sustituye w n a,n)
        (e',a,D) -> (e',sustituye w n a,n+1)

where v = length w - 1
```

3.1 Lenguaje aceptado por una máquina de Turing estándar

El objetivo de esta sección es implementar la función:

```
lenguajeAceptado :: MTE -> [Cadena],
```

la cual, como su nombre lo indica, recibe una máquina de Turing estándar y nos devuelve el lenguaje aceptado por la misma. Para llevar acabo esta tarea, vamos a explotar la técnica de evaluación de HASKELL y vamos también a apoyarnos de lleno en las listas por comprensión, con el objeto de simplicar de manera substancial el trabajo. Pero antes, tenemos que definir la función de transición extendida δ^*:

```
deltaEstrella :: MTE -> Configuracion -> Bool
```

La función consiste en aplicar δ de manera recursiva hasta llegar al estado final o hasta llegar al estado qr; en realidad es la operación $\vdash^*$ aplicando la función tantas veces como sea necesario hasta obtener una respuesta de aceptación a partir de una configuración. Observemos que lo que necesitamos, suponiendo a cf_i como configuración inicial, es lo siguiente:

```
deltaEstrella mte cf_i = true syss cf_i |-* (q_f, ...)
deltaEstrella mte cf_i = false syss cf_i |-* (q_r, ...)
```

siendo q_f el estado final de mte. De modo que la implementación queda como se ve a continuación:

```
deltaEstrella :: MTE -> Configuracion -> Bool
deltaEstrella (mt,d) ("qf", w, n) = True
deltaEstrella (mt,d) ("qr", w, n) = False
deltaEstrella (mt,d) (q, w, n) =
               deltaEstrella (mt,d) (delta d (q,w,n))
```

También necesitamos ajustar la definición 2.1.4 [Capítulo 2, pág. 9] del leguaje aceptado por una MTE que utilice configuraciones de la forma (q, w, n).

Definición 3.1.1 (Lenguaje aceptado por una MTE)**.** Sea $MT = (mt, \delta)$ una MTE tal que $mt = (Q, q_0, q_f, \Sigma, \Gamma)$ y $w \in \Sigma^*$; decimos que w es *aceptada* por MT si y sólo si partiendo de la configuración inicial $cf_i = (q_0, w, 0)$ finalmente llegamos a una configuración con el estado q_f. Esto es:

$$cf_i \vdash^* (q_f, xaz, n)$$

en donde $x, z \in \Gamma^*$, $a \in \Gamma$ y n indica la posición de a en la cadena xaz. Definimos el *lenguaje aceptado* por MT como el conjunto $L(MT)$ de cadenas de entrada que acepta MT.

Veamos cómo se comporta la máquina T_1 que acepta al lenguaje $L(T_1) = \{a^n b^n \mid n \geq 0\}$ del ejemplo 1 [Capítulo 2, pág. 10], al procesar la cadena $w_1 = aabb$:

Ejecución de la cadena $w_1 = aabb$:

$$\begin{array}{lllllllll} (q_0, aabb, 0) & \vdash & & & & & & & \\ & \vdash & (q_1, Xabb, 1) & \vdash & (q_1, Xabb, 2) & \vdash & (q_2, XaYb, 1) & \vdash & \\ & \vdash & (q_2, XaYb, 0) & \vdash & (q_0, XaYb, 1) & \vdash & (q_1, XXYb, 2) & \vdash & \\ & \vdash & (q_1, XXYb, 3) & \vdash & (q_2, XXYY, 2) & \vdash & (q_2, XXYY, 1) & \vdash & \\ & \vdash & (q_0, XXYY, 2) & \vdash & (q_3, XXYY, 3) & \vdash & (q_3, XXYY\flat, 4) & \vdash & \\ & \vdash & (q_4, XXYY\flat\flat, 5) & & & & & & \end{array}$$

$\therefore (q_0, aabb, 0) \vdash^*$ ***true***

Si intentamos lo mismo con la cadena *aabbab* llegaríamos a un punto en el que no podríamos seguir con el cómputo de la cadena que se encuentra en la cinta y acabaríamos en una configuración con el estado "qr", obteniendo una respuesta de rechazo para la cadena *aabbab*. Tomando en cuenta que al procesar una cadena w siempre partimos de la configuración $(qi\ mt, w, 0)$ como configuración inicial, entonces una implementación adecuada de una función que determine si una cadena es aceptada o no por una MTE es la siguiente:

```
aceptaCadena :: MTE -> Cadena -> Bool
aceptaCadena (mt,d) c = deltaEstrella (mt,d) (qi mt, c, 0)
```

Para probar estas funciones tenemos que cargar en HASKELL los archivos "ModeloMTE.hs" y "EjemplosModeloMTELA.hs" tal y como se indica en el Apéndice B, en donde se muestran las especificaciones sobre el uso de las funciones contenidas en los módulos programados en este trabajo. Una vez cargados los archivos, ejecutamos las siguientes instrucciones:

```
Main> aceptaCadena mteAnBn "aabb"
True
Main> aceptaCadena mteAnBn "baabb"
False
Main> aceptaCadena mteAnBn "aaaaaabbbbbb"
True
Main> aceptaCadena mteAnBn "abababababab"
False
```

Se concluye esta sección con la implementación de la función objetivo *lenguajeAceptado* la cual sólo tiene que llamar a la función *aceptaCadena* para cada una de las cadenas generadas por la estrella de *Kleene*. La importancia de la evaluación perezosa se puede observar en la instrucción *klns* (*s mt*), ya que, realizar esto en un lenguaje que no utilice este tipo de evaluación, no resultaría tan sencillo, teniendo que implementar funciones para manipular el resultado, que es de donde obtenemos nuestras cadenas que posteriormente serán analizadas con la función *aceptaCadena* en la instrucción *aceptaCadena* (mt, d) *c*. La implementación de la función *lenguajeAceptado* se muestra a continuación:

```
lenguajeAceptado :: MTE -> [Cadena]
lenguajeAceptado (mt,d) =
        [c|c <- klns (s mt), aceptaCadena (mt,d) c]
```

La desventaja de esta implementación es que no sabemos qué pasó durante el análisis del efecto de la máquina sobre la cadena, es decir, hasta este momento es como una caja negra para nosotros, ya que sólo podemos decir dos de los estados que se observaron en el proceso y que son: el estado inicial y alguno de los estados que nos permiten decidir si la cadena fue aceptada o no, q_f o q_r. Para observar paso a paso cada uno de los estados de la máquina durante el proceso de aceptación se implementa la función *pintaDeltaEstrella*. Esta implementación queda como se muestra a continuación:

```
pintaDeltaEstrella :: MTE -> Configuracion -> IO()
pintaDeltaEstrella (mt,d) ("qf",w,n) =
     putStrLn ("|-"++(take n w)++"("++"qf"++")"++(drop n w))
     >> putStrLn("|-True")
pintaDeltaEstrella (mt,d) ("qr",w,n) =
     putStrLn ("|-"++(take n w)++"("++"qr"++")"++(drop n w))
     >> putStrLn("|-False")
pintaDeltaEstrella (mt,d) (q,w,n) = putStrLn "|-"++(take n w)
     ++"("++q++")"++(drop n w) ) >> pintaDeltaEstrella (mt,d)
     (delta d (q,w,n))

pintaProcesoAceptacion :: MTE -> Cadena -> IO()
pintaProcesoAceptacion (mt,d) c =
        pintaDeltaEstrella (mt,d) ((qi mt),c,0)
```

No es necesario explicar a detalle estas funciones ya que *pintaDeltaEstrella* y *pintaProceso Aceptacion* son completamente análogas a las funciones *deltaEstrella* y *aceptaCadena* respectivamente. La única diferencia es que ahora estamos pintando todos los pasos intermedios del proceso. En el siguiente ejemplo se muestra la salida de la función *pintaProcesoAceptacion* para la cadena *aabb*:

```
Main> pintaProcesoAceptacion mte "aabb"
      |-(q0)aabb
      |-X(q1)abb
      |-Xa(q1)bb
```

```
|-X(q2)aYb
|-(q2)XaYb
|-X(q0)aYb
|-XX(q1)Yb
|-XXY(q1)b
|-XX(q2)YY
|-X(q2)XYY
|-XX(q0)YY
|-XXY(q3)Y
|-XXYY(q3)_
|-XXYY_(qf)_
|-True
```

Cabe resaltar que decidimos pintar las configuraciones poniendo el estado en la posición relacionada con el símbolo que está siendo leído para que sea más sencillo seguir el proceso y no utilizamos la notación original (q, w, n). Por ejemplo, en el segundo paso, estamos en el estado "q1" leyendo el símbolo a, entonces, se tiene que aplicar la siguiente transición:

```
d "q1" 'a' = ("q1",'a',D).
```

Posteriormente continuamos con el proceso hasta obtener alguna respuesta.

3.2 Cómputo de una función por una máquina de Turing estándar

En esta sección implementamos el cómputo de una función por una MTE. En algunos textos, se refieren a este tema como "lenguaje traducido por una máquina de Turing estándar". Pero antes de entrar de lleno en la implementación veamos la definición formal de este concepto.

Definición 3.2.1 (Lenguaje traducido o cómputo de una función por una máquina de Turing estándar)**.** Sea $MT = (mt, \delta)$ una MTE tal que $mt = (Q, q_0, q_f, \Sigma, \Gamma)$ y f una función parcial en Σ^* con valores en Γ^*. Se afirma que MT computa a f si para cada $w \in \Sigma^*$ para la cual f está definida, se tiene que:

$$(q_0, w, 0) \vdash^* (q_f, f(w), 0)$$

y MT no acepta a ninguna otra $w \in \Sigma^*$.

El comportamiento de la máquina T_1 que computa la función $f(1^n) = 1^{2n}$ definida en el ejemplo 2 [Capítulo 2, pág. 11] sobre la cadena $w_1 = 11$ se muestra a continuación:

Ejecución sobre la cadena $w_1 = 11$:

$$
\begin{array}{lllllllll}
(q_0, 11, 0) & \vdash & & & & & & & \\
 & \vdash & (q_0, x1, 1) & \vdash & (q_0, xx\flat, 2) & \vdash & (q_1, xx\flat, 1) & \vdash & \\
 & \vdash & (q_2, x1\flat, 2) & \vdash & (q_1, x11, 1) & \vdash & (q_1, x11, 0) & \vdash & \\
 & \vdash & (q_2, 111, 1) & \vdash & (q_2, 111, 2) & \vdash & (q_2, 111\flat, 3) & \vdash & \\
 & \vdash & (q_1, 1111, 2) & \vdash & (q_1, 1111, 1) & \vdash & (q_1, 1111, 0) & \vdash & \\
 & \vdash & (q_1, \flat 1111, 0) & \vdash & (q_{f1}, \flat 1111, 0) & & & &
\end{array}
$$

$\therefore f(11) \vdash^* \mathbf{1111}$

La primera función que vamos a implementar es:

```
deltaEstrellaT :: MTE -> Configuracion -> Cadena
```

la cual es análoga a *deltaEstrella*, sólo que ahora nos devuelve la cadena que se encuentra en la cinta después de que termina el proceso. Se distinguen tres casos:

- Cuando llegamos al estado final devolvemos la cadena que está en la cinta y el proceso termina.
- El hecho de que lleguemos a "qr" indica que el cómputo quedó bloqueado en algún punto. Esto significa que la cadena de entrada no pertenece al dominio de δ y lo indicamos devolviendo la cadena "Dominio Invalido".
- En el tercer caso ejecutamos un movimiento aplicando *delta* a la configuración (q, w, n); el resultado de este movimiento es la entrada del paso recursivo.

La función queda implementada como se ve a continuación:

```
deltaEstrellaT :: MTE -> Configuracion -> Cadena
deltaEstrellaT (mt,d) ("qf",w,n) = (take n w) ++ (drop n w)
deltaEstrellaT (mt,d) ("qr",w,n) = "Dominio Invalido"
deltaEstrellaT (mt,d) (q,w,n) =
               deltaEstrellaT (mt,d) (delta d (q,w,n))
```

La única forma de que una cadena c pertenezca al dominio de la función f es que la podamos procesar por completo partiendo de la configuración (q_0, c, 0) y llegar al estado final al terminar el cómputo, como se contempla en el primer caso de *deltaEstrellaT*. Para implementar la función que computa un lenguaje *lenguajeTraducido* tenemos que generar las cadenas de entrada para *deltaEstrellaT*, después verificar si pertenecen al dominio de la función y en caso de ser así generamos el par (c,$f(c)$). La implementación de la función *lenguajeTraducido* es la siguiente:

```
lenguajeTraducido :: MTE -> [(Cadena,Cadena)]
lenguajeTraducido (mt,d) =
  [(c, deltaEstrellaT (mt,d) (qi mt, c, 0))| c <- klns (s mt),
    deltaEstrellaT (mt,d) (qi mt, c, 0) /= "Dominio Invalido"]
```

La definición en HASKELL para la máquina T_1 que computa la función $f(1^n) = 1^{2n}$ del ejemplo 2 [Capítulo 2, pág. 11] es la siguiente:

```
mtF2n = MT ["q0","qf","q1","q2"]
    "q0"
    "qf"
    ['1']
    ['_','1','x']

mteF2n = (mtF2n,d1) where
    d1 "q0" '1' = ("q0",'x',D)
    d1 "q0" '_' = ("q1",'_',I)
    d1 "q1" 'x' = ("q2",'1',D)
    d1 "q2" '1' = ("q2",'1',D)
    d1 "q2" '_' = ("q1",'1',I)
    d1 "q1" '1' = ("q1",'1',I)
    d1 "q1" '_' = ("qf",'_',N)
    d1 _ _ = ("qr",'E',N)
```

Probemos la función pidiendo los primeros diez pares del lenguaje traducido de esta máquina cuya definición la podemos encontrar en el archivo "EjemplosModeloMTELT.hs":

```
Main> take 10 (lenguajeTraducido mteF2n)
      [("1","_11"),
      ("11","_1111"),
      ("111","_111111"),
      ("1111","_11111111"),
      ("11111","_1111111111"),
      ("111111","_111111111111"),
      ("1111111","_11111111111111"),
      ("11111111","_1111111111111111"),
      ("111111111","_11111111111111111"),
      ("1111111111","_1111111111111111111")]
```

Las funciones encargadas de pintar cada uno de los pasos del proceso son *pintaDelta EstrellaT* y *pintaProcesoTraduccion*:

```
pintaDeltaEstrellaT :: MTE -> Configuracion -> IO()
pintaDeltaEstrellaT (mt,d) ("qf",w,n) =
   putStrLn ("|-"++(take n w)++"("++"qf"++")"++(drop n w))
pintaDeltaEstrellaT (mt,d) ("qr",w,n) =
   putStrLn ("|-"++(take n w)++"("++"qr"++")"++(drop n w))
             >> putStrLn("|-Dominio Invalido")
pintaDeltaEstrellaT (mt,d) (q,w,n) =
   putStrLn ("|-"++(take n w)++"("++q++")"++(drop n w))
           >> pintaDeltaEstrellaT (mt,d) (delta d (q,w,n))
```

```
pintaProcesoTraduccion :: MTE -> Cadena -> IO()
pintaProcesoTraduccion (mt,d) c =
          pintaDeltaEstrellaT (mt,d) (qi mt, c, 0)
```

Por último se muestra cómo trabajan estas funciones mediante la ejecución de la máquina sobre la cadena "11":

```
Main> pintaProcesoTraduccion mteF2n "11"
      |-(q0)11
      |-x(q0)1
      |-xx(q0)_
      |-x(q1)x_
      |-x1(q2)_
      |-x(q1)11
      |-(q1)x11
      |-1(q2)11
      |-11(q2)1
      |-111(q2)_
      |-11(q1)11
      |-1(q1)111
      |-(q1)1111
      |-(q1)_1111
      |-(qf)_1111
```

3.3 Lenguaje generado por una máquina de Turing estándar

El objetivo de esta sección es implementar la función:

```
lenguajeGenerado :: MTE -> [Cadena]
```

que nos devuelve el lenguaje generado por una máquina de Turing. Este proceso comienza con un lenguage $L = \varnothing$, la configuración inicial en q_0 y la cinta en blanco. Cada vez que regresemos a q_0 la cadena que se encuentra sobre la cinta es agregada a L. Para implementar esta función tenemos que considerar la definición que se muestra a continuación:

Definición 3.3.1 (Lenguaje generado por una máquina de Turing estándar)**.** Sea $MT = (mt, \delta)$ una MTE tal que $mt = (Q, q_0, \Sigma, \Gamma)$ y $L \subseteq \Sigma^*$. Se afirma que MT genera a L si para cada $w \in L$ se tiene que:

$$(q_0, \flat, 0) \vdash^* (q_0, w, t)\ t \in \mathbb{N}$$

Tomemos la máquina T_1 que genera cadenas sobre el alfabeto $\{0, 1\}$ en orden lexicográfico del ejemplo 3 [Capítulo 2, pág. 13] y el conjunto L_1 el cual se encuentra inicialmente vacío

y veamos cómo funciona.

Cómputo de T_1:

$$
\begin{array}{lllllllll}
(q_0,\flat,0) & \vdash & & & & & & & \\
 & \vdash & (q_1,\flat\flat,1) & \vdash & (q_2,\flat\flat,0) & \vdash & (q_3,\flat\flat,1) & \vdash & \\
 & \vdash & \mathbf{(q_0,\flat 0,0)} & \vdash & (q_1,\flat 0,1) & \vdash & (q_1,\flat 0\flat,2) & \vdash & \\
 & \vdash & (q_2,\flat 0\flat,1) & \vdash & \mathbf{(q_0,\flat 1\flat,0)} & \vdash & (q_1,\flat 1\flat,1) & \vdash & \\
 & \vdash & (q_1,\flat 1\flat,2) & \vdash & (q_2,\flat 1\flat,1) & \vdash & (q_2,\flat 0\flat,0) & \vdash & \\
 & \vdash & (q_3,\flat 0\flat,1) & \vdash & (q_3,\flat 0\flat,2) & \vdash & \mathbf{(q_0,\flat 00,1)} & \vdash & \\
 & \vdash & (q_0,\flat 00,0) & \vdash & (q_1,\flat 00,1) & \vdash & (q_1,\flat 00,2) & \vdash & \\
 & \vdash & (q_1,\flat 00\flat,3) & \vdash & (q_2,\flat 00\flat,2) & \vdash & \mathbf{(q_0,\flat 01\flat,1)} & \vdash & \\
 & \vdash & (q_0,\flat 01\flat,0) & \vdash & (q_1,\flat 01\flat,1) & \vdash & (q_1,\flat 01\flat,2) & \vdash & \\
 & \vdash & (q_1,\flat 01\flat,3) & \vdash & (q_2,\flat 01\flat,2) & \vdash & (q_2,\flat 00\flat,1) & \vdash & \\
 & \vdash & \mathbf{(q_0,\flat 10\flat,0)} & \vdash & (q_1,\flat 10\flat,1) & \vdash & (q_1,\flat 10\flat,2) & \vdash & \\
 & \vdash & (q_1,\flat 10\flat,3) & \vdash & (q_2,\flat 10\flat,2) & \vdash & \mathbf{(q_0,\flat 11\flat,1)} & \vdash & \\
 & \vdash & (q_0,\flat 11\flat,0) & \vdash & (q_1,\flat 11\flat,1) & \vdash & (q_1,\flat 11\flat,2) & \vdash & \\
 & \vdash & (q_1,\flat 11\flat,3) & \vdash & (q_2,\flat 11\flat,2) & \vdash & (q_2,\flat 10\flat,1) & \vdash & \\
 & \vdash & (q_2,\flat 00\flat,0) & \vdash & (q_3,\flat 00\flat,1) & \vdash & (q_3,\flat 00\flat,2) & \vdash & \\
 & \vdash & (q_3,\flat 00\flat,3) & \vdash & \mathbf{(q_0,\flat 000,2)} & \vdash & (q_0,\flat 000,1) & \vdash & \\
 & \vdash & (q_0,\flat 000,0) & & & & & &
\end{array}
$$

Después de este proceso tenemos que:

$$L_1 = \{0, 1, 00, 01, 10, 11, 000\}$$

Primero necesitamos una función análoga a *deltaEstrella* en la cual nos apoyaremos para implementar la función objetivo de esta sección. Esta función es *deltaEstrellaG* y consta de tres casos:

- Cuando llegamos al estado de rechazo devolvemos [ ] ya que cuando observamos el estado "qr" significa que hemos terminado de generar el lenguaje reconocido por la máquina (siempre y cuando éste no sea infinito).
- En el segundo caso, por definición agregamos la cadena a la lista ya que se observa el estado inicial ("q0") pero tenemos que continuar con el proceso de generación de cadenas con la instrucción $(++\ deltaEstrellaG\ (mt, d)\ (delta\ d\ (q0, w, n)))$, es decir, sin limpiar la cinta.
- En el último caso sólo continuamos con el proceso sin agregar algo a la lista.

La implementación de la función *deltaEstrellaG* es la siguiente:

```
deltaEstrellaG :: MTE -> Configuracion -> [Cadena]
deltaEstrellaG (mt,d) ("qr",w,n) = []
deltaEstrellaG (mt,d) ("q0",w,n) =
             [(take (n + 1) w )++(drop (n + 1) w)]  ++
             deltaEstrellaG (mt,d) (delta d ("q0",w,n))
deltaEstrellaG (mt,d) (q,w,n) =
             deltaEstrellaG (mt,d) (delta d (q,w,n))
```

Sabemos por definición que la configuración inicial para generar un lenguaje con una máquina de Turing estándar es $(q_0, \flat, 0)$, por lo que la implementación de la función *lenguaje Generado* se vuelve bastante intuitiva quedando como se muestra a continuación:

```
lenguajeGenerado :: MTE -> [Cadena]
lenguajeGenerado (mt,d) =
         [c|c <- deltaEstrellaG (mt,d) (qi mt,"_",0)]
```

La definición de T_1 en HASKELL es la siguiente:

```
mtOrdLex = MT ["q0","q1","q2","q3"]
    "q0"
    ""
    ['0','1']
    ['_','0','1']

mteOrdLex = (mtOrdLex,d1) where
d1 "q0" '0' = ("q0",'0',I)
d1 "q0" '1' = ("q0",'1',I)
d1 "q0" '_' = ("q1",'_',D)
d1 "q1" '0' = ("q1",'0',D)
d1 "q1" '1' = ("q1",'1',D)
d1 "q1" '_' = ("q2",'_',I)
d1 "q2" '1' = ("q2",'0',I)
d1 "q2" '0' = ("q0",'1',I)
d1 "q2" '_' = ("q3",'_',D)
d1 "q3" '0' = ("q3",'0',D)
d1 "q3" '_' = ("q0",'0',I)
d1 _ _ = ("qr",'E',N)
```

Para probar esta función utilizamos la máquina *mteOrdLex* definida en el archivo "EjemplosModeloMTELG.hs":

```
Main> take 13 (lenguajeGenerado mteOrdLex)
[
"_","_0","_1_","_00","_00","_01_","_01_",
"_10_","_11_","_11_","_000","_000","_000"
]
```

El resultado contiene cadenas repetidas. Una opción para depurar la lista, puede ser por medio de la siguiente función:

```
quitaCadenasRepetidas :: [Cadena] -> [Cadena]
quitaCadenasRepetidas [] = []
quitaCadenasRepetidas (w:ws) = if elem w ws
              then quitaCadenasRepetidas ws
              else w:quitaCadenasRepetidas ws
```

De manera que ahora podemos pedir los elementos del lenguaje generado con la siguiente instrucción:

```
Main> quitaCadenasRepetidas(take 13 (lenguajeGenerado mteOrdLex))
      [
      "_","_0","_1_","_00","_01_","_10_","_11_","_000"
      ]
```

Se concluye este capítulo resaltando que las implementaciones de las funciones *lenguaje Aceptado*, *lenguajeTraducido* y *lenguajeGenerado* fueron prácticamente inmediatas haciendo uso de los elementos que proporciona HASKELL mencionados en el segundo capítulo.

4 Variantes de la máquina de Turing estándar

En este capítulo cubriremos la implementación de las variantes de la máquina de Turing estándar mencionadas en el capítulo 2 [pág. 14]. Como vimos en ese capítulo, las diferencias entre el modelo estándar y cualquiera de las variantes se ve reflejado en la definición y manejo de la cinta, lo que provoca que tengamos que redefinir las configuraciones y la función de transición para cada variante, tanto en la teoría como en la implementación.

Estas diferencias nos obligan a alterar las funciones que manejan los conceptos de configuración y función de transición. Sin embargo, son alteraciones menores y la idea de fondo es la misma que para el modelo estándar. Además de la implementación de estos cambios tendremos que implementar algunas funciones sencillas que nos servirán de apoyo en el desarrollo de los módulos de este capítulo. Cada módulo corresponde a una variante en particular y aunque las variantes tienen muchos aspectos en común, cada una será tratada de manera independiente.

Para poder llevar a cabo la implementación de las variantes presentadas en este capítulo debemos de suponer que siempre estamos partiendo de la implementación del modelo estándar, ya que éste será la base para el desarrollo de cada una de ellas.

4.1 Implementación de la máquina de Turing multipista (MTMP)

Una configuración para esta variante es un par de la forma:

$$(q, [x_0\underline{a_0}y_0, x_1\underline{a_1}y_1, ..., x_m\underline{a_m}y_m])$$

en donde $x_i, y_i \in (\Gamma \cup \{\flat\})^*$, $a_i \in (\Gamma \cup \{\flat\})$ y la representaremos por medio de la terna (q, [$x_0a_0y_0$, $x_1a_1y_1$, ..., $x_ma_my_m$], n), en donde n indica la posición del sector al que apunta la cabeza de la cinta; en este caso, dicho sector está formado por las $a_i's$, las cuales se encuentran alineadas. Mientras que en Haskell definimos de manera directa una configuración para la máquina multipista como sigue:

```
type Configuracion = (Estado,[Cadena],Int)
```

La diferencia con el modelo estándar es el segundo parámetro, ya que ahora manejamos una lista de cadenas, que contiene tantas cadenas como número de pistas en las que se haya dividido la cinta. De modo que la primera cadena de la lista es el contenido de la primera pista, la segunda cadena de la lista es el contenido de la segunda pista y así sucesivamente. Dado que la la función de transición para esta variante toma la forma:

$$\delta(q, (a_0, a_1, ..., a_m)) = (p, (b_0, b_1, ..., b_m), M)$$

las transiciones tienen que manejar una lista de símbolos:

```
type Delta = Estado -> [Simbolo] -> (Estado,[Simbolo],Movimiento)
```

Por comodidad renombraremos el tipo MTE que hacía referencia a máquina de Turing estándar por $MTMP$, cuyas siglas se refieren a la máquina de Turing multipista. Teniendo entonces:

```
type MTMP = (MaqT, Delta)
```

El resto de los tipos se queda igual. Por ejemplo, la definición en HASKELL para la máquina T_1 que acepta cadenas de la forma $a^n b^n$ con $n \geq 1$ del ejemplo 4 [Capítulo 2, pág. 15] es la siguiente:

```
mtAnBn = MT ["q0","qf","q1","q2","q3"]
            "q0"
            "qf"
            ['a','b']
            ['_','a','b','*']

mtmpAnBn = (mtAnBn,d) where
        d "q0" ['a','_'] = ("q1",['a','*'],D)
        d "q0" ['b','*'] = ("q3",['b','*'],D)
        d "q1" ['a','_'] = ("q1",['a','_'],D)
        d "q1" ['b','_'] = ("q2",['b','*'],I)
        d "q1" ['b','*'] = ("q1",['b','*'],D)
        d "q2" ['a','_'] = ("q2",['a','_'],I)
        d "q2" ['a','*'] = ("q0",['a','*'],D)
        d "q2" ['b','*'] = ("q2",['b','*'],I)
        d "q3" ['b','*'] = ("q3",['b','*'],D)
        d "q3" ['_','_'] = ("qf",['_','_'],I)
        d _ _ =("qr",['E','E'],N)
```

A continuación damos una definición por casos para los movimientos en la cinta de una máquina multipista análoga a la que presentamos para el modelo estándar, que maneje las transiciones.

Definición 4.1.1 (Movimientos en la cinta de una máquina de Turing multipista)**.** Sea cf_a la configuración actual de la cinta mientras que cf_s será la nueva configuración después de ejecutar un paso computacional. Entonces:

I. Configuración actual $cf_a = (q, [c_0c_1...c_n, ..., d_0d_1...d_n], n)$

1) Si $\delta(q, [c_n, ..., d_n]) = (p, [a_n, ..., b_n], I)$ entonces:
$$cf_s = (p, [c_0c_1...c_{n-1}a_n, ..., d_0d_1...d_{n-1}b_n], n-1)$$

2) Si $\delta(q, [c_n, ..., d_n]) = (p, [a_n, ..., b_n], N)$ entonces:
$$cf_s = (p, [c_0c_1...c_{n-1}a_n, ..., d_0d_1...d_{n-1}b_n], n)$$

3) Si $\delta(q, [c_n, ..., d_n]) = (p, [a_n, ..., b_n], D)$ entonces:
$$cf_s = (p, [c_0c_1...c_{n-1}a_n\flat, ..., d_0d_1...d_{n-1}b_n\flat], n+1)$$

II. Configuración actual $cf_a = (q, [c_0c_1...c_n, ..., d_0d_1...d_n], 0)$

1) Si $\delta(q, [c_0, ..., d_0]) = (p, [a_0, ..., b_0], I)$ entonces:
$$cf_s = (p, [\flat a_0c_1...c_n, ..., \flat b_0d_1...d_n], 0)$$

2) Si $\delta(q, [c_0, ..., d_0]) = (p, [a_0, ..., b_0], N)$ entonces:
$$cf_s = (p, [a_0c_1...c_n, ..., b_0d_1...d_n], 0)$$

3) Si $\delta(q, [c_0, ..., d_0]) = (p, [a_0, ..., b_0], D)$ entonces:
$$cf_s = (p, [a_0c_1...c_n, ..., b_0d_1...d_n], 1)$$

III. Configuración actual $cf_a = (q, [c_0c_1...c_{i-1}c_ic_{i+1}...c_n, ..., d_0d_1...d_{i-1}d_id_{i+1}...d_n], i)$, con $0 < i < n$:

1) Si $\delta(q, [c_i, ..., d_i]) = (p, [a_i, ..., b_i], I)$ entonces:
$$cf_s = (p, [c_0c_1...c_{i-1}a_ic_{i+1}...c_n, ..., d_0d_1...d_{i-1}b_id_{i+1}...d_n], i-1)$$

2) Si $\delta(q, [c_i, ..., d_i]) = (p, [a_i, ..., b_i], N)$ entonces:
$$cf_s = (p, [c_0c_1...c_{i-1}a_ic_{i+1}...c_n, ..., d_0d_1...d_{i-1}b_id_{i+1}...d_n], i)$$

3) Si $\delta(q, [c_i, ..., d_i]) = (p, [a_i, ..., b_i], D)$ entonces:
$$cf_s = (p, [c_0c_1...c_{i-1}a_ic_{i+1}...c_n, ..., d_0d_1...d_{i-1}b_id_{i+1}...d_n], i+1)$$

Para hacer posible la implementación de la función δ para esta variante en base a esta definición, necesitamos dos funciones más, la función *sustituyeColumna* que ejecuta de manera recursiva la función *sustituye* para reemplazar un símbolo en cada una de las pistas de la cinta y la función *columna* que recibe una configuración y nos devuelve el sector que está siendo observado por la cabeza de la cinta. La implementación de estas funciones es la siguiente:

```
sustituyeColumna :: [Cadena] -> Int -> [Simbolo] -> [Cadena]
sustituyeColumna [] _ _ = []
sustituyeColumna (lw:lws) n (ls:lss) =
               sustituye lw n ls: sustituyeColumna lws n lss

columna :: Configuracion -> [Simbolo]
columna (_, [], _) = []
columna (q, (l:ls), n) = l!!(n): columna (q, ls, n)
```

Implementamos el comportamiento de la función de transición δ con base en la definición anterior y bajo la misma idea que la implementación para el modelo estándar. De hecho, respetamos el mismo orden en que fueron presentados los casos, mientras que la función *deltaEstrella* conserva la misma implementación excepto por el hecho de que ahora recibe una máquina $MTMP$:

```
delta :: Delta -> Configuracion -> Configuracion
delta d (e,ls,n)
|(n==v && n/=0) = case d e (columna (e,ls,n)) of
      (e',le,I) -> (e',sustituyeColumna ls n le,n-1)
      (e',le,N) -> (e',sustituyeColumna ls n le,n)
      (e',le,D) -> (e',sustituyeColumna (map (++"_") ls) n le,n+1)

|(n==0) = case d e (columna (e,ls,n)) of
      (e',le,I) -> (e',sustituyeColumna (map ("_"++) ls) 0 le,0)
(e',le,N) -> (e',sustituyeColumna ls 0 le,0)
(e',le,D) -> if n==v
             then (e',sustituyeColumna (map (++"_") ls) n le,n+1)
             else (e',sustituyeColumna ls n le,n+1)

|otherwise = case d e (columna (e,ls,n)) of
      (e',le,I) -> (e',sustituyeColumna ls n le,n-1)
      (e',le,N) -> (e',sustituyeColumna ls n le,n)
      (e',le,D) -> (e',sustituyeColumna ls n le,n+1)

where v = length (head ls) - 1

deltaEstrella :: MTMP -> Configuracion -> Bool
deltaEstrella (mt,d) ("qf", w, n) = True
deltaEstrella (mt,d) ("qr", w, n) = False
deltaEstrella (mt,d) (q, w, n) =
              deltaEstrella (mt,d) (delta d (q,w,n))
```

Por último, ajustamos la función *aceptaCadena* para concluir con el objetivo de esta sección ya que la función *lenguajeAceptado* sólo sufre un cambio menor, que consiste en recibir y propagar el parámetro que indica el número de pistas de la cinta. La diferencia principal con el modelo estándar se ve reflejado en la configuración inicial para el cómputo de la cadena, ya que esta vez resulta un poco más complicado definirla. Construimos la configuración inicial $c_i = (q, [w_1, ..., w_n], p)$ de la siguiente forma:

- El estado q de c_i es q_0, es decir, el estado inicial de la máquina.
- $[w_1, ..., w_n]$ se construye como sigue: en la primera pista colocamos la cadena que vamos a procesar, en este caso $c = w_1$, y posteriormente agregamos $n - 1$ pistas a la cinta con la instrucción:

  ```
  ++ (replicate (n - 1) ...
  ```

- También tenemos que llenar cada pista con espacios en blancos, tantos como símbolos tenga la cadena a procesar mediante la instrucción:

  ```
  (replicate (length c) ' '
  ```

  Recordemos que la lectura en la cinta es por sectores, por lo que es imperativo que las pistas tengan el mismo número de símbolos y así evitar problemas con el índice de lectura, es decir, las pistas deben tener en todo momento un símbolo que leer en la posición indicada por el índice de la configuración.

- Comenzamos leyendo el primer sector de la cinta correspondiente al índice 0.

La implementación de la función *aceptaCadena* se muestra a continuación:

```
aceptaCadena :: MTMP -> Cadena -> Int -> Bool
aceptaCadena (mt,d) c n = deltaEstrella (mt,d) (qi mt, [c] ++
             (replicate (n - 1) (replicate (length c) '_')), 0)
```

La función *lenguajeAceptado* se ve prácticamente igual:

```
lenguajeAceptado :: MTMP -> Int -> [Cadena]
lenguajeAceptado (mt,d) n =
                 [c|c <- klns (s mt), aceptaCadena (mt,d) c n]
```

La implementación de estas funciones se encuentra en el archivo "ModeloMTMP.hs" mientras que el archivo "EjemplosModeloMTMP.hs" contiene la definición de los ejemplos de máquinas correspondientes a esta variante. Las primeras cinco cadenas del lenguaje aceptado por T_1 las obtenemos con la siguiente instrucción:

```
Main> take 5 (lenguajeAceptado mtmpAnBn 2)
["ab","aabb","aaabbb","aaaabbbb","aaaaabbbbb"]
```

Presentamos el código necesario para ver paso a paso la ejecución del proceso de aceptación sobre una cadena.

```
pintaCadena :: Cadena -> Int -> Cadena
pintaCadena (c:cs) 0 = "("++[c]++")" ++ cs
pintaCadena (c:cs) (n+1) = [c] ++ pintaCadena cs n

pintaCinta :: [Cadena] -> Int -> Cadena
pintaCinta [] _ = []
pintaCinta (l:ls) n = pintaCadena l n ++ "  " ++pintaCinta ls n

pintaDeltaEstrella :: MTMP -> Configuracion -> IO()
pintaDeltaEstrella (mt,d) ("qf",lc,n) =
                   putStrLn("|- "++"qf"++"  "++pintaCinta lc n)
                   >> putStrLn("|- True")
pintaDeltaEstrella (mt,d) ("qr",lc,n) =
                   putStrLn("|- "++"qr"++"  "++pintaCinta lc n)
```

```
                    >> putStrLn("|- False")
pintaDeltaEstrella (mt,d) (q,lc,n) =
                    putStrLn("|- "++q++"  "++pintaCinta lc n)
                    >> pintaDeltaEstrella (mt,d) (delta d (q,lc,n))

pintaProcesoAceptacion :: MTMP -> Cadena -> Int -> IO()
pintaProcesoAceptacion (mt,d) c n =
                   pintaDeltaEstrella (mt,d) (qi mt, [c] ++
                   (replicate (n - 1) (replicate (length c) '_')), 0)
```

El proceso práctico es inmediato del teórico. A continuación se muestra la ejecución de ambos sobre la cadena "aabb":

Teórico

$$
\begin{array}{lll}
(q_0, [aabb, \flat\flat\flat\flat], 0) & \vdash & \\
& \vdash & (q_1, [aabb, *\flat\flat\flat], 1) \quad \vdash \quad (q_1, [aabb, *\flat\flat\flat], 2) \quad \vdash \\
& \vdash & (q_2, [aabb, *\flat * \flat], 1) \quad \vdash \quad (q_2, [aabb, *\flat * \flat], 0) \quad \vdash \\
& \vdash & (q_0, [aabb, *\flat * \flat], 1) \quad \vdash \quad (q_1, [aabb, * * *\flat], 2) \quad \vdash \\
& \vdash & (q_1, [aabb, * * *\flat], 3) \quad \vdash \quad (q_2, [aabb, * * **], 2) \quad \vdash \\
& \vdash & (q_2, [aabb, * * **], 1) \quad \vdash \quad (q_0, [aabb, * * **], 2) \quad \vdash \\
& \vdash & (q_3, [aabb, * * **], 3) \quad \vdash \quad (q_3, [aabb\flat, * * * *\flat], 4) \quad \vdash \\
& \vdash & (q_f, [aabb\flat, * * * *\flat], 3)
\end{array}
$$

Práctico

```
Main> pintaProcesoAceptacion mtmpAnBn "aabb" 2
      |- q0  (a)abb  (_)___
      |- q1  a(a)bb  *(_)__
      |- q1  aa(b)b  *_(_)_
      |- q2  a(a)bb  *(_)*_
      |- q2  (a)abb  (*)_*_
      |- q0  a(a)bb  *(_)*_
      |- q1  aa(b)b  **(*)_
      |- q1  aab(b)  ***(_)
      |- q2  aa(b)b  **(*)*
      |- q2  a(a)bb  *(*)**
      |- q0  aa(b)b  **(*)*
      |- q3  aab(b)  ***(*)
      |- q3  aabb(_)  ****(_)
      |- qf  aab(b)_  ***(*)_
      |- True
```

4.2 Implementación de la máquina de Turing multicinta (MTMC)

Partiendo del modelo estándar y teniendo en cuenta que una configuración para esta variante es un par de la forma:

$$(q, [w_0, w_1, ..., w_{n-1}])$$

en donde $w_i \in (\Gamma \cup \{\flat\})^*$, se representa esta configuración por medio del par (q, $[(w_0, m_0)$, (w_1, m_1), ..., $(w_{n-1}, m_{n-1})\,]$), en donde en cada par de la forma (w_i, m_i) se indica el contenido de la i - ésima cinta por medio de w_i mientras que m_i se refiere al índice del símbolo de w_i que está siendo observado en esa cinta. En Haskell definimos de manera directa una configuración para la máquina multicinta como sigue:

```
type Configuracion = (Estado,[(Cadena,Int)])
```

Se observa una lista de pares, en donde cada par indica el contenido de la cinta en cuestión y la posición de la cabeza en esa cinta. Dicha lista contiene tantos pares como número de cintas tenga la máquina. Entonces, la lectura y el reemplazo se hará por bloques, y cada bloque en realidad es un arreglo de símbolos (un símbolo por cada cinta). Dado que la función de transición para esta variante toma la forma:

$$\delta(q, (a_0, a_1, ..., a_{n-1})) = (p, [(b_0, M_0), (b_1, M_1), ..., (b_n, M_{n-1})])$$

tenemos que cambiar el tipo de las transiciones. En éste caso no estamos leyendo la misma posición en cada una de las cintas, ya que el movimiento en las cintas no es uniforme. A cada una se le aplica la sustitución del símbolo que está siendo leído y posteriormente el movimiento indicado por cada M_i. Los índices. La función de transición en Haskell se define como se muestra a continuación:

```
type Delta = Estado -> [Simbolo] -> (Estado,[(Simbolo,Movimiento)])
```

Por comodidad renombraremos el tipo MTE por $MTMC$, para referirnos a la máquina de Turing multicinta. Entonces:

```
type MTMC = (MaqT, Delta)
```

La definición en Haskell para la máquina T_1 de dos cinta que acepta cadenas de la forma $a^n b^n c^n$ con $n \geq 0$ del ejemplo 5 [Capítulo 2, pág. 16] es la siguiente:

```
mtAnBnCn = MT ["q0","qf","q1","q2","q3"]
       "q0"
       "qf"
       ['a','b','c']
       ['_','a','b','c','X']
```

```
mtmcAnBnCn = (mtAnBnCn,d) where
   d "q0" ['a','_'] = ("q1",[('a',D),('X',D)])
 d "q1" ['a','_'] = ("q1",[('a',D),('X',D)])
 d "q1" ['b','_'] = ("q2",[('b',N),('_',I)])
 d "q2" ['b','X'] = ("q2",[('b',D),('X',I)])
 d "q2" ['c','_'] = ("q3",[('c',N),('_',D)])
 d "q3" ['c','X'] = ("q3",[('c',D),('X',D)])
 d "q3" ['_','_'] = ("qf",[('_',N),('_',N)])
 d "q0" ['_','_'] = ("qf",[('_',N),('_',N)])
 d _ _ = ("qr",[('E',N),('E',N)])
```

Esta máquina está disponible en el archivo "EjemplosModeloMTMC.hs".

La definición de los movimientos para la máquina multicinta es igual que la utilizada para el modelo estándar, sólo tenemos que aplicar la definición a cada una de las cintas y posteriormente hacer el cambio de estado. Para hacer posible la implementación de la función δ para esta variante alteramos las funciones *sustituyeColumna* y *columna* vistas en la sección anterior para que manejen la lista de los índices de las posiciones en que se encuentra la cabeza en cada una de las cintas:

```
sustituyeColumna :: [Cadena] -> [Int] -> [Simbolo] -> [Cadena]
sustituyeColumna [] _ _ = []
sustituyeColumna (lw:lws) (ln:lns) (ls:lss) =
              sustituye lw ln ls: sustituyeColumna lws lns lss

columna :: Configuracion -> [Simbolo]
columna (_, []) = []
columna (q, ((l,n):lns)) = l!!(n): columna (q, lns)
```

La función *ejecutaMovimiento* se encarga de hacer las actualizaciones correspondientes en una cinta. La implementación de dicha función prácicamente es igual a la implementación de la función δ del modelo estándar, sólo que esta vez no cargamos el estado de la configuración, como se muestra a continuación:

```
ejecutaMovimiento :: (Cadena,Int) -> (Simbolo,Movimiento) -> (Cadena,Int)
ejecutaMovimiento (w,n) (s,m)
            |(n==v && n/=0) = case m of
                    I -> (sustituye w n s,n-1)
                    N -> (sustituye w n s,n)
                    D -> ((sustituye w n s)++"_",n+1)

            |(n==0) = case m of
                    I -> ('_':sustituye w n s,0)
                    N -> (sustituye w n s,0)
                    D -> if n==v
                         then ((sustituye w n s)++"_",1)
                         else (sustituye w n s,n+1)
```

```
        |otherwise = case m of
                I -> (sustituye w n s,n-1)
                N -> (sustituye w n s,n)
                D -> (sustituye w n s,n+1)

        where v = length w - 1
```

La función *actualizaCintas* se apoya en *ejecutaMovimiento* para realizar los cambios derivados de la función de transición al ejecutar un paso computacional en cada una de las cintas.

```
actualizaCintas :: [(Cadena,Int)] -> [(Simbolo,Movimiento)] -> [(Cadena,Int)]
actualizaCintas [] _ = []
actualizaCintas (lc:lcs) (lp:lps) =
                    ejecutaMovimiento lc lp:actualizaCintas lcs lps
```

A continuación definimos las funciones *delta* y *deltaEstrella* para la máquina de Turing multicinta. Ambas funciones conservan los tipos y en el caso de *deltaEstrella* se conserva también la implementación:

```
delta :: Delta -> Configuracion -> Configuracion
delta d (e,lc) = (fst lp, actualizaCintas lc (snd lp))
      where lp = d e (columna (e,lc))

deltaEstrella :: MTMC -> Configuracion -> Bool
deltaEstrella (mt,d) ("qf",lc) = True
deltaEstrella (mt,d) ("qr",lc) = False
deltaEstrella (mt,d) (q,lc) = deltaEstrella (mt,d) (delta d (q,lc))
```

Por último, ajustamos la función *aceptaCadena* para concluir con el objetivo de esta sección, mientras que la función *lenguajeAceptado* no sufre cambio alguno. La diferencia principal con el modelo estándar se refleja en la configuración inicial. En la implementación de la función *aceptaCadena* se recibe la máquina, la cadena de entrada y el número de cintas. Dentro del cuerpo de la función tomamos el segundo parámetro de entrada formando un par para la primera cinta con la cadena que se va a procesar; esto lo hacemos con la instrucción $(c, 0)$ y creamos $(n-1)$ pares para el resto de las cintas, en donde el primer argumento es una cadena de blancos de la misma longitud de w y segundo argumento corresponde al índice de la posición de la cabeza para cada cinta:

```
aceptaCadena :: MTMC -> Cadena -> Int -> Bool
aceptaCadena (mt,d) c n = deltaEstrella (mt,d) (qi mt, (c,0)
             :replicate (n - 1) ((replicate (length c) '_'),0))
```

Las definiciones de las funciones se encuentran en el archivo "ModeloMTMC.hs" mientras que los ejemplos están disponibles en el archivo "EjemplosModeloMTMC.hs". Obtenemos las primeras cinco cadenas del lenguaje generado por T_1 con la siguiente instrucción:

```
Main> take 5 (lenguajeAceptado mtmcAnBnCn 2)
["abc","aabbcc","aaabbbccc","aaaabbbbcccc","aaaaabbbbbccccc"]
```

Concluimos esta sección con el código necesario para observar paso a paso la ejecución del proceso de aceptación sobre una cadena.

```
pintaCadena :: (Cadena,Int) -> Cadena
pintaCadena (c:cs,0) = "("++[c]++")" ++ cs
pintaCadena (c:cs,n+1) = [c] ++ pintaCadena (cs,n)

pintaCinta :: [(Cadena,Int)] -> Cadena
pintaCinta [] = []
pintaCinta (l:ls) =  "["++ pintaCadena l ++ "]  " ++ pintaCinta ls

pintaDeltaEstrella :: MTMC -> Configuracion -> IO()
pintaDeltaEstrella (mt,d) ("qf",lc) =
                          putStrLn("|- "++"qf"++"  "++pintaCinta lc) >>
                          putStrLn("|- True")
pintaDeltaEstrella (mt,d) ("qr",lc) =
                          putStrLn("|- "++"qr"++"  "++pintaCinta lc) >>
                          putStrLn("|- False")
pintaDeltaEstrella (mt,d) (q,lc) =
                          putStrLn("|- "++q++"  "++pintaCinta lc) >>
                          pintaDeltaEstrella (mt,d) (delta d (q,lc))

pintaProcesoAceptacion :: MTMC -> Cadena -> Int -> IO()
pintaProcesoAceptacion (mt,d) c n =
                          pintaDeltaEstrella (mt,d) (qi mt, (c,0):
                          replicate (n - 1) ((replicate (length c) '_'),0))
```

La ejecución del proceso de aceptación para la cadena "aabbcc" con la máquina *mtmcAn BnCn* se muestra a continuación:

```
Main> pintaProcesoAceptacion mtmcAnBnCn "aabbcc" 2
|- q0  [(a)abbcc]  [(_)_____]
|- q1  [a(a)bbcc]  [X(_)____]
|- q1  [aa(b)bcc]  [XX(_)___]
|- q2  [aa(b)bcc]  [X(X)____]
|- q2  [aab(b)cc]  [(X)X____]
|- q2  [aabb(c)c]  [(_)XX____]
|- q3  [aabb(c)c]  [_(X)X____]
|- q3  [aabbc(c)]  [_X(X)____]
|- q3  [aabbcc(_)]  [_XX(_)___]
|- qf  [aabbcc(_)]  [_XX(_)___]
|- True
```

En el marco teórico la ejecución de este proceso se ve como sigue:

$$
\begin{aligned}
(q_0, [(aabbcc, 0), (\flat\flat\flat\flat\flat\flat, 0)]) &\vdash \\
&\vdash (q_1, [(aabbcc, 1), (X\flat\flat\flat\flat\flat, 1)]) \vdash \\
&\vdash (q_1, [(aabbcc, 2), (XX\flat\flat\flat\flat, 2)]) \vdash \\
&\vdash (q_2, [(aabbcc, 2), (XX\flat\flat\flat\flat, 1)]) \vdash \\
&\vdash (q_2, [(aabbcc, 3), (XX\flat\flat\flat\flat, 0)]) \vdash \\
&\vdash (q_2, [(aabbcc, 4), (\flat XX\flat\flat\flat\flat, 0)]) \vdash \\
&\vdash (q_3, [(aabbcc, 4), (\flat XX\flat\flat\flat\flat, 1)]) \vdash \\
&\vdash (q_3, [(aabbcc, 5), (\flat XX\flat\flat\flat\flat, 2)]) \vdash \\
&\vdash (q_3, [(aabbcc\flat, 6), (\flat XX\flat\flat\flat\flat, 3)]) \vdash \\
&\vdash (q_f, [(aabbcc\flat, 6), (\flat XX\flat\flat\flat\flat, 3)])
\end{aligned}
$$

4.3 Implementación de la máquina de Turing no determinista (MTND)

Para la implementación de esta variante nos apoyaremos de manera directa en la implementación del modelo estándar, ya que podemos reutilizar gran parte del código; por ejemplo, todos los tipos usados, los movimientos sobre la cinta y funciones auxiliares. Esto se debe a que los componentes de la máquina se tratan igual; de hecho, la configuración inicial para el proceso de aceptación y la ejecución del proceso son los mismos, hasta que nos encontramos con un no determinismo. En ese momento el proceso deja de ser lineal para poder explorar a la par todas las ramas abiertas. Sin embargo, cada rama abierta se maneja de manera análoga a como se maneja la cinta en el modelo estándar. Las ramas que presenten el estado "qr" se cierran y dejan de contemplarse mientras que si alguna rama presenta el estado "qf" el proceso termina y la cadena es aceptada por la máquina.

La primera diferencia entre el modelo estándar y el no determinista, se ve reflejada en la función de transición. Recordemos que las transiciones para esta variante toman la siguiente forma:

$$\delta(q, a) = \{(p_0, b_0, M_0), (p_1, b_1, M_1), ..., (p_{n-1}, b_{n-1}, M_{n-1})\},$$

lo que sugiere cambiar la definición del tipo *Delta* en la implementación por:

```
type Delta = Estado -> Simbolo -> [(Estado, Simbolo, Movimiento)]
```

La máquina T_1 que acepta el lenguaje $L = \{w : w$ contiene la subcadena cab o la subcadena $abc\}$ del ejemplo 6 [Capítulo 2, pág. 18] se define como sigue:

```
mtCABoABC = MT ["q0", "qf", "q1", "q2", "q3", "q4"]
         "q0"
         "qf"
        ['a','b','c']
        ['_','a','b','c']
```

```
mtndCABoABC = (mtCABoABC,d) where
        d "q0" 'a' = [("q0",'a',D)]
        d "q0" 'b' = [("q0",'b',D)]
        d "q0" 'c' = [("q0",'c',D),("q1",'c',D),("q2",'c',I)]
        d "q1" 'a' = [("q3",'a',D)]
        d "q2" 'b' = [("q4",'b',I)]
        d "q3" 'b' = [("qf",'b',D)]
        d "q4" 'a' = [("qf",'a',I)]
        d _ _ = [("qr",'E',N)]
```

En el modelo multicinta utilizamos las funciones *ejecutaMovimiento* y *actualizaCintas*. Tenemos que ajustar estas funciones para manejar el cambio de estado en cada una de las cintas cuando se presente el no determinismo:

```
ejecutaMovimiento :: Configuracion
                  -> (Estado,Simbolo,Movimiento) -> Configuracion
ejecutaMovimiento (e,w,n) (e',s,m)
            |(n==v && n/=0) = case m of
                          I -> (e',sustituye w n s,n-1)
                          N -> (e',sustituye w n s,n)
                          D -> (e',(sustituye w n s)++"_",n+1)

            |(n==0) = case m of
             I -> (e','_':sustituye w n s,n)
             N -> (e',sustituye w n s,n)
             D -> if n==v
                  then (e',(sustituye w n s)++"_",n+1)
                  else (e',sustituye w n s,n+1)

            |otherwise = case m of
                          I -> (e',sustituye w n s,n-1)
                          N -> (e',sustituye w n s,n)
                          D -> (e',sustituye w n s,n+1)

            where v = length w - 1
```

La función *actualizaCintas* es la encargada de manejar el no determinismo. Tomamos una configuración y vamos a buscar dentro de nuestras transiciones una que sea aplicable a dicha configuración. Dado que la máquina es no determinista, la imagen de la transición tiene que ser una lista, la cual puede tener uno o más elementos en donde cada elemento es una terna de la forma (*Estado*, *Simbolo*, *Movimiento*). Podemos dar seguimiento al proceso por medio de un árbol, en donde cada rama es el resultado de aplicarle a la configuración actual cada una de las ternas de la lista, devolviendo una lista de configuraciones (cada configuración representa una rama).

```
actualizaCintas :: Configuracion
```

```
                        -> [(Estado,Simbolo,Movimiento)] -> [Configuracion]
actualizaCintas _ [] = []
actualizaCintas c (lp:lps) = ejecutaMovimiento c lp:actualizaCintas c lps
```

El tipo de la función *delta* vista en la sección anterior cambia ya que tiene que devolver la lista de configuraciones de cada una de las hojas. Cada hoja crece a una rama si es que hay posibilidad de llevar a cabo una transición a partir de la configuración de la hoja y cada rama representa un reconocimiento. También la función *deltaEstrella* se ve afectada por el no determinismo, ya que como mencionamos antes, en cada paso tenemos que verificar dos cosas: que las hojas no contengan el estado de error y si alguna hoja contiene el estado final de la máquina. Para esto nos apoyamos en las funciones *buscaEstadosError* y *buscaEstadoFinal*. La función *buscaEstadosError* toma la lista de configuraciones y regresa la misma lista pero sin aquellas configuraciones con el estado "qr"; en caso de que todas las configuraciones presenten este estado, la función devuelve la lista vacía. La función *buscaEstadoFinal* recorre las configuraciones verificando si en alguna de ellas hemos llegado al estado final, si la encontramos devolvemos $True$; en otro caso, devolvemos $False$. Entonces las funciones *buscaEstadoError* y *buscaEstadoFinal* sugieren ser los casos base para la función *deltaEstrella*. La implementación de dichas funciones se muestra a continuación:

```
buscaEstadosError :: [Configuracion] -> [Configuracion]
buscaEstadosError [] = []
buscaEstadosError ((a,b,c):lcs) =
                          if a == "qr"
                          then buscaEstadosError lcs
                          else (a,b,c):buscaEstadosError lcs

buscaEstadoFinal :: [Configuracion] -> Bool
buscaEstadoFinal [] = False
buscaEstadoFinal ((a,b,c):lcs) =
                           if a == "qf"
                           then True
                           else buscaEstadoFinal mt lcs
```

La justificación de estas funciones se puede observar al ejecutar el proceso teórico de aceptación sobre la cadena "aacbabcc", en el cual se muestra la manera en la que se van limpiando las ramas que no tiene caso seguir analizando, como se muestra a continuación:

$$
\begin{array}{ll}
[(q_0, aacbabcc, 0)] & \vdash \\
 & \vdash \qquad [(q_0, aacbabcc, 1)] \qquad \vdash \\
 & \vdash \qquad [(q_0, aacbabcc, 2)] \qquad \vdash \\
 & \vdash \quad [(q_0, aacbabcc, 3), (q_1, aacbabcc, 3), (q_2, aacbabcc, 1)] \quad \vdash \\
 & \vdash \quad [(q_0, aacbabcc, 4), (q_r, aacEabcc, 3), (q_r, aEcbabcc, 1)] \quad \vdash \\
 & \vdash \qquad [(q_0, aacbabcc, 5)] \qquad \vdash \\
 & \vdash \qquad [(q_0, aacbabcc, 6)] \qquad \vdash \\
 & \vdash \quad [(q_0, aacbabcc, 7), (q_1, aacbabcc, 7), (q_2, aacbabcc, 5)] \quad \vdash \\
 & \vdash \quad [(q_0, aacbabcc, 8), (q_r, aacbabcE, 7), (q_4, aacbabcc, 4)] \quad \vdash \\
 & \vdash \qquad [(q_r, aacbabccE, 8), (q_f, aacbabcc, 3)] \qquad \vdash
\end{array}
$$

Para el paso recursivo nos apoyamos en la función *deltaAuxiliar*, que se encarga de ejecutar un movimiento a la vez en cada configuración que tengamos. El objetivo de esta función es simular que estamos procesando todas las configuraciones al mismo tiempo. La implementación del bloque de funciones para la función de transición es la siguiente:

```
delta :: Delta -> Configuracion -> [Configuracion]
delta d (e,w,n) = actualizaCintas (e,w,n) lp
      where lp = d e (w!!n)

deltaAuxiliar :: Delta -> [Configuracion] -> [Configuracion]
deltaAuxiliar _ [] = []
deltaAuxiliar  d (lc:lcs) = delta d lc++deltaAuxiliar d lcs

deltaEstrella :: MTND -> [Configuracion] -> Bool
deltaEstrella (mt,d) lc
              |(buscaEstadosError lc == []) = False
              |(buscaEstadoFinal lc) = True
              |otherwise = deltaEstrella (mt,d) (deltaAuxiliar d lc)
      where ls = buscaEstadosError lc
```

En la implementación de la función *aceptaCadena* renombramos el tipo MTE por $MTND$ y cambiamos la instrucción $(qimt, c, 0)$ por $[(qimt, c, 0)]$, es decir, encapsulamos la configuración inicial en una lista:

```
aceptaCadena :: MTND -> Cadena -> Bool
aceptaCadena (mt,d) c = deltaEstrella (mt,d) [(qi mt, c, 0)]
```

La función para obtener el lenguaje aceptado queda intacta. Las implementaciones de estas funciones se encuentran en el archivo "ModeloMTND.hs" mientras que las definiciones de los ejemplos se encuentran en el archivo "EjemplosModeloMTND.hs". Obtenemos las primeras ocho cadenas del lenguaje aceptado por T_1 con la siguiente instrucción:

```
Main> take 8 (lenguajeAceptado mtndCABoABC)
["abc","cab","aabc","abca","abcb","abcc","acab","babc"]
```

Para observar a detalle el proceso de aceptación se requieren las siguientes funciones:

```
pintaConfiguracion :: Configuracion -> String
pintaConfiguracion (e,c,0) = "("++e++")" ++ c
pintaConfiguracion (e,c:cs,n+1) = [c] ++ pintaConfiguracion (e,cs,n)

pintaConfiguraciones :: [Configuracion] -> String
pintaConfiguraciones [] = []
pintaConfiguraciones (c:cs) = pintaConfiguracion c
                                ++ "    " ++pintaConfiguraciones cs

pintaDeltaEstrella :: MTND -> [Configuracion] -> IO()
pintaDeltaEstrella (mt,d) lc
```

```
            |(buscaEstadosError lc == []) =
                  putStrLn("|- " ++ pintaConfiguraciones lc)
                  >> putStrLn("|- False")
            |(buscaEstadoFinal lc) =
                  putStrLn("|- " ++ pintaConfiguraciones lc)
                  >> putStrLn("|- True")
            |otherwise =
                  putStrLn("|- " ++ pintaConfiguraciones lc)
                  >> pintaDeltaEstrella (mt,d) (deltaAuxiliar d lc)
            where ls = buscaEstadosError lc

pintaProcesoAceptacion :: MTND -> Cadena -> IO()
pintaProcesoAceptacion (mt,d) c =
                                 pintaDeltaEstrella (mt,d) [(qi mt, c, 0)]
```

Por ejemplo, la ejecución del proceso de aceptación sobre la cadena "aacbabcc" se muestra a continuación:

```
Main> pintaProcesoAceptacion mtndCABoABC "aacbabcc"
|- (q0)aacbabcc
|- a(q0)acbabcc
|- aa(0)cbabcc
|- aac(q0)babcc    aac(q1)babcc    a(q2)acbabcc
|- aacb(q0)abcc    aac(qr)Eabcc    a(qr)Ecbabcc
|- aacba(q0)bcc    aac(qr)Eabcc    a(qr)Ecbabcc
|- aacbab(q0)cc    aac(qr)Eabcc    a(qr)Ecbabcc
|- aacbabc(q0)c    aacbabc(q1)c    aacba(q2)bcc    ...
...aac(qr)Eabcc    a(qr)Ecbabcc
|- aacbabcc(q0)_    aacbabcc(q1)_    aacbab(q2)cc  ...
...aacbabc(qr)E    aacb(q4)abcc    aac(qr)Eabcc    ...
...a(qr)Ecbabcc
|- aacbabcc(qr)E    aacbabcc(qr)E    aacbab(qr)Ec  ...
...aacbabc(qr)E    aac(qf)babcc    aac(qr)Eabcc    ...
...a(qr)Ecbabcc
|- True
```

Con esto concluímos la implementación de las variantes del modelo estándar de la máquina de Turing.

En el siguiente capítulo nos dedicamos a la máquina universal de Turing cuya importancia radica en el hecho de que todas las máquinas de Turing para el modelo estándar se pueden simular por medio de ésta y por consiguiente se recomienda sea leído cuidadosamente.

5 Máquina universal de Turing (MUT)

La máquina universal de Turing evita el tener que crear una máquina para cada algoritmo en específico ya que con ella se puede simular el proceso de ejecución de cualquier máquina estándar y funciona como sigue:

La máquina universal de Turing recibe como parámetro una *cadena* que corresponde a una máquina de Turing estándar M de propósito especial y una *cadena* z, que se interpreta como entrada de M. La máquina universal de Turing es capaz de simular el comportamiento de M sobre z.

El primer paso es formular un sistema de notación en el que puedan codificarse tanto M como z en un alfabeto fijo. Esta codificación se rige mediante la función f_c, la cual definiremos más adelante. El aspecto crucial de la codificación es que no debe destruir información; dadas las cadenas $f_c(M)$ y $f_c(z)$, debe ser posible reconstruir M y z; este proceso se conoce como *decodificación*. Por convención se utiliza el alfabeto $\{0, 1\}$, si bien debe recordarse que la máquina que se codifica puede tener un alfabeto mucho mayor, se empieza por asignar enteros positivos a cada estado, a cada símbolo de los alfabetos y a cada una de las tres direcciones (I, N y D) de la máquina que se pretende codificar.

Para dar cabida a cualquier máquina que se quiera codificar y tener la certeza de que la codificación sea uno a uno, se debe lograr que ningún símbolo en un alfabeto reciba el mismo número que otro símbolo, lo mismo debe ocurrir al codificar los estados. Para llevar a cabo la codificación se numeran los símbolos del alfabeto Γ en forma consecutiva, el símbolo blanco ocupa el primer lugar, mientras que el conjunto de estados debe tener en la primera posición el estado inicial, en la segunda posición el estado final y en las posiciones restantes los estados que faltan listados en orden.

5.1 Codificación

Este proceso toma como entrada una máquina de Turing estándar M y le aplica la función de codificación f_c convirtiendo todo en cadenas de ceros y unos, ya que éstos son los únicos símbolos reconocidos por la máquina universal. La función f_c primero codifica el

alfabeto Γ, codifica el conjunto de estados y codifica las direcciones de movimiento de la cabeza. Posteriormente lleva a cabo la codificación de la función de transición. Una vez que tenemos las codificaciones de todos los casos de la función de transición, concatenamos estas codificaciones separándolas con un cero y el resultado es la codificación de la máquina M. La definición de la función f_c se muestra a continuación:

Definición 5.1.1 (Función de codificación f_c)**.** La función f_c asigna una cadena de unos a cada símbolo del alfabeto Γ comenzando por el símbolo $\flat$, a cada estado de Q en el siguiente orden q_0, q_f, ..., y a cada una de las tres direcciones.
Sea:

1. Símbolos:

$$f_c(\flat) = 1$$
$$f_c(a_i) = 1^{i+1} \text{ (para cada } a_i \in \Gamma)$$

2. Estados:

$$f_c(q_0) = 1$$
$$f_c(q_f) = 11$$
$$f_c(q_i) = 1^{i+2} \text{ (para cada } q_i \in Q)$$

3. Movimientos:

$$f_c(I) = 1$$
$$f_c(N) = 11$$
$$f_c(D) = 111$$

Posteriormente codificamos cada caso t de la función de transición de una máquina estándar, descrito con la fórmula:

$$\delta\ (p, a) = (q, b, M)$$

mediante la cadena:

$$f_c\ (t) = f_c(p)0f_c(a)0f_c(q)0f_c(b)0f_c(M)0$$

Entonces, una máquina de Turing estándar M se codifica con la cadena:

$$f_c\ (M) = f_c(t_0)0f_c(t_1)0...f_c(t_n)0$$

en donde t_0, t_1, ..., t_n son los distintos casos de la función de transición de la máquina M, dispuestos en un orden arbitrario, lo que hace que la codificación no sea única. Por último, cualquier cadena $w = w_0w_1...w_k$, donde $w_i \in \Gamma$ se codifica con la cadena:

$$f_c\ (w) = 0f_c(w_0)0f_c(w_1)0...f_c(w_k)0$$

El cero al comienzo de la cadena $f_c\ (w)$ se incluye para que en una cadena compuesta de la forma $f_c\ (M) f_c\ (w)$ no haya duda en donde termina la codificación de la máquina, es decir, $f_c\ (M)$. Aunque la codificación de una máquina no es única, cualquier cadena de ceros y unos puede ser a lo sumo la codificación de una máquina estándar.

Para ejemplificar el proceso de codificación, consideremos la máquina $M = (Q, [q_f], [a, b], \Gamma, q_0, \delta)$ que acepta el lenguaje $L = \{a^n b^n \mid n > 0\ \}$ y la cadena $w = aabb$, donde:

$$\begin{aligned} Q = &\quad [q_0, q_f, q_1, q_2, q_3] \\ \Gamma = &\quad [\flat, a, b, X, Y] \end{aligned}$$

y

$$\begin{aligned} \delta(q_0, a) &= (q_1, X, D) \quad (t_1) \\ \delta(q_0, Y) &= (q_3, Y, D) \quad (t_2) \\ \delta(q_1, a) &= (q_1, a, D) \quad (t_3) \\ \delta(q_1, Y) &= (q_1, Y, D) \quad (t_4) \\ \delta(q_1, b) &= (q_2, Y, I) \quad (t_5) \\ \delta(q_2, a) &= (q_2, a, I) \quad (t_6) \\ \delta(q_2, Y) &= (q_2, Y, I) \quad (t_7) \\ \delta(q_2, X) &= (q_0, X, D) \quad (t_8) \\ \delta(q_3, Y) &= (q_3, Y, D) \quad (t_9) \\ \delta(q_3, \flat) &= (q_f, \flat, D) \quad (t_{10} \end{aligned}$$

La codificación de Q y de Γ se muestra a continuación:

$$\begin{aligned} f_c(q_0) &= 1 \\ f_c(q_f) &= 11 \\ f_c(q_1) &= 1^{1+2} = 111 \\ f_c(q_2) &= 1^{2+2} = 1111 \\ f_c(q_3) &= 1^{3+2} = 11111 \end{aligned}$$

$$
\begin{aligned}
f_c(\flat) &= 1 \\
f_c(a) &= 1^{1+1} = 11 \\
f_c(b) &= 1^{2+1} = 111 \\
f_c(X) &= 1^{3+1} = 1111 \\
f_c(Y) &= 1^{4+1} = 11111
\end{aligned}
$$

Mientras que la codificación de la función de transición δ da como resultado:

$$
\begin{aligned}
f_c(t_1) &= f_c(q_0)0f_c(a)0f_c(q_1)0f_c(X)0f_c(D)0 &&= 101101110111101110 \\
f_c(t_2) &= f_c(q_0)0f_c(Y)0f_c(q_3)0f_c(Y)0f_c(D)0 &&= 10111110111110111111101110 \\
f_c(t_3) &= f_c(q_1)0f_c(a)0f_c(q_1)0f_c(a)0f_c(D)0 &&= 111011011101101110 \\
f_c(t_1) &= f_c(q1)0f_c(Y)0f_c(q1)0f_c(Y)0f_c(D)0 &&= 11101111101110111111101110 \\
f_c(t_5) &= f_c(q_1)0f_c(b)0f_c(q_2)0f_c(Y)0f_c(I)0 &&= 111011101111011111010 \\
f_c(t_6) &= f_c(q_2)0f_c(a)0f_c(q_2)0f_c(a)0f_c(I)0 &&= 111101101111011010 \\
f_c(t_7) &= f_c(q_2)0f_c(Y)0f_c(q_2)0f_c(Y)0f_c(I)0 &&= 11110111110111011111010 \\
f_c(t_8) &= f_c(q_2)0f_c(X)0f_c(q_0)0f_c(X)0f_c(D)0 &&= 111101111010111101110 \\
f_c(t_9) &= f_c(q_3)0f_c(Y)0f_c(q_3)0f_c(Y)0f_c(D)0 &&= 1111101111101111101111101110 \\
f_c(t_{10}) &= f_c(q_3)0f_c(\flat)0f_c(q_f)0f_c(\flat)0f_c(D)0 &&= 11111010110101110
\end{aligned}
$$

5.1.1. Implementación del proceso de codificación

El objetivo de este apartado es la implementación de la función *codifica* que corresponde a la función f_c. Las primeras dos funciones que vamos a implementar para llevar a cabo esta tarea son $codificaMovimientos$ y *obtenPosicion*. En la función $codificaMovimientos$ se hace la codificación de los movimientos de manera directa. Mientras que en la función *obtenPosicion* se utilizan variables de tipo polimórfico ya que será utilizada tanto para determinar la posición de un símbolo dentro del alfabeto Γ como para determinar la posición de un estado dentro de Q. La implementación de estas funciones se muestra a continuación:

```
codificaMovimientos :: Movimiento -> String
codificaMovimientos I = "1"
codificaMovimientos N = "11"
codificaMovimientos D = "111"

obtenPosicion :: (Eq a) => a -> [a] -> Int
obtenPosicion _ [] = 0
obtenPosicion e (x:xs) = if (elem e (x:xs))
```

```
                    then if x == e
                         then 0
                         else 1 + obtenPosicion e xs
                    else -length (x:xs)
```

La base de la función *codifica* es la función *codificaTransiciones*. Esta función recibe la máquina de Turing estándar que se va a codificar, la lista L de todos los pares posibles cuya forma sea $(e,\ s)$, en donde $e \in Q$, $s \in \Gamma$ y devolvemos la codificación que le corresponde a la función de transición. Entonces, contemplamos dos casos en la función *codificaTransiciones*:

1. En el primer caso devolvemos la lista vacía cuando no encontremos un par (*Estado*, *Simbolo*), en donde tanto *Estado* como *Simbolo* coincidan con el *estado actual* y *simbolo actual* de alguno de los casos definidos en δ, esto lo determinamos una vez que hayamos recorrido toda la lista.

2. En el segundo caso tenemos dos opciones: cuando δ nos manda al estado q_r o cuando nos manda a un estado de Q. Cuando llegamos al estado q_r simplemente nos pasamos a estudiar el siguiente par de la lista; pero cuando llegamos a un estado de Q buscamos la posición p de cada componente de la transición, generando para cada uno de ellos una cadena con $p+1$ copias del carácter 1, es decir, les asignamos una codificación en base a la posición de este componente en la lista de estados o en la lista que representa el alfabeto Γ. Después concatenamos dichas codificaciones colocando un cero entre una codificación y otra. Aplicamos el mismo proceso al resto de la lista.

La implementación de la función *codificaTransiciones* es la siguiente:

```
codificaTransiciones :: MTE -> [(Estado,Simbolo)] -> String
codificaTransiciones _ [] = []
codificaTransiciones (mt,d) ((e, s):xs) = case d e s of
                ("qr",s',m) -> codificaTransiciones (mt,d) xs
                (e',s',m) ->
                     ("0" ++ replicate ((obtenPosicion e le) + 1) '1' ++
                     "0" ++ replicate ((obtenPosicion s ls) + 1) '1' ++
                     "0" ++ replicate ((obtenPosicion e' le) + 1) '1' ++
                     "0" ++ replicate ((obtenPosicion s' ls) + 1) '1' ++
                     "0" ++ (codificaMovimientos m) ++ "0")
                     ++ codificaTransiciones (mt,d) xs
                where le = q mt
                      ls = g mt
```

La implementación de la función *codifica* se muestra a continuación:

```
codifica :: MTE -> String
codifica mte = codificaTransiciones mte lp
    where lp = [(e,s)| e <- q (fst mte) , s <- g (fst mte)]
```

La defínción de la máquina *mtePal* que acepta palíndromos sobre el alfabeto $\{a, b\}$ está disponible en el archivo "EjemplosModeloMTELA.hs", mientras que la implentación

de la función *codifica* se encuentra disponible en el archivo "MUT.hs". El resultado de aplicarle el proceso de codificación a la máquina *mtePal* es el siguiente:

```
mtPal = MT ["q0","qf","q1","q2","q3","q4","q5"]
            "q0"
            "qf"
           ['a','b']
           ['_','a','b']

mtePal = (mtPal,d7) where
          d7 "q0" 'a' = ("q1",'_',D)
          d7 "q0" 'b' = ("q4",'_',D)
          d7 "q0" '_' = ("qf",'_',D)
          d7 "q1" 'a' = ("q1",'a',D)
          d7 "q1" 'b' = ("q1",'b',D)
          d7 "q1" '_' = ("q2",'_',I)
          d7 "q2" '_' = ("qf",'_',D)
          d7 "q2" 'a' = ("q3",'_',I)
          d7 "q3" 'a' = ("q3",'a',I)
          d7 "q3" 'b' = ("q3",'b',I)
          d7 "q3" '_' = ("q0",'_',D)
          d7 "q4" 'a' = ("q4",'a',D)
          d7 "q4" 'b' = ("q4",'b',D)
          d7 "q4" '_' = ("q5",'_',I)
          d7 "q5" 'b' = ("q3",'_',I)
          d7 "q5" '_' = ("qf",'_',D)
          d7 _ _ = ("qr",'E',N)



Main> codifica mte
      "01010110101110
       0101101110101110
       01011101111110101110
       0111010111101010
       011101101110110110
       01110111011011101110
       01111010110101110
       011110110111101010
       01111101010101110
       011111011011111011010
       0111110111011111110111010  ... (*)
       011111010111111101010
       01111110110111111101101110
       011111101110111111011101110
       01111111010110101110
```

```
01111111011101111101010"
```

Verificamos el proceso descomponiendo el bloque de codificación marcado con $(*)$. Este bloque igual que cualquier caso de la función δ consta de 5 elementos que son: estado actual e_a, símbolo actual s_a, estado siguiente e_s, símbolo siguiente s_s y dirección del movimiento m. Como habíamos mencionado, la codificación de cada elemento queda delimitada por los ceros añadidos durante el proceso. Entonces, las codificaciones para los elementos del caso $(*)$ son las siguientes: 11111, 111, 11111, 111 y 1. Como el orden de los elementos se conserva tenemos que:

$$\begin{aligned} f_c(e_a) &= 11111 \\ f_c(s_a) &= 111 \\ f_c(e_s) &= 11111 \\ f_c(s_s) &= 111 \\ f_c(m) &= 1 \end{aligned}$$

Tomando los elementos correspondientes en la lista de estados y en la lista de símbolos del alfabeto Γ que correspondan al número de unos usados para su codificación tenemos que:

$$\begin{aligned} e_a &= q_3 \\ s_a &= b \\ e_s &= q_3 \\ s_s &= b \\ m &= I \end{aligned}$$

Por lo tanto, 011111011101111110111010 es la codificación de la transición:

$$\delta\ (q_3,\ b) = (q_3,\ b,\ I)$$

En las secciones restantes de este capítulo vamos a implementar el funcionamiento de la máquina universal de Turing y también la función inversa a la codificación (decodificación). En ambas secciones trabajaremos con la máquina que acepta palíndromos sobre el alfabeto $\{a, b\}$.

5.2 Funcionamiento de la máquina universal de Turing

La máquina universal es una máquina multicinta que funciona con tres cintas independientes. Las cintas no se tratan de la forma convencional como se vio en el cuarto capítulo. A continuación se describe el rol de cada cinta:

- Cinta 1 (C_1): almacena la codificación en ceros y unos de alguna máquina estándar. Esta codificación se obtiene aplicando el proceso *codifica* descrito en la sección anterior.

- Cinta 2 (C_2): almacena la codificación en ceros y unos de la cadena de entrada. Esta cinta se representa con una terna de la forma $(Cadena, Int, Int)$. El primer parámetro es la codificación de la cadena sobre la cinta, el segundo parámetro indica la posición en donde empieza la codificación del símbolo que estamos leyendo mientras que el tercer parámetro indica el número de unos que tiene la codificación de dicho símbolo.
- Cinta 3 (C_3): almacena la codificación en ceros y unos del estado actual.

El funcionamiento de la máquina universal es análogo al de una máquina estándar y se puede ejecutar en el siguiente orden:

1. Extraemos de C_2 la codificación del símbolo que está siendo leído, la cual queda delimitada por el segundo y tercer elemento de la terna, llamémosle $cod_{sl'}$.
2. Concatenamos el contenido de C_3, llamémosle $cod_{ea'}$ con $cod_{sl'}$.
3. Veamos a C_1 como un conjunto de codificaciones concatenadas, cada una de ellas forma lo que llamaremos un *bloque de codificación*. Cada bloque esta compuesto por cinco ceros que delimitan las codificaciones de los elementos originales de la máquina estándar, es decir, cada bloque es de la forma 0 cod_{ea} 0 cod_{sl} 0 cod_{es} 0 cod_{se} 0 cod_m0. Entonces tenemos que buscar en C_1 un bloque tal que $cod_{ea} = cod_{ea'}$ y $cod_{sl} = cod_{sl'}$. En caso de no encontrarlo la cadena no es aceptada y termina la ejecución.
4. Una vez hecho esto tomamos la "imagen del bloque" que sería la parte 0 cod_{es} 0 cod_{se} 0 cod_m0 y procedemos a realizar las siguientes sustituciones:
   - En C_2 sustituimos $cod_{sl'}$ por cod_{se}.
   - Actualizamos el segundo elemento de C_2 dependiendo del movimiento indicado por la transición, ya que tenemos que movernos al inicio de la codificación del símbolo que está a la izquierda del que fue reemplazado en caso de que $cod_m = 1$, o al inicio de la codificación que está a la derecha del que fue reemplazado en caso de que $cod_m = 111$, o seguiremos apuntando a la misma posición en caso de que $cod_m = 11$. Si nos encontramos en los extremos de la cadena de C_2 y el movimiento que se nos indica es hacia un punto en el que no hay algo que leer, tenemos que concatenar la cadena 010, la cual corresponde a la codificación del símbolo "blanco".
   - Actualizamos el tercer elemento de la terna de C_2 con el número de unos de la codificación del símbolo que se está leyendo (el que quedó después de realizar el paso anterior).
   - Reemplazamos el contenido de C_3 con la codificación del nuevo estado, es decir, cod_{se}.

A continuación se realiza un comparativo paso a paso entre el funcionamiento de la máquina universal y el modelo estándar, tomando a C_1 como la codificación de la máquina que acepta palíndromos sobre el alfabeto {a, b}. Para esto hemos numerado con subíndices cada bloque:

$C_1 =$

$$01010110101110_0010110111010111010_1010111011111101011102_0111010111101010_3$$
$$0111011011101101110_4011101110111011101110_501111010110101110_6$$
$$0111101101111101010_701111101010101110_8011111011011111011010_9$$
$$0111110111011111011101010_10011111101011111110101011_1101111101101111110110111012_{12}$$
$$0111111011101111110111011101_13011111110101101011104_0111111101110111111010105_{15}$$

Máquina universal de Turing:

$$\begin{aligned}
(C_1, (011011101110110, 1, 2), 010) &\vdash \\
&\vdash (C_1, (01011101110110, 3, 3), 01110) \vdash \\
&\vdash (C_1, (01011101110110, 7, 3), 01110) \vdash \\
&\vdash (C_1, (01011101110110, 11, 2), 01110) \vdash \\
&\vdash (C_1, (0101110111011010, 14, 1), 01110) \vdash \\
&\vdash (C_1, (0101110111011010, 11, 2), 011110) \vdash \\
&\vdash (C_1, (010111011101010, 7, 3), 0111110) \vdash \\
&\vdash (C_1, (010111011101010, 3, 3), 0111110) \vdash \\
&\vdash (C_1, (010111011101010, 1, 1), 0111110) \vdash \\
&\vdash (C_1, (010111011101010, 3, 3), 010) \vdash \\
&\vdash (C_1, (0101011101010, 5, 3), 01111110) \vdash \\
&\vdash (C_1, (0101011101010, 9, 1), 01111110) \vdash \\
&\vdash (C_1, (0101011101010, 5, 3), 011111110) \vdash \\
&\vdash (C_1, (01010101010, 3, 1), 0111110) \vdash \\
&\vdash (C_1, (01010101010, 5, 1), 010) \vdash \\
&\vdash (C_1, (01010101010, 7, 1), 0110)
\end{aligned}$$

Máquina de Turing estándar:

$$\begin{aligned}
(q_0, \underline{a}bba) &\vdash \\
&\vdash (q_1, \flat\underline{b}ba) \vdash \\
&\vdash (q_1, \flat b\underline{b}a) \vdash \\
&\vdash (q_1, \flat bb\underline{a}) \vdash \\
&\vdash (q_1, \flat bba\underline{\flat}) \vdash \\
&\vdash (q_2, \flat bb\underline{a}\flat) \vdash \\
&\vdash (q_3, \flat b\underline{b}\flat\flat) \vdash \\
&\vdash (q_3, \flat\underline{b}b\flat\flat) \vdash \\
&\vdash (q_3, \underline{\flat}bb\flat\flat) \vdash \\
&\vdash (q_0, \flat\underline{b}b\flat\flat) \vdash \\
&\vdash (q_4, \flat\flat\underline{b}\flat\flat) \vdash \\
&\vdash (q_4, \flat\flat b\underline{\flat}\flat) \vdash \\
&\vdash (q_5, \flat\flat\underline{b}\flat\flat) \vdash \\
&\vdash (q_3, \flat\underline{\flat}\flat\flat\flat) \vdash \\
&\vdash (q_0, \flat\flat\underline{\flat}\flat\flat) \vdash \\
&\vdash (q_0, \flat\flat\flat\underline{\flat}\flat)
\end{aligned}$$

La respuesta de aceptación de la máquina universal sobre una cadena, depende de la respuesta de la máquina estándar sobre esa misma cadena. Cuando la máquina estándar acepte, la máquina universal también lo hará y viceversa.

5.2.1. Implementación del funcionamiento de la máquina universal de Turing

Primero definimos los tipos de cada una de las cintas y de la máquina universal de Turing:

```
type Cinta1 = Cadena
type Cinta2 = (Cadena, Int, Int)
type Cinta3 = Cadena

type MUT = (Cinta1, Cinta2, Cinta3)
```

Posteriormente implementamos una serie de funciones auxiliares para llevar a cabo el proceso tal y como se describió en la sección anterior. A continuación presentamos la implementación y descripción de estas funciones:

- La función *poscero* toma una cadena y regresa la posición del n-ésimo cero dentro de esta cadena recorriéndola de izquierda a derecha:

```
poscero :: String -> Int -> Int
poscero [] _ = 0
poscero _ 0 = 0
poscero (w:ws) n = if w == '0'
                   then 1 + poscero ws (n-1)
                   else 1 + poscero ws n
```

- La función *bloques* toma la imagen de una transición y la descompone por bloques. Manejamos un subíndice que indica el número de bloque que está siendo analizado. Por lo tanto 01010110101110_0 hace referencia al primer bloque de la codificación de la máquina *mtePal*. Por ejemplo, si consideramos 01010110101110_0, entonces la imagen sería 110101110 de modo que la función *bloques* devolvería $(11, 1, 111)$:

```
bloques :: String -> (String, String, String)
bloques [] = ("","","")
bloques ws = (take (n - 1) ws,
              take ((poscero (drop n ws) 1) - 1) (drop n ws),
              tail (reverse (drop (n + (poscero (drop n ws) 1)) ws)))
            where n = (poscero ws 1)
```

- La función *imagen* recorre la cadena almacenada en la primera cinta de izquierda a derecha observando cada uno de los bloques que la componen. Dado que un bloque contiene la codificación de una transición, la función *imagen* compara la primera parte de cada bloque, la cual consta de la codificación del estado actual y codificación del símbolo leído, con la cadena *cad* y devuelve la imagen del bloque que coincida con la cadena de entrada descompuesta en una terna de la forma (cod_{es}, cod_{se}, cod_m):

```
imagen :: String -> String -> (String, String, String)
imagen [] _ = ("","","")
```

```
imagen ct1 cad = if cad == (take (length cad) b)
                   then bloques(drop (length cad) b)
                   else imagen (drop (length b) ct1) cad
                 where b = take (poscero ct1 6) ct1
```

- La función *mueve* recibe una cadena w, un índice n y un indicador de movimiento m. Cuando m vale cero el movimiento será hacia la izquierda mientras que cuando su valor es uno, el movimiento será a la derecha. Nos movemos sobre w desde el n - ésimo caracter hasta encontrar la posición inicial de la codificación del elemento que encuentra a la izquierda o a la derecha de w dependiendo del valor de m:

```
mueve :: String -> Int -> Int -> Int
mueve [] _ _ = 0
mueve _ 0 0 = 1
mueve w n  m = if (m == 0)
               then (if w!!n == '0'
                     then (n+1)
                     else mueve w (n-1) 0)
               else (if w!!n == '0'
                     then (n+1)
                     else mueve w (n+1) 1)
```

Pasamos a las dos funciones centrales de esta sección, la función *deltaMUT* y la función *deltaEstrellaMUT* las cuales son análogas a las expuestas en la sección de la máquina de Turing estándar.

La función *deltaMUT* recibe la máquina universal con las tres cintas y la terna (cod_{es}, cod_{se}, cod_m), la cual será utilizada para hacer las sustituciones correspondientes en las cintas dos y tres. Los casos a distinguir en esta implementación son:

- Caso 1) Cuando el estado siguiente viene como nulo (“ ”) el contenido de la segunda cinta se queda igual y sólo se hace la sustitución en la tercera cinta por un estado nulo; esto significa que la cadena no fue aceptada.

- Caso 2)Cuando el movimiento es a la izquierda (1) se distinguen dos posibilidades: si nos encontramos en la codificación del primer símbolo de la cadena w al intentar movernos una codificación a la izquierda se producirá un desbordamiento, ya que no existe tal codificación, así que se procede a hacer los reemplazos correspondientes en la segunda y tercera cinta, tal y como se mencionó en la sección anterior y se concatena un cero junto con la codificación del símbolo blanco al principio de w para evitar el desbordamiento. Cuando estamos en cualquier otra posición, sólo procedemos a hacer las actualizaciones en ambas cintas.

- Caso 3)Cuando el movimiento es a la derecha (111) el proceso es análogo al caso 2, sólo que ahora se agregará la codificación del símbolo blanco cuando nos encontremos en la codificación del extremo derecho de w.

- Caso 4)Cuando no hay movimiento (11) no existe el riesgo de desbordamiento, así que sólo hacemos los reemplazos correspondientes.

La implementación de la función *deltaMUT* es la siguiente:

```
deltaMUT :: MUT -> (Cadena, Cadena, Cadena) -> (Cinta2, Cadena)
deltaMUT (c1, (w, p, l), e) (es, s, m)
      | (es == "") = ((w, p, l), "")
      | (m == "1") = case p of
            1 -> (("010" ++ s ++ drop (1+l) w, 1, 1), eds)
            _ -> ((nw,
                 mueve ((take p w) ++ s ++ (drop (p+l) w)) (p-2) 0,
                 p - (mueve nw (p-2) 0) - 1),
                 eds)

      | (m == "111") = case l' of
            1 -> (((take p w)++s++"010", p+(length s)+1, 1), eds)
            _ -> ((nw,
                 p + length s + 1,
                 (mueve nw (p + length s + 1) 1) - (p + length s + 2)),
                 eds)

      | otherwise = ((nw, p, length s), eds)

      where
            nw = (take p w) ++ s ++ (drop (p+l) w)
            l' = (length w) - (p+l)
           eds = "0" ++ es ++ "0"
```

La función *deltaEstrellaMUT* contempla dos casos base y el paso recursivo. En el primer caso base llegamos al estado final cuya codificación es 11, por lo que devolvemos *true*, ya que la cadena fue aceptada. En el segundo caso base no encontramos alguna transición que se le pueda aplicar a la máquina; para indicar este hecho usamos el estado *nulo* y en consecuencia devolvemos $false$, ya que la cadena no fue aceptada. El paso recursivo, se aplica cuando no hemos obtenido una respuesta definitiva de aceptación pero podemos continuar con el proceso a través de la función *deltaMUT*. La implementación de la función *deltaEstrellaMUT* se muestra a continuación:

```
deltaEstrellaMUT :: MUT -> Bool
deltaEstrellaMUT (c1, (w, p, l), "0110") = True
deltaEstrellaMUT (c1, (w, p, l), "")     = False
deltaEstrellaMUT (c1, (w, p, l), edo) =
              deltaEstrellaMUT (c1, (fst par), (snd par))
          where par = deltaMUT
                (c1, (w, p, l), edo)
                (imagen c1 (edo ++ take (l+1) (drop p w)))
```

La función objetivo de esta sección es la función *aceptaCadena* y se define de manera directa mandando llamar a la función *deltaEstrellaMUT*:

```
aceptaCadena :: MUT -> Bool
aceptaCadena mut = deltaEstrellaMUT mut
```

Para probar esta función consideremos las siguientes codificaciones:

- Cinta 1. Codificación de la máquina estándar que acepta palíndromos sobre el alfabeto $\{a, b\}$:

$$\text{cinta1} =$$
$$0101101110101110011101011110101001111010110101110011101110111011101110$$
$$0111011011101101110011110111011111010100111110111011111011110$$
$$0111110101010111001011101111110101110011111101011111101010$$
$$0111111101011010111001010110101110011111011011111011010$$
$$0111111011011111101101110011111101110111111011101110$$
$$01111111011101111101010$$

- Cinta 2. Codificación de la cadena "aba" y los índices apuntando a la codificación del primer símbolo de la cadena de entrada:

$$\text{cinta2} = (01101110110,\ 1,\ 2)$$

- Cinta 3. Codificación del estado inicial:

$$\text{cinta3} = 010$$

```
Main> aceptaCadena (cinta1,("01101110110", 1, 2),"010")
True
```

La función *pintaProcesoAceptacionMUT* nos permite observar el proceso de aceptación sobre una cadena paso a paso:

```
pintaProcesoAceptacionMUT :: MUT -> IO()
pintaProcesoAceptacionMUT (c1, (w, p, l), "0110") = putStrLn("|- True")
pintaProcesoAceptacionMUT (c1, (w, p, l), "")     = putStrLn("|- False")
pintaProcesoAceptacionMUT (c1, (w, p, l), edo)
               = putStrLn("|- Cod. Cinta 1")
              >> putStrLn("   Cinta 2: "++w)
              >> putStrLn("   Cinta 3: "++edo)
              >> pintaProcesoAceptacionMUT (c1, (fst par), (snd par))
 where par = deltaMUT
                (c1, (w, p, l), edo)
                (imagen c1 (edo ++ take (l+1) (drop p w)))
```

El resultado de ejecutar esta función con los parámetros anteriores es el siguiente:

```
Main> pintaProcesoAceptacionMUT (cinta1,("01101110110", 1, 2),"010")
|- Cod. Cinta 1
   Cinta 2: 01101110110
```

```
   Cinta 3: 010
|- Cod. Cinta 1
   Cinta 2: 0101110110
   Cinta 3: 01110
|- Cod. Cinta 1
   Cinta 2: 0101110110
   Cinta 3: 01110
|- Cod. Cinta 1
   Cinta 2: 010111011010
   Cinta 3: 01110
|- Cod. Cinta 1
   Cinta 2: 010111011010
   Cinta 3: 011110
|- Cod. Cinta 1
   Cinta 2: 01011101010
   Cinta 3: 0111110
|- Cod. Cinta 1
   Cinta 2: 01011101010
   Cinta 3: 0111110
|- Cod. Cinta 1
   Cinta 2: 01011101010
   Cinta 3: 010
|- Cod. Cinta 1
   Cinta 2: 010101010
   Cinta 3: 01111110
|- Cod. Cinta 1
   Cinta 2: 010101010
   Cinta 3: 011111110
|- True
```

Se omitió la codificación completa de la máquina en la primera cinta ya que ésta es constante durante todo el proceso, y en su lugar se utilizó la etiqueta "Cod. Cinta 1".

5.3 Decodificación

El proceso de *decodificación* esencialmente consiste en tomar una cadena de ceros y unos, que es la codificación de una máquina de Turing estándar, y regresarla a su formato original.

Para la decodificación de estados utilizamos la etiqueta "q", dado que ya sabemos que la codificación del estado inicial y la codificación del estado final es "1" y "11" respectivamente, asignamos a esta codificación los estados con las etiquetas "q0" y "qf" en el proceso de decodificación, mientras que para cualquier otra codificación de un estado, definimos la decodificación añadiendo a la etiqueta "q" un índice que resulta de restarle dos unidades a la longitud de la codificación.

En cuanto a la decodificación de las cadenas que representan símbolos vamos a usar las letras del abecedario. Sabemos que la cadena "1" representa la codificación del símbolo blanco, por lo que la cadena "11" será decodificada como 'a', la cadena "111" como 'b' y así sucesivamente.

Cabe aclarar que si tomamos una máquina de Turing estándar y le aplicamos los procesos de codificación y enseguida el de decodificación, vamos a obtener una máquina que en esencia resuelve la misma tarea, pero no significa que sea una copia fiel de la máquina original. Ya que:

- El proceso de decodificación fija q_0 y q_f para designar el estado inicial y el estado final respectivamente.
- El único símbolo que con seguridad será respetado es el *símbolo blanco* ya que los demás se asignan tomando las letras del abecedario en orden. Por ejemplo, si utilizamos una máquina que acepte palíndromos un poco más complejos como "anitalavalatina" necesitaríamos un alfabeto de al menos siete símbolo: ♭, a, i, l, n, t, v y si son introducidos en ese orden al proceso de codificación y posteriormente al de decodificación, tendríamos lo siguiente:

```
Codificacion:              Decodificacion:

    '_' -> "1"                 "1"       -> '_'
    'a' -> "11"                "11"      -> 'a'
    'i' -> "111"               "111"     -> 'b'
    'l' -> "1111"              "1111"    -> 'c'
    'n' -> "11111"             "11111"   -> 'd'
    't' -> "111111"            "111111"  -> 'e'
    'v' -> "1111111"           "1111111" -> 'f'
```

  por lo que al final tendríamos una máquina que en lugar de aceptar "anitalavalatina" aceptaría "adbeacafacaebda". Esto significa que el lenguaje aceptado por la máquina resultante será isomorfo[1] al lenguaje aceptado por la máquina original. Ante esto, lo adecuado es definir una máquina que acepta palíndromos sobre un alfabeto de siete símbolos cualesquiera que estos sean, evitando así la pérdida de información.

5.3.1. Implementación del proceso de decodificación

La decodificación revierte el proceso de codificación en el mismo orden en el que se hizo este último dividiendo primero una cadena de ceros y unos que representa a una máquina de Turing estándar, en bloques de la forma:

$$0f_c(ea)0f_c(sl)0f_c(es)0f_c(se)0f_c(m)0$$

[1] El isomorfismo de lenguajes puede consultarse en los libros "An Introduction to Formal Languages and Automata"[1], "Introduction to the Theory of Computation"[2] y "Lenguajes Formales y Teoría de la Computación"[3]

Esto se hace extrayendo la subcadena delimitada por el primero y sexto cero, obteniendo un bloque de la cadena e iterando sobre el resto de la codificación. Se toma cada uno de los bloques y se descompone en cinco subbloques diferenciando el estado actual, el símbolo leído, el estado siguiente, el símbolo escrito y la dirección del movimiento.

La primera función que vamos a implementar decodifica a los distintos movimientos:

```
decodificaMovimientos :: Cadena -> Movimiento
decodificaMovimientos "1" = I
decodificaMovimientos "11" = N
decodificaMovimientos "111" = D
```

Para descomponer los bloques utilizamos la función *obtenBloques* que en sí, es una extensión de la función *bloques* mostrada en la sección anterior:

```
obtenBloques :: String -> (String, String, String, String, String)
obtenBloques [] = ("","","","","")
obtenBloques ws = (take (a-1) ws,
        take (b-(a+1)) (drop a ws),
        take (c-(b+1)) (drop b ws),
        take (d-(c+1)) (drop c ws),
        (drop d ws))
   where
        a = poscero ws 1
        b = poscero ws 2
        c = poscero ws 3
        d = poscero ws 4
```

Cada uno de los elementos obtenidos en la tupla representa la codificación de un elemento de alguna transición. La función *procesaBloques* es la encargada de procesar estas codificaciones devolviéndonos los elementos decodificados. En la implementación de la función *procesaBloques* se utilizan las instrucciones:

```
map show [1..]

Hugs> take 10 (map show [1..])
["1","2","3","4","5","6","7","8","9","10"]
```

y

```
['a'..]

Hugs> take 10 ['a'..]
"abcdefghij"
```

En ambas se observa la importancia de la evaluación perezosa, ya que nuevamente no nos tenemos que preocupar por implementar funciones para manipular los resultados obtenidos y simplemente hacemos uso de ellos. La implementación de la función *procesaBloques* se muestra a continuación:

```
procesaBloques :: (Cadena, Cadena, Cadena, Cadena, Cadena)
                  ->
                  ((Estado, Simbolo), (Estado, Simbolo, Movimiento))
procesaBloques (s1, s2, s3, s4, s5) =
                                      (
                                      (estados!!(length s1),
                                       simbolos!!(length s2)),
                                      (estados!!(length s3),
                                       simbolos!!(length s4),
                                       decodificaMovimientos s5)
                                       )
where estados = ["q0", "qf"] ++ ["q"++indice | indice <- (map show [1..])]
  simbolos = ['_'] ++ [letra | letra <- ['a'..]]
```

Concluimos esta sección con la implementación de la función *decodifica*. Esta función recibe una cadena de ceros y unos y genera una máquina de Turing estándar, la cual quedará definida por la decodificación de las transiciones. Lo primero que se hace es verificar que la cadena corresponda a la codificación de una máquina, en caso de que no sea así, devolvemos una máquina que no hace nada. Para esto declaramos el siguiente tipo:

```
type TransicionesDec = [((Estado,Cadena), (Estado,Simbolo,Cadena))]
```

A continuación mostramos la implementación de la función *decodifica*:

```
decodifica :: Cadena -> TransicionesDec
decodifica [] = []
decodifica cad =
if (((validaBloque (obtenBloques (take (n-2) (tail cad))) /= True)
|| (validaSimbolos (take (n-2) (tail cad)) /= True)) && cad /= [])
    then [(("q0", ' '), ("  ", ' ', N))]
else (procesaBloques (obtenBloques (take (n-2) (tail cad))))
    :decodifica (drop n cad)
where n = poscero cad 6
```

Para terminar, asociamos una vista que presente los resultados de la función $Transiciones\ Dec$ con formato, por medio de la función *pintaTransiciones*:

```
pintaTransiciones :: TransicionesDec -> String
pintaTransiciones [] = ""
pintaTransiciones (((ea,sa),(es,ss,m)):ms) =
                    ea ++ "  '" ++ [sa] ++ "'  " ++ "="
                    ++ "  " ++ es ++ "  '" ++ [ss] ++ "'  " ++
                    show m ++ "\n" ++ pintaTransiciones ms

instance Show TransicionesDec where
         show transicionesDec = pintaTransiciones transicionesDec
```

Podemos probar estas funciones sobre la máquina de Turing estándar que acepta palíndromos sobre el alfabeto $\{a, b\}$ ejecutando la siguiente instrucción:

```
Main> decodifica (codifica mte)
      q0  '_'  =  qf  '_'  D
      q0  'a'  =  q1  '_'  D
      q0  'b'  =  q4  '_'  D
      q1  '_'  =  q2  '_'  I
      q1  'a'  =  q1  'a'  D
      q1  'b'  =  q1  'b'  D
      q2  '_'  =  qf  '_'  D
      q2  'a'  =  q3  '_'  I
      q3  '_'  =  q0  '_'  D
      q3  'a'  =  q3  'a'  I
      q3  'b'  =  q3  'b'  I
      q4  '_'  =  q5  '_'  I
      q4  'a'  =  q4  'a'  D
      q4  'b'  =  q4  'b'  D
      q5  '_'  =  qf  '_'  D
      q5  'b'  =  q3  '_'  I
```

6 Conclusiones

En este trabajo se han modificado ligeramente las definiciones clásicas de la máquina de Turing. Estas modificaciones sólo han sido de forma y no de fondo, por lo que las implementaciones realizadas corresponden de manera fiel a las definiciones matemáticas originales, explotando aquellas características de Haskell que le dan un valor agregado sobre otros lenguajes cuando pretendemos abordar temas correspondientes a módelos matemáticos.

En resumen, la evaluación perezosa, las listas por comprensión, el paradigma funcional, el manejo y creación de tipos fueron las características de Haskell que hicieron posible el desarrollo de los módulos expuestos en este trabajo, ya que:

- La evaluación perezosa hace que tenga sentido intentar obtener el lenguaje aceptado, el lenguaje generado y el lenguaje traducido por una máquina de Turing, sin necesidad de implementar funciones auxiliares para manipular estos lenguajes aunque no sean finitos. Aunque es claro que si pretendemos observar el comportamiento de una máquina sobre cada una de las cadenas de un lenguaje no finito, el proceso no terminaría.
- El mecanismo utilizado por las listas por comprensión ha facilitado mucho el trabajo dado que nos permite implementar acciones con tan sólo indicarle a Haskell qué es lo que debe hacer, sin necesidad de indicarle cómo lo tiene que hacer. La ventaja principal es que en Haskell las listas por comprensión se pueden manipular como simples listas, permitiendo tener una herramienta poderosa en un tipo de dato básico.
- Para crear los tipos de datos requeridos en la máquina de Turing no hicimos más que declararlos de manera directa, lo cual da como resultado el tener un código compacto, elegante e intuitivo.

Por último, vale la pena resaltar que en este trabajo teníamos dos objetivos principales los cuales consideramos han sido cumplidos: crear una herramienta que sirva como apoyo a la docencia en temas correspondientes a la máquina de Turing y hacer hincapié en el poder de Haskell si éste se utiliza de forma adecuada.

A Simulación de autómatas por medio de la máquina de Turing

En este apartado cubriremos los temas de simulación de un autómata finito determinista (AFD) y simulación de un autómata de pila sin transiciones ε (AP) con máquinas de Turing. Vale la pena mencionar que ambas simulaciones son de aplicación general, ya que todo lenguaje regular puede ser reconocido por un autómata finito determinista y todo lenguaje libre de contexto puede ser reconocido por un autómata de pila sin transiciones ε.
La correspondencia entre autómatas y máquinas de Turing es directa ya que básicamente contienen los mismos elementos. Ambas cuentan con un estado inicial, al menos un estado final, un conjunto de estados, un alfabeto y una función de transición, así que para manejar los autómatas usaremos la misma notación. Además, manejaremos un sólo estado final en los autómatas, tomando en cuenta que cualquier autómata con más de un estado final puede transformarse a uno que cuente con un único estado final.

A.1 Autómata finito determinista

Recordemos la definición de un autómata finito determinista:

Definición A.1.1 (Autómata finito determinista (AFD))**.** Un autómata finito determinista es una 5-tupla $T = (Q, F, \Sigma, q_0, \delta)$ en donde:

- Q es un conjunto de estados finitos, $Q \neq \varnothing$.
- F es un conjunto de estados finales con $F \subseteq Q$, $F \neq \varnothing$.
- Σ es un conjunto finito de símbolo de entrada, $\Sigma \neq \varnothing$.
- q_0 estado inicial con $q_0 \in Q$.
- δ es la función (parcial) sobre la cual se rige el autómata con $\delta : Q \times \Sigma \longrightarrow Q$.

Falta definir la función de transición para la máquina de Turing en relación con la función δ del autómata. Para diferenciar los elementos entre un modelo y otro, colocaremos etiquetas, de tal forma que Q_{MTE} hará referencia a los estados de la máquina de Turing mientras que Q_{AFD} hará referencia a los estados del autómata; lo mismo sucede con el resto de los elementos.

Definición A.1.2. La máquina de Turing estándar $T = (Q_{MTE}, F_{MTE}, \Sigma_{MTE}, \Gamma_{MTE}, q_{0_{MTE}}, \delta_{MTE})$ asociada al autómata finito determinista $A = (Q_{AFD}, F_{AFD}, \Sigma_{AFD}, q_{0_{AFD}}, \delta_{AFD})$ tal que $L(T) = L(A)$, se construye como sigue:

- $Q_{MTE} = Q_{AFD} \cup \{q_f\}$.
- $F_{MTE} = q_f$.
- $\Sigma_{MTE} = \Sigma_{AFD}$.
- $\Gamma_{MTE} = \Sigma_{AFD}$.
- $q_{0_{MTE}} = q_{0_{AFD}}$.
- Para cada transición $\delta_{AFD}\ (q, a) = p$ en donde $q \notin F_{AFD}$ definida en el autómata, entonces, tendremos la transición $\delta_{MTE}\ (q, a) = (p, \flat, D)$ en la máquina de Turing. Para contemplar las transiciones hacia el estado final, sólo agregamos:

  $$\delta_{MTE}\ (q, F) = (qf, \flat, N)$$

  Esta transición es el resultado de la estrategia que estamos usando. Cuando vamos a examinar una cadena con una máquina de Turing, la cual fue resultado de un autómata, le concatenamos el símbolo F al final como un delimitador de la misma. De modo que cuando estemos leyendo el símbolo F y nos encontremos en un estado final, significa que hemos recorrido toda la cadena y por consiguiente, ésta es aceptada por la máquina.

Para ejemplificar esta definición tomemos el autómata A que acepta el lenguaje $L \subseteq \{0, 1\}^*$ de cadenas con un número par de ceros y un número impar de unos, cuya definición es $A = (\{q_0, q_1, q_2, q_3\}, \{q_3\}, \{0, 1\}, q_0, \delta_A)$ en donde:

$$\begin{aligned}
\delta_A(q_0, 0) &= q_1 \\
\delta_A(q_0, 1) &= q_3 \\
\delta_A(q_1, 0) &= q_0 \\
\delta_A(q_1, 1) &= q_2 \\
\delta_A(q_2, 0) &= q_3 \\
\delta_A(q_2, 1) &= q_1 \\
\delta_A(q_3, 0) &= q_2 \\
\delta_A(q_3, 1) &= q_0
\end{aligned}$$

Por definición la máquina de Turing estándar T asociada al autómata A es la tupla $(\{q_0, q_1, q_2, q_3\}, \{q_3\}, \{0, 1\}, \{0, 1\}, q_0, \delta_T)$, en donde:

$$\begin{aligned}
\delta_T(q_0, 0) &= (q_1, \flat, D) \\
\delta_T(q_0, 1) &= (q_3, \flat, D) \\
\delta_T(q_1, 0) &= (q_0, \flat, D) \\
\delta_T(q_1, 1) &= (q_2, \flat, D) \\
\delta_T(q_2, 0) &= (q_3, \flat, D) \\
\delta_T(q_2, 1) &= (q_1, \flat, D) \\
\delta_T(q_3, 0) &= (q_2, \flat, D) \\
\delta_T(q_3, 1) &= (q_0, \flat, D) \\
\delta_T(q_3, F) &= (q_f, \flat, N)
\end{aligned}$$

El código de este módulo se encuentra en el archivo "AFDMTE.hs". Como veremos más adelante al ejecutar este módulo se envia la definición de la máquina de Turing relacionada con el autómata a un fichero ".hs" para que podamos utilizar las funciones del módulo de programas de la MTE sobre esta máquina.

Consideremos el autómata $afdPar0Impar1$ que acepta cadenas con un número par de ceros y un número impar de unos, cuya definición se encuentra en el archivo "EjemplosAFD.hs".

```
afPar0Impar1 = AF ["q0", "q1", "q2", "q3"]
                  "q0"
                  "q3"
                 ['0', '1']

afdPar0Impar1 = (afPar0Impar1, d) where
                d "q0" '0'  =  "q1"
                d "q0" '1'  =  "q3"
                d "q1" '0'  =  "q0"
                d "q1" '1'  =  "q2"
                d "q2" '0'  =  "q3"
                d "q2" '1'  =  "q1"
                d "q3" '0'  =  "q2"
                d "q3" '1'  =  "q0"
                d _ _ = "qr"
```

Al ejecutar la función $convierteAFD$ del archivo "AFDMTE.hs" sobre el autómata $afdPar0Impar1$ se obtiene la siguiente MTE:

```
mtAFD = MT ["q0", "q1", "q2", "q3", "qf"]
           "q0"
           "qf"
          ['0', '1']
          ['_', '0', '1']
```

```
mteAFD = (mtAFD, d) where
          d "q0" '0' = ("q1", '_', D)
          d "q0" '1' = ("q3", '_', D)
          d "q1" '0' = ("q0", '_', D)
          d "q1" '1' = ("q2", '_', D)
          d "q2" '0' = ("q3", '_', D)
          d "q2" '1' = ("q1", '_', D)
          d "q3" '0' = ("q2", '_', D)
          d "q3" '1' = ("q0", '_', D)
          d "q3" 'F' = ("qf", '_', N)
          d _ _ = ("qr", 'E', D)
```

A.2 Autómata de pila sin transiciones ε

Para construir una máquina de Turing a partir de un autómata de pila sin transiciones ε, vamos a utilizar una máquina multicinta. Esta máquina constará de dos cintas: en la primera colocaremos la cadena que se va a estudiar, mientras que la segunda hará el papel de la pila del autómata.

Definición A.2.1 (Autómata de pila (AP))**.** Un autómata de pila es una 7-tupla $T = (Q, F, \Sigma, \Gamma, q_0, Z_0, \delta)$ en donde:

- Q es un conjunto de estados finitos, $Q \neq \varnothing$.
- F es un conjunto de estados finales con $F \subseteq Q$, $F \neq \varnothing$.
- Σ es un conjunto finito de símbolo de entrada, $\Sigma \neq \varnothing$.
- Γ es un conjunto finito de símbolo de la pila, $\Gamma \neq \varnothing$.
- q_0 estado inicial con $q_0 \in Q$.
- Z_0 símbolo inicial de la pila con $Z_0 \in \Gamma$.
- $\delta : Q \times (\Sigma \cup \{\flat\}) \times \Gamma \longrightarrow$ el conjunto de subconjuntos finitos de $Q \times \Gamma^*$.

En el proceso de construcción de la máquina de Turing relacionada con un autómata de pila sin transiciones ε se contemplan las tres operaciones que se pueden realizar en la pila del autómata, es decir, no alterar la pila, meter algún símbolo en la pila o sacar un símbolo de la misma.

Definición A.2.2. La máquina de Turing multicinta $T = (Q_{MTMC}, F_{MTMC}, \Sigma_{MTMC}, \Gamma_{MTMC}, q_{0_{MTMC}}, \delta_{MTMC})$, asociada a un autómata de pila $A = (Q_{AP}, F_{AP}, \Sigma_{AP}, \Gamma_{AP}, q_{0_{AP}}, Z_0, \delta_{AP})$ tal que $L(T) = L(A)$, se construye como sigue:

- $Q_{MTMC} = Q_{AP} \cup \{q_f\}$, tal que $q_f \notin Q_{AP}$.

- $F_{MTMC} = \{q_f\}$.
- $\Sigma_{MTMC} = \Sigma_{AP}$.
- $\Gamma_{MTMC} = \Gamma_{AP}$.
- $q_{0_{MTMC}} = q_{0_{AP}}$.
- Sea P la pila sobre la que trabaja el autómata A. Los casos a considerar son los siguientes:
  - Sí P está vacía, no es posible ningún movimiento.
  - No alteramos P. Para cada transición de la forma $\delta_{AP}(q, s_e, s_p) = (q', s_p)$ definida en A, entonces, tendremos la transición $\delta_{MTMC}\ (q, [s_e, s_p]) = (q', [(\flat, D), (s_p, N)])$ en la máquina de Turing.
  - Sacamos un símbolo de P. Para cada transición de la forma $\delta_{AP}(q, s_e, s_p) = (q', \varepsilon)$ definida en A donde s_p es el tope de P, entonces, tendremos la transición δ_{MTMC} $(q, [s_e, s_p]) = (q',\ [(\flat, D),\ (\flat, D)])$ en la máquina de Turing.
  - Metemos un símbolo en P. Para cada transición de la forma $\delta_{AP}(q, s_e, s_p) = (q', s_n s_p)$ definida en A donde s_p es el tope de P, entonces, tenemos que agregar las siguientes transiciones a la máquina de Turing:
    - $\delta_{MTMC}\ (q, [s_e, s_p]) = (q, [(\flat, N), (s_p, I)])$.
    - $\delta_{MTMC}\ (q, [\flat, \flat]) = (q', [(\flat, D), (s_n, N)])$.

  Por último agregamos las transiciones correspondientes hacia q_f:

$$\delta_{MTMC}\ (q, [F, x]) = (q_f, [(\flat, N), (\flat, N)]), \text{ para cada } q \in Q_{AP} \text{ con } x \in \Gamma_{AP}$$

Como ejemplo, consideremos el autómata de pila A que acepta el lenguaje:

$$L = \{w \in \{a, b\}^* \mid n_a(w) > n_b(w)\ \}$$

cuya definición es $A = (\{q_0, q_1\},\ \{q_1\},\ \{a, b\},\ \{a, b, Z_0\},\ q_0,\ Z_0,\ \delta_A$ en donde:

$$\begin{aligned}
\delta_A(q_0, a, Z_0) &= (q_1, Z_0) \\
\delta_A(q_0, b, Z_0) &= (q_0, bZ_0) \\
\delta_A(q_0, a, b) &= (q_0, \flat) \\
\delta_A(q_0, b, b) &= (q_0, bb) \\
\delta_A(q_1, a, Z_0) &= (q_1, aZ_0) \\
\delta_A(q_1, b, Z_0) &= (q_0, Z_0) \\
\delta_A(q_1, a, a) &= (q_1, aa) \\
\delta_A(q_1, b, a) &= (q_1, \flat)
\end{aligned}$$

Por definición la máquina de Turing asociada al autómata A es la tupla ($\{q_0, q_1, q_f\}$, $\{q_f\}$, $\{a, b\}$, $\{a, b, Z\}$, q_0, δ_T), en donde:

$$\begin{aligned}
\delta_{MTMC}(q_0, [a, Z]) &= (q_1, [(\flat, D), (Z, N)]) \\
\delta_{MTMC}(q_0, [b, Z]) &= (q_0, [(\flat, N), (Z, I)]) \\
\delta_{MTMC}(q_0, [\flat, \flat]) &= (q_0, [(\flat, D), (b, N)]) \\
\delta_{MTMC}(q_0, [a, b]) &= (q_0, [(\flat, D), (\flat, D)]) \\
\delta_{MTMC}(q_0, [b, b]) &= (q_0, [(\flat, N), (b, I)]) \\
\delta_{MTMC}(q_0, [\flat, \flat]) &= (q_0, [(\flat, D), (b, N)]) \\
\delta_{MTMC}(q_1, [a, Z]) &= (q_1, [(\flat, N), (Z, I)]) \\
\delta_{MTMC}(q_1, [\flat, \flat]) &= (q_1, [(\flat, D), (a, N)]) \\
\delta_{MTMC}(q_1, [b, Z]) &= (q_0, [(\flat, D), (Z, N)]) \\
\delta_{MTMC}(q_1, [a, a]) &= (q_1, [(\flat, N), (a, I)]) \\
\delta_{MTMC}(q_1, [\flat, \flat]) &= (q_1, [(\flat, D), (a, N)]) \\
\delta_{MTMC}(q_1, [b, a]) &= (q_1, [(\flat, D), (\flat, D)]) \\
\delta_{MTMC}(q_1, [F, x]) &= (q_f, [(\flat, N), (\flat, N)])
\end{aligned}$$

El código de este módulo se encuentra en el archivo "APMTMC.hs"

Consideremos el autómata $apfMasAsqueBs$ que acepta cadenas con más letras a que letras b, cuya definición se encuentra en el archivo "EjemplosAPMTMC.hs".

```
afMasAsqueBs = AF ["q0", "q1"]
                  "q0"
                  "q1"
                 ['a', 'b']
                 ['Z', 'a', 'b']

apfMasAsqueBs = (afMasAsqueBs, d) where
                d "q0" 'a' 'Z'  =  ("q1", ['Z'])
                d "q0" 'b' 'Z'  =  ("q0", ['b', 'Z'])
                d "q0" 'a' 'b'  =  ("q0", [])
                d "q0" 'b' 'b'  =  ("q0", ['b', 'b'])
                d "q1" 'a' 'Z'  =  ("q1", ['a', 'Z'])
                d "q1" 'b' 'Z'  =  ("q0", ['Z'])
                d "q1" 'a' 'a'  =  ("q1", ['a', 'a'])
                d "q1" 'b' 'a'  =  ("q1", [])
                d _ _ _  =  ("qr", [])
```

Al ejecutar la función *convierteAP* del archivo "APMTMC.hs" sobre el autómata $apf\ MasAsqueBs$ se obtiene la siguiente $MTMC$:

```
mtAP = MT [ "q0", "q1", "qf"]
```

```
            "q0"
            "qf"
          [ 'a', 'b']
          [ '_', 'Z', 'a', 'b']

mtmcAP = (mtAP, d) where
          d "q0" ['a','Z'] = ("q1", [('_', D), ('Z', N)])
          d "q0" ['a','b'] = ("q0", [('_', D), ('_', D)])
          d "q0" ['b','Z'] = ("q0", [('_', N), ('Z', I)])
          d "q0" ['_', '_'] = ("q0", [('_', D), ('b', N)])
          d "q0" ['b','b'] = ("q0", [('_', N), ('b', I)])
          d "q0" ['_', '_'] = ("q0", [('_', D), ('b', N)])
          d "q1" ['a','Z'] = ("q1", [('_', N), ('Z', I)])
          d "q1" ['_', '_'] = ("q1", [('_', D), ('a', N)])
          d "q1" ['a','a'] = ("q1", [('_', N), ('a', I)])
          d "q1" ['_', '_'] = ("q1", [('_', D), ('a', N)])
          d "q1" ['b','Z'] = ("q0", [('_', D), ('Z', N)])
          d "q1" ['b','a'] = ("q1", [('_', D), ('_', D)])
          d "q1" ['F', _] = ("qf",[('_',N),('_',N)])
          d _ _ = ("qr",[('E',N),('E',N)])
```

B Utilización de los módulos implementados

Con el presente documento se adjuntan quince archivos de los cuales siete corresponden a los códigos fuente de los programas presentados en este trabajo, mientras que los ocho restantes contienen ejemplos de cada uno de los módulos programados. A continuación se listan los nombres de los archivos con una breve descripción de sus contenidos:

1. Códigos fuente:

   a) ModeloMTE.hs − contiene las definiciones de las funciones *lenguajeAceptado*, *lenguajeTraducido* y *lenguajeGenerado* utilizando una máquina de Turing estándar. También contiene las definiciones de las funciones que nos muestran paso a paso la ejecución de los procesos de aceptación y traducción de una cadena: *pintaProcesoAceptacion* y *pintaProcesoTraduccion*.

   b) ModeloMTMP.hs − contiene las definiciones de las funciones *lenguajeAceptado* y *pintaProcesoAceptacion* utilizando una máquina de Turing multipista.

   c) ModeloMTMC.hs − contiene las definiciones de las funciones *lenguajeAceptado* y *pintaProcesoAceptacion* utilizando una máquina de Turing multicinta.

   d) ModeloMTND.hs − contiene las definiciones de las funciones *lenguajeAceptado* y *pintaProcesoAceptacion* utilizando una máquina de Turing no determinista.

   e) MUT.hs − contiene las definiciones de las funciones correspondientes a los módulos de codificación, funcionamiento y decodificación de la máquina universal de Turing.

   f) AFDMTE.hs − contiene las definiciones para la función *convierteAFD* que convierte un autómata finito determinista en una máquina de Turing estándar.

   g) APMTMC.hs − contiene las definiciones para la función *convierteAP* que convierte un autómata de pila sin transiciones ε en una máquina de Turing multicinta.

2. Ejemplos:

   a) EjemplosModeloMTELA.hs − contiene ejemplos de máquinas de Turing estándar para el proceso de aceptación de lenguajes.

   b) EjemplosModeloMTELT.hs − contiene ejemplos de máquinas de Turing estándar para el proceso de traducción de lenguajes.

c) EjemplosModeloMTELG.hs − contiene ejemplos de máquinas de Turing estándar para el proceso de generación de lenguajes.

d) EjemplosModeloMTMP.hs − contiene ejemplos de máquinas de Turing multipista para el proceso de aceptación de lenguajes.

e) EjemplosModeloMTMC.hs − contiene ejemplos de máquinas de Turing multicinta para el proceso de aceptación de lenguajes.

f) EjemplosModeloMTND.hs − contiene ejemplos de máquinas de Turing no deterministas para el proceso de aceptación de lenguajes.

g) EjemplosAFD.hs − contiene ejemplos de autómatas finitos deterministas a los cuales se les puede aplicar la función $convierteAFD$ para obtener la máquina de Turing estándar relacionada con estos autómatas.

h) EjemplosAP.hs − contiene ejemplos de autómatas de pila a los cuales se les puede aplicar la función $convierteAP$ para obtener la máquina de Turing multicinta relacionada con estos autómatas.

Para probar cada uno de los módulos se tiene que iniciar el intérprete de HASKELL con la intrucción:

$$hugs\ -98\ +o$$

para que nos permita usar extensiones $Hugs/Ghc$ (-98) y para poder sobrecargar ejemplares de la clase $Show$ $(+o)$.

B.1 Máquina de Turing estándar

Para ejecutar ejemplos del modelo estándar primero cargamos el archivo "ModeloMTE.hs" y posteriormente el archivo que contiene la definición particular de alguna máquina estándar. Este archivo se pueden cargar con la instrucción:

$$ModeloMTE > : also\ Nombredelarchivo.hs$$

Por ejemplo, en el archivo "EjemplosModeloMTELA.hs" tenemos la definición de $mteAnBn$, una máquina de Turing estándar, tal que:

$$L(mteAnBn) = \{w \mid w = a^n b^n\ n > 0\}.$$

Una vez cargado el archivo "EjemplosModeloMTELA.hs" podemos utilizar la máquina $mteAnBn$ ejecutando la siguiente instrucción:

```
Main> take 5 (lenguajeAceptado mteAnBn)
```

obteniendo como resultado:

```
["ab","aabb","aaabbb","aaaabbbb","aaaaabbbbb"]
```

o bien:

```
Main> pintaProcesoAceptacion mteAnBn "aabb"
```

obteniendo como resultado:

```
|-(q0)aabb
|-X(q1)abb
|-Xa(q1)bb
|-X(q2)aYb
|-(q2)XaYb
|-X(q0)aYb
|-XX(q1)Yb
|-XXY(q1)b
|-XX(q2)YY
|-X(q2)XYY
|-XX(q0)YY
|-XXY(q3)Y
|-XXYY(q3)_
|-XXYY_(qf)_
|-True
```

En este mismo archivo tenemos las definiciones de las siguientes máquinas de Turing estándar para el lenguaje aceptado:

- $mteAnBn$:

$$L(mteAnBn) = \{w \mid w = a^n b^n,\ n > 0\}.$$

- $mtePar0$:

$$L(mtePar0) = \{w \mid w \in \{0,\ 1\}^+,\ n_0(w) = 2m\ \}.$$

  en donde, $n_0(w)$ devuelve el número de ceros que contiene la cadena w y $m \in \mathbb{N}$.

- $mteABA$:

$$L(mteABA) = \{w \mid w \in \{a,\ b\}^*,\ w = w_1\ aba\ w_2\ \}.$$

  en donde, w_1, w_2 $\{a, b\}^*$.

- $mtePal$:

$$L(mtePal) = \{w \mid w \in \{a,\ b\}^*,\ Pal(w)\ \}.$$

  en donde, $Pal(w)$ indica que la cadena w cumple con ser palíndromo.

- $mteSS$:

$$L(mteSS) = \{w \mid w = ss,\ s \in \{a,\ b\}^+\ \}.$$

- *mteAnBnCn*:

$$L(mteAnBnCn) = \{w \mid w = a^n b^n c^n,\ n > 0\}.$$

Para el modelo estándar también contamos con el archivo "EjemplosModeloMTELT.hs", con el cual podemos probar las funciones $lenguajeTraducido$ y $pintaProcesoTraduccion$ con las siguientes máquinas:

- *mtF2n* computa la función:

$$f(w^n) = w^{2n}$$

  en donde $w \in \{1\}^+$ y $n \in \mathbb{N}$.

- *mteReversa* computa la función:

$$f(w) = w^R$$

  en donde $w \in \{a, b\}^+$ y $\mid w \mid = 2n + 1$, $n \geq 0$.

- *mteReemplaza* computa la función:

$$f(a) = b\ f(b) = a$$

  dada $w \in \{a, b\}^+$, reemplazamos cada símbolo de w cambiando a por b y viceversa.

- *mteIncrementa* computa la función:

$$f(n_B) = n_B + 1$$

  incrementa en uno a un número binario n_B.

- *mteMod2* computa la función:

$$f(n) = n\ mod\ 2$$

  en donde, $n \in \{1\}^+$ y el valor de n como número queda determinado por $\mid n \mid$.

- *mteSuma* computa la función:

$$f(w_1 0 w_2) = w_1 w_2$$

  en donde, $w_1 \in \{1\}^+$ y $w_2 \in \{1\}^*$.

Por último tenemos el archivo "EjemplosModeloMTELG.hs", con el cual podemos probar la función *lenguajeGenereado* con la máquina *mteOrdLex* que genera cadenas en $\{0, 1\}^*$ respetando el orden lexicográfico. Recordemos que el proceso de generación de lenguajes agrega la cadena que hay en la cinta cada vez que regresamos al estado inicial, lo que da lugar a que tengamos cadenas repetidas. Por lo que esta función la podemos ejecutar con la instrucción:

$$quitaCadenasRepetidas\ (take\ n\ (lenguajeGenerado\ mteOrdLex))$$

B.2 Variantes de la máquina de Turing

Para cada una de las variantes tenemos dos archivos, uno que contiene el código fuente de las definiciones para las funciones *lenguajeAceptado* y *pintaProcesoAceptacion*, mientras que el otro es un archivo con un ejemplo de cómo debe definirse una máquina en particular. El archivo "EjemplosModeloMTMP.hs" contiene ejemplos de la variante multipista. Una vez que se haya cargado el archivo podemos ejecutar las funciones ya mencionadas de manera similar al modelo estándar. La máquina de ejemplo para esta variante es *mtmpAnBn*, tal que:

$$L(mtmpAnBn) = \{w \mid w = a^n b^n,\ n > 0\}.$$

mtmpAnBn utiliza dos pistas. La diferencia con las instrucciones que se ejecutaban para el modelo estándar radica en que ahora tenemos que indicarle a las funciones el número de pistas (o cintas) que maneja la máquina. Por ejemplo, si ejecutamos:

```
Main> take 6 (lenguajeAceptado mtmpAnBn 2)
```

obtenemos:

```
["ab","aabb","aaabbb","aaaabbbb","aaaaabbbbb","aaaaaabbbbbb"]
```

o bien:

```
Main> pintaProcesoAceptacion mtmpAnBn "aabb" 2
```

obteniendo como resultado:

```
|- q0  (a)abb  (_)___
|- q1  a(a)bb  *(_)__
|- q1  aa(b)b  *_(_)_
|- q2  a(a)bb  *(_)*_
|- q2  (a)abb  (*)_*_
|- q0  a(a)bb  *(_)*_
|- q1  aa(b)b  **(*)_
|- q1  aab(b)  ***(_)
|- q2  aa(b)b  **(*)*
|- q2  a(a)bb  *(*)**
|- q0  aa(b)b  **(*)*
|- q3  aab(b)  ***(*)
|- q3  aabb(_)  ****(_)
|- qf  aab(b)_  ***(*)_
|- True
```

En cuanto al modelo multicinta, tenemos los archivos "ModeloMTMP.hs" y "EjemplosModeloMTMP.hs". Contamos con dos ejemplos para esta variante:

- *mtmcAnBnCn* máquina de dos cintas tal que:

$$L(mtmcAnBnCn) = \{w \mid w = a^n b^n c^n,\ n > 0\}.$$

La ejecución es completamente análoga a la máquina multipista. De modo que basta con poner las instrucciones:

take 3 (*lenguajeAceptado mtmcAnBnCn* 2)
pintaProcesoAceptacion mtmcAnBnCn "*aabbcc*"

- *mtmcScS* máquina de dos cintas tal que:

$$L(mtmcScS) = \{wcw \mid w \in \{a, b\}^*\ \}.$$

Por último tenemos el archivo "EjemplosModeloMTND.hs" para el modelo no determinista. El ejemplo que hemos considerado para este modelo es la máquina *mtndCABoABC*, que acepta el lenguaje:

$$L(mtndCABoABC) = \{w \mid w = w_1\underline{cab}w_2 \text{ o } w = w_1\underline{abc}w_2\ \},$$

en donde w_1, $w_2 \in \{a, b, c\}^*$. Dado que este modelo siempre empieza con una cinta, podemos ejecutar la siguiente instrucción:

```
Main> take 7 (lenguajeAceptado mtndCABoABC)
```

obtenemos:

```
["abc","cab","aabc","abca","abcb","abcc","acab"]
```

o bien:

```
Main> pintaProcesoAceptacion mtndCABoABC "acab"
```

obteniendo como resultado:

```
|- (q0)acab
|- a(q0)cab
|- ac(q0)ab    ac(q1)ab    (q2)acab
|- aca(q0)b    aca(q3)b    (qr)Ecab
|- acab(q0)_    acab(qf)_    (qr)Ecab
|- True
```

B.3 Máquina universal de Turing

La máquina universal de Turing se puede probar utilizando las máquinas definidas para el modelo estándar. Sólo tenemos que importar los archivos "ModeloMTE.hs" y "EjemplosModeloMTELA.hs", colocando las líneas:

```
import ModeloMTE
import EjemplosModeloMTELA
```

al principio del archivo "MUT.hs", ya que este es el que se cargará en el intérprete de HASKELL, porque contiene todas las definiciones necesarias para probar los módulos que corresponden a la máquina universal. Una vez cargado el archivo, podemos ejecutar las siguientes funciones con todas y cada una de las máquinas definidas en los ejemplos.

En las siguientes ejecuciones, utilizaremos la definición de la máquina *mteAnBn*:

```
Main> mteAnBn

Estados:: q0  qf  q1  q2  q3

Estado Inicial:: q0

Estado Final:: qf

Alfabeto de Entrada:: a b

Alfabeto de la Cinta:: _ a b X Y

Funcion de Transicion::
                     d q0 a =  q1 X D
                     d q0 Y =  q3 Y D
                     d q1 a =  q1 a D
                     d q1 b =  q2 Y I
                     d q1 Y =  q1 Y D
                     d q2 a =  q2 a I
                     d q2 X =  q0 X D
                     d q2 Y =  q2 Y I
                     d q3 _ =  qf _ D
                     d q3 Y =  q3 Y D
```

Si ejecutamos la codificación sobre *mteAnBn*:

```
Main> codifica mteAnBn
```

devuelve como resultado:

```
"0101101110111101110010111110111110111110111001110110111011011101110
 0111011101111011111010011101111101110111110111001111011011101111011010
 0111101111010111101110011110111110111101111010011111010110101110
 011111011111011111101111101110"
```

En este módulo podemos ejecutar las funciones *aceptaCadena* y *pintaProcesoAceptacion MUT*.

Ejecutemos las funciones anteriores con la máquina *mteAnBn* y la cadena "aabb" cuya codificación es "011011011101110". Si ejecutamos *aceptaCadena*:

```
Main> aceptaCadena
     (codifica mteAnBn, ("011011011101110", 1, 2), "010")
```

obtenemos como resultado:

```
True
```

o bien:

```
Main> pintaProcesoAceptacionMUT
     (codifica mteAnBn, ("011011011101110", 1, 2), "010")
```

devuelve como resultado:

```
|- Cod. Cinta 1
   Cinta 2: 011011011101110
   Cinta 3: 010
|- Cod. Cinta 1
   Cinta 2: 01111011011101110
   Cinta 3: 01110
|- Cod. Cinta 1
   Cinta 2: 01111011011101110
   Cinta 3: 01110
|- Cod. Cinta 1
   Cinta 2: 0111101101111101110
   Cinta 3: 011110
|- Cod. Cinta 1
   Cinta 2: 0111101101111101110
   Cinta 3: 011110
|- Cod. Cinta 1
   Cinta 2: 0111101101111101110
   Cinta 3: 010
|- Cod. Cinta 1
   Cinta 2: 01111011101111101110
   Cinta 3: 01110
|- Cod. Cinta 1
   Cinta 2: 01111011101111101110
   Cinta 3: 01110
|- Cod. Cinta 1
   Cinta 2: 011110111101111110111110
   Cinta 3: 011110
|- Cod. Cinta 1
   Cinta 2: 011110111101111110111110
   Cinta 3: 011110
|- Cod. Cinta 1
   Cinta 2: 011110111101111110111110
   Cinta 3: 010
|- Cod. Cinta 1
```

```
   Cinta 2: 01111011110111110111110
   Cinta 3: 0111110
|- Cod. Cinta 1
   Cinta 2: 011110111101111101111010
   Cinta 3: 0111110
|- True
```

Para el proceso de decodificación podemos ejecutar:

```
Main> decodifica (codifica mteAnBn)
```

Obteniendo como resultado:

```
q0  'a'  =  q1  'c'  D
q0  'd'  =  q3  'd'  D
q1  'a'  =  q1  'a'  D
q1  'b'  =  q2  'd'  I
q1  'd'  =  q1  'd'  D
q2  'a'  =  q2  'a'  I
q2  'c'  =  q0  'c'  D
q2  'd'  =  q2  'd'  I
q3  '_'  =  qf  '_'  D
q3  'd'  =  q3  'd'  D
```

B.4 Autómata finito determinista

En el archivo "EjemplosAFD.hs" podemos encontrar la definición de una autómata finito determinista que acepta cadenas con un número par de ceros y un número impar de unos. Esta definición se muestra a continuación:

```
afPar0Impar1 = AF ["q0", "q1", "q2", "q3"]
                  "q0"
                  "q3"
                 ['0', '1']

afdPar0Impar1 = (afPar0Impar1, d) where
      d "q0" '0'  =  "q1"
      d "q0" '1'  =  "q3"
      d "q1" '0'  =  "q0"
      d "q1" '1'  =  "q2"
      d "q2" '0'  =  "q3"
      d "q2" '1'  =  "q1"
      d "q3" '0'  =  "q2"
      d "q3" '1'  =  "q0"
      d _ _ = "qr"
```

Una vez cargado el archivo, obtendremos la máquina de Turing correspondiente a $afdPar0$ $Impar1$ ejecutando la siguiente instrucción:

```
Main> convierteAFD afdPar0Impar1
```

Se pide el nombre del archivo en donde se coloca la máquina:

```
Archivo de Resultados:
```

Introducimos el nombre del archivo:

```
MTEPar0Impar1
```

Posteriormente se muestran los siguientes mensajes:

```
Se genero el archivo MTEPar0Impar1.hs

con la MTE relacionada al AFD:


Estados:: q0  q1  q2  q3

Estado Inicial:: q0

Estado Final:: q3

Alfabeto de Entrada:: 0 1

Funcion de Transicion::
               d q0 0 =  q1
                   d q0 1 =  q3
                   d q1 0 =  q0
                   d q1 1 =  q2
                   d q2 0 =  q3
                   d q2 1 =  q1
                   d q3 0 =  q2
                   d q3 1 =  q0
```

El resultado se encuentra en el archivo "MTEPar0Impar1.hs" y es el siguiente:

```
import ModeloMTE

mtAFD = MT [ "q0", "q1", "q2", "q3", "qf" ]
             "q0"
             "qf"
           [ '0', '1']
           [ '_', '0', '1']
```

```
mteAFD = (mtAFD, d) where
          d "q0" '0' = ("q1", '_', D)
          d "q0" '1' = ("q3", '_', D)
          d "q1" '0' = ("q0", '_', D)
          d "q1" '1' = ("q2", '_', D)
          d "q2" '0' = ("q3", '_', D)
          d "q2" '1' = ("q1", '_', D)
          d "q3" '0' = ("q2", '_', D)
          d "q3" '1' = ("q0", '_', D)
          d "q3" 'F' = ("qf", '_', N)
          d _ _ = ("qr", 'E', D)
```

Observemos que en toda transición de la máquina se respetan los estados de la transición original del autómata y siempre son de la forma ($estado$, $\flat$, D) excepto las dos últimas. En particular la penúltima transición es especial, ya que corresponde a la última transición que se le aplica a una cadena que es aceptada por la máquina. Recordemos que esta transición la agregamos ya que cuando ejecutamos el proceso de aceptación sobre una cadena, a ésta le concatenamos (internamente) el símbolo F al final indicando el límite de la misma.

Para comprobar que la conversión fue la adecuada, podemos cargar el archivo "MTEPar0 Impar1.hs" y ejecutar la función $lenguajeAceptadoAFDMTE$ del módulo $ModeloMTE.hs$:

```
Main> take 10 (lenguajeAceptadoAFDMTE mteAFD)
```

Obteniendo como resultado:

```
["1","001","010","100","111","00001","00010","00100","00111","01000"]
```

Cuando se quiere comprobar la aceptación de una cadena por medio de una máquina de Turing derivada de un autómata finito determinista, tenemos que usar las funciones $aceptaCadenaAFDMTE$, o bien, $pintaProcesoAceptacionAFDMTE$:

```
Main> aceptaCadenaAFDMTE mteAFD "000100011"
      True
Main> aceptaCadenaAFDMTE mteAFD "100100011"
      False
Main> pintaProcesoAceptacionAFDMTE mteAFD "000100011"
      |-(q0)000100011F
      |-_(q1)00100011F
      |-__(q0)0100011F
      |-___(q1)100011F
      |-____(q2)00011F
      |-_____(q3)0011F
      |-______(q2)011F
      |-_______(q3)11F
      |-________(q0)1F
      |-_________(q3)F
      |-_________(qf)_
```

```
      |-True

Main> pintaProcesoAceptacionAFDMTE mteAFD "100100011"
      |-(q0)100100011F
      |-_(q3)00100011F
      |-__(q2)0100011F
      |-___(q3)100011F
      |-____(q0)00011F
      |-_____(q1)0011F
      |-______(q0)011F
      |-_______(q1)11F
      |-________(q2)1F
      |-_________(q1)F
      |-_________E(qr)_
      |-False
```

B.5 Autómata de pila sin transiciones ε

El módulo de conversión de un autómata de pila sin transiciones ε a una máquina de Turing multicinta funciona de manera análoga al de conversión de un autómata finito determinista. En el archivo "EjemplosAPMTMC.hs" podemos encontrar la definición de un autómata de pila sin transiciones ε que acepta cadenas que contienen un mayor número de letras a que de letras b. Esta definición se muestra a continuación:

```
afMasAsqueBs = AF ["q0", "q1"]
                  "q0"
                  "q1"
                  ['a', 'b']
                  ['Z', 'a', 'b']

apfMasAsqueBs = (afMasAsqueBs, d) where
            d "q0" 'a' 'Z'  =  ("q1", ['Z'])
            d "q0" 'b' 'Z'  =  ("q0", ['b', 'Z'])
            d "q0" 'a' 'b'  =  ("q0", [])
            d "q0" 'b' 'b'  =  ("q0", ['b', 'b'])
            d "q1" 'a' 'Z'  =  ("q1", ['a', 'Z'])
            d "q1" 'b' 'Z'  =  ("q0", ['Z'])
            d "q1" 'a' 'a'  =  ("q1", ['a', 'a'])
            d "q1" 'b' 'a'  =  ("q1", [])
            d _ _ _  =  ("qr", [])
```

Una vez cargado el archivo, obtendremos la máquina de Turing correspondiente ejecutando la siguiente instrucción:

```
Main> convierteAP apfMasAsqueBs
```

Se pide el nombre del archivo en donde se coloca la máquina:

```
Archivo de Resultados:
```

Introducimos el nombre del archivo:

```
MTEMasAsqueBs
```

Posteriormente se muestran los siguientes mensajes:

```
Se genero el archivo MTEMasAsqueBs.hs

con la MTE relacionada al AP:


Estados:: q0  q1

Estado Inicial:: q0

Estado Final:: q1

Alfabeto de Entrada:: a b

Funcion de Transicion::
                          d q0 a Z = (q1, Z)
                          d q0 a b = (q0, [])
                          d q0 b Z = (q0, bZ)
                          d q0 b b = (q0, bb)
                          d q1 a Z = (q1, aZ)
                          d q1 a a = (q1, aa)
                          d q1 b Z = (q0, Z)
                          d q1 b a = (q1, [])
```

El resultado, se encuentra en el archivo “MTEMasAsqueBs.hs” y es el siguiente:

```
import ModeloMTMC

mtAP = MT [ "q0", "q1", "qf"]
            "q0"
            "qf"
          [ 'a', 'b']
          [ '_', 'Z', 'a', 'b']

mtmcAP = (mtAP, d) where
          d "q0" ['a','Z'] = ("q1", [('_', D), ('Z', N)])
          d "q0" ['a','b'] = ("q0", [('_', D), ('_', D)])
          d "q0" ['b','Z'] = ("q0", [('_', N), ('Z', I)])
          d "q0" ['_', '_'] = ("q0", [('_', D), ('b', N)])
```

```
d "q0" ['b','b'] = ("q0", [('_', N), ('b', I)])
d "q0" ['_', '_'] = ("q0", [('_', D), ('b', N)])
d "q1" ['a','Z'] = ("q1", [('_', N), ('Z', I)])
d "q1" ['_', '_'] = ("q1", [('_', D), ('a', N)])
d "q1" ['a','a'] = ("q1", [('_', N), ('a', I)])
d "q1" ['_', '_'] = ("q1", [('_', D), ('a', N)])
d "q1" ['b','Z'] = ("q0", [('_', D), ('Z', N)])
d "q1" ['b','a'] = ("q1", [('_', D), ('_', D)])
d "q1" ['F', _] = ("qf",[('_',N),('_',N)])
d _ _ = ("qr",[('E',N),('E',N)])
```

Para comprobar que la conversión fue la adecuada, podemos cargar el archivo "MTEMasAs queBs.hs" y ejecutar la función $lenguajeAceptadoAPMTMC$ del módulo $ModeloMTMC.hs$:

```
Main> take 30 (lenguajeAceptadoAPMTMC mtmcAP)
```

Obteniendo como resultado:

```
["a",
 "aa",
 "aaa","aab","aba","baa",
 "aaaa","aaab","aaba","abaa","baaa",
 "aaaaa","aaaab","aaaba","aaabb","aabaa","aabab","aabba","abaaa",
 "abaab","ababa","abbaa","baaaa","baaab","baaba","babaa","bbaaa",
 "aaaaaa","aaaaab","aaaaba"]
```

Cuando se quiere comprobar la aceptación de una cadena por medio de una máquina de Turing derivada de un autómata de pila sin transiciones ε, tenemos que usar las funciones $aceptaCadenaAPMTMC$, o bien, $pintaProcesoAceptacionAPMTMC$:

```
Main> aceptaCadenaAPMTMC mtmcAP "abababababababa"
      True

Main> aceptaCadenaAPMTMC mtmcAP "abababababababb"
      False

Main> pintaProcesoAceptacionAPMTMC mtmcAP "abababababababa"
      |- q0  [(a)bababababababaF]  [(Z)______________]
      |- q1  [_(b)ababababababaF]  [(Z)______________]
      |- q0  [__(a)bababababababaF]  [(Z)______________]
      |- q1  [___(b)ababababababaF]  [(Z)______________]
      |- q0  [____(a)bababababababaF]  [(Z)______________]
      |- q1  [_____(b)ababababababaF]  [(Z)______________]
      |- q0  [______(a)bababababababaF]  [(Z)______________]
      |- q1  [_______(b)ababababababaF]  [(Z)______________]
      |- q0  [________(a)bababababababaF]  [(Z)______________]
      |- q1  [_________(b)ababababababaF]  [(Z)______________]
      |- q0  [__________(a)bababababababaF]  [(Z)______________]
```

```
      |- q1  [__________(b)aF]  [(Z)___________]
      |- q0  [___________(a)F]  [(Z)___________]
      |- q1  [_____________(F)]  [(Z)___________]
      |- qf  [_____________(_)]  [(_)___________]
      |- True

Main> pintaProcesoAceptacionAPMTMC mtmcAP "abababababab"
      |- q0  [(a)bababababbF]  [(Z)___________]
      |- q1  [_(b)abababababbF]  [(Z)___________]
      |- q0  [__(a)bababababbF]  [(Z)___________]
      |- q1  [___(b)abababababbF]  [(Z)___________]
      |- q0  [____(a)bababababbF]  [(Z)___________]
      |- q1  [_____(b)abababbF]  [(Z)___________]
      |- q0  [______(a)bababbF]  [(Z)___________]
      |- q1  [_______(b)ababbF]  [(Z)___________]
      |- q0  [________(a)babbF]  [(Z)___________]
      |- q1  [_________(b)abbF]  [(Z)___________]
      |- q0  [__________(a)bbF]  [(Z)___________]
      |- q1  [___________(b)bF]  [(Z)___________]
      |- q0  [___________(b)F]  [(Z)___________]
      |- q0  [___________(_)F]  [(_)Z___________]
      |- q0  [_____________(F)]  [(b)Z___________]
      |- qr  [_____________(E)]  [(E)Z___________]
      |- False
```

Bibliografía

[1] Peter Linz.*An Introduction to Formal Languages and Automata*. Jones and Bartlett Computer Science. Third Edition, November 6, 2000.

[2] Michael Sipser.*Introduction to the Theory of Computation*. An International Thomson Publishing Company. Second Edition, February 15, 2005.

[3] John Martin.*Lenguajes Formales y Teoría de la Computación*. Mc Graw Hill. Tercera Edición, Julio, 2005.

[4] Graham Hutton.*Programming in Haskell*. Cambridge University Press. Paperback Edition, 2007.

[5] Simon Peyton Jones.*Haskell 98 Languaje and Libraries: The Revised Report*. Cambridge University Press. First Edition, 2003.

[6] Alex Borbón A., Walter Mora F.*Edición de Textos Científicos LaTeXComposición, Gráficos y Beamer*. Escuela de Matemáticas Instituto Tecnológico de Costa Rica. Textos Universitarios.

[7] A. M. Turing.*Computing Machinery and Intelligence*. Creative Computing. January, 1980.