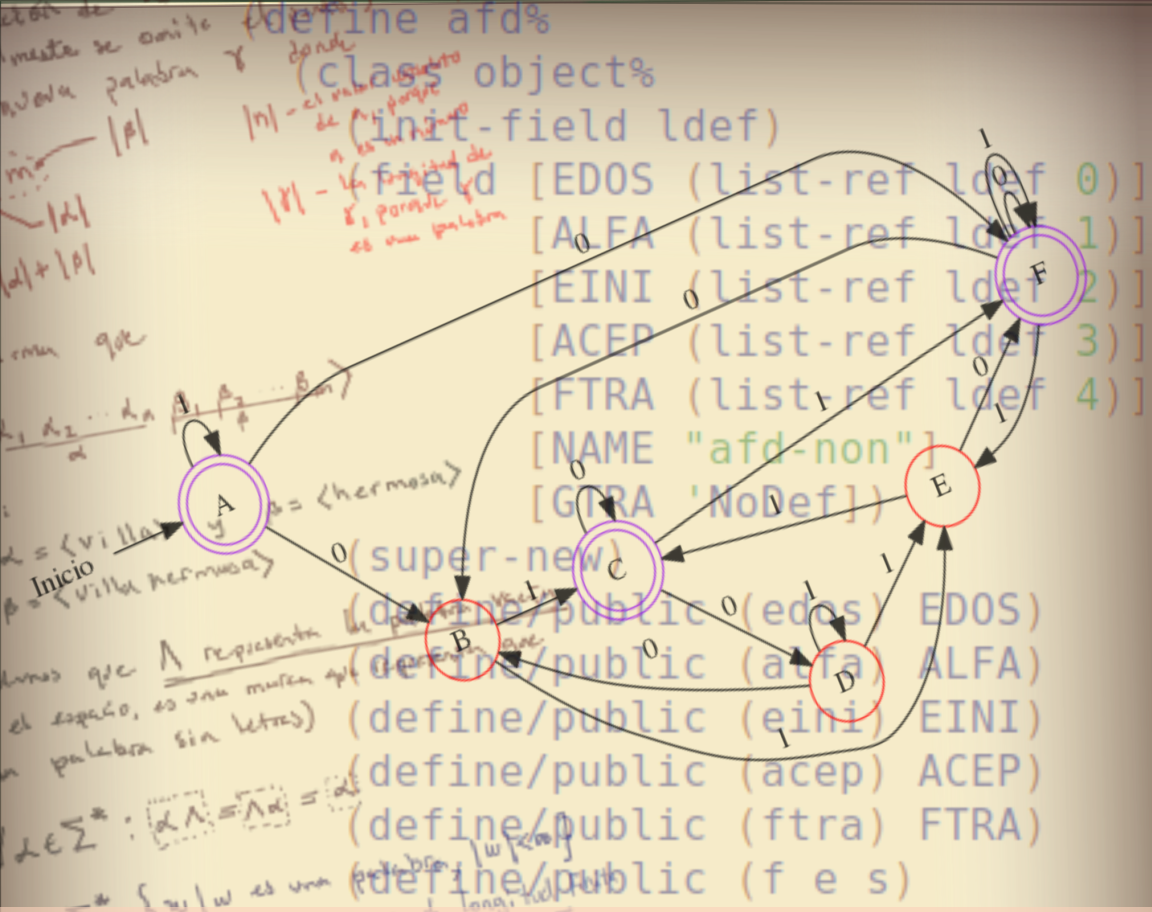


Lenguajes y Autómatas

Una perspectiva funcional con Racket

Abdiel E. Cáceres González

;

Lenguajes y Autómatas

Una perspectiva funcional con Racket

Abdiel E. Cáceres González

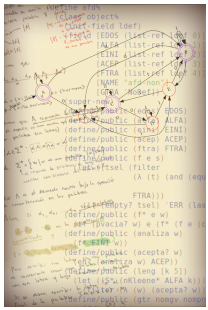


Imagen en portada y título de capítulos:
"Las estructuras de la mente"
por Abdiel E. Cáceres González.

Lenguajes y Autómatas Una perspectiva funcional con Racket. / Abdiel E. Cáceres González (Autor). – Primera edición. – Villahermosa, Centro, Tabasco: Universidad Juárez Autónoma de Tabasco, 2019. 270 páginas – (Colección: Héctor Ochoa Bacelis, Textos de enseñanza de ciencias básicas).

ISBN: 978-607-606-507-5

Incluye referencias bibliográficas al final.

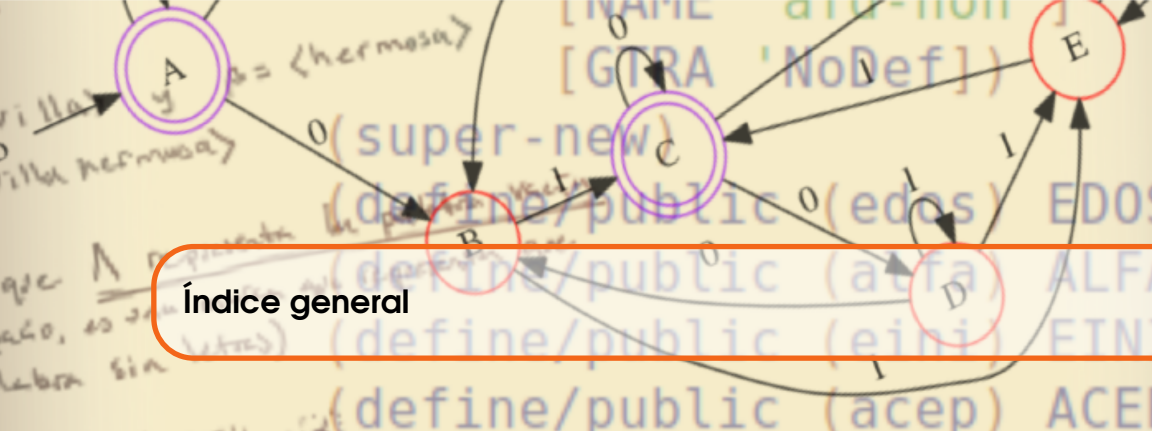
PFCE

Primera edición, 2019

Av. Universidad s/n, Zona de la Cultura, Col. Magisterial, C. P. 86040
Villahermosa, Centro, Tabasco.
www.ujat.mx
ISBN: 978-607-606-507-5

Apoyo editorial: Calíope Bastar Dorantes, José Manuel Vázquez Broca.
Corrección de estilo: Francisco Cubas Jiménez.
Diseño: Abdiel Emilio Cáceres González.
Hecho en Villahermosa, Tabasco, México.

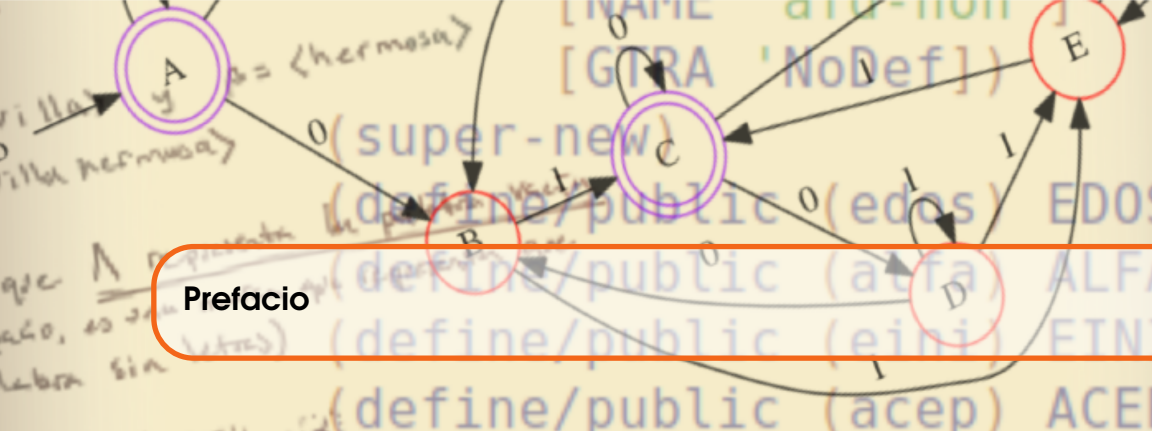
Para Amparo y Sara Camila



Índice general

III

Autómatas



Prefacio

Este libro es el resultado de algunos años de coleccionar y mejorar funciones que se han empleado en cursos de Lenguajes Formales y Autómatas, en la Universidad Juárez Autónoma de Tabasco, en México.

El contenido de este material abarca diferentes temas fundamentales, desde lógica proposicional hasta la teoría clásica de autómatas finitos. El enfoque presentado es completametne práctico, programando casi todas las funciones matemáticas que usualmente se estudian solamente en teoría, en prácticamente la mayoría de los libros de texto.

Esta obra está dirigida especialmente para las personas de habla hispana que participen en un primer curso de autómatas finitos. La intención es lograr un conocimiento práctico y profundo de los temas fundamentales de la teoría de autómatas y al mismo tiempo mejore las habilidades de programción funcional.

Se empleado el lenguaje de programación `DrRacket` para implementar las funciones matemáticas, porque es un lenguaje con una sintaxis muy simple y fácil de aprender, lo cual es sumamente útil para expresar en código las ideas y conceptos de la teoría de lenguajes formales y autómatas finitos. A lo largo de este texto, se desarrollan expresiones en código fuente para definir desde los conceptos más simples y fundamentales, hasta las expresiones más complejas que son utilizadas. Aún en los últimos programas se mantiene un nivel de simpleza

bastante aceptable para todo nivel de usuarios.

El contenido del libro se ha dividido en cuatro partes: fundamentos, de símbolos a lenguajes, autómatas y finalmente, manipulación de autómatas finitos.

En la primera parte, en la parte de fundamentos, se tratan principalmente los temas de lógica de primer orden, desde los valores de verdad hasta los cuantificadores; la teoría de conjuntos, relaciones y funciones; en la segunda parte se introducen los primeros términos relacionados con los lenguajes, es decir, los símbolos con los que se define un alfabeto y a su vez que permiten la creación de palabras y lenguajes.

La tercera parte incluye los temas de autómatas finitos deterministas, indeterministas e indeterministas con transiciones nulas, en esta parte se introduce el paradigma orientado a objetos para implementar cada tipo de autómata; finalmente en la cuarta parte se utilizan todas las herramientas definidas para poder manipular autómatas, reduciendo el tamaño de ellos, cambiando el nombre de los estados o el alfabeto, o bien uniendo o concatenando autómatas.

Como una de las metas de este libro ha sido proporcionar al lector el conocimiento necesario, para obtener la habilidad de definir funciones que le permitan un conocimiento práctico del tema, se ha incluido más de 90 programas en `DrRacket` que han sido probados y que sustentan las bases de los conceptos descritos en el libro.

El código fuente es una parte muy importante del texto, se incluyen las definiciones y expresiones que se han tipografiado respetando los colores que aparecen en el IDE de `DrRacket`, esta característica es notable solo en la versión electrónica:

- Los identificadores como `mayores-de`, en letra `TrueType` coloreada en `azul`.
- Las expresiones atómicas se escriben en `azul` con un apóstrofe que le antecede, como en `'p`, que se interpreta con el significado /p/, a diferencia del identificador `p` cuyo significado debe establecerse antes de utilizar el símbolo.
- Las constantes como `pi` o los números de cualquier clase se escriben en color `verde`, como `3.1416`, o `4`.
- El resultado de las evaluaciones se escribe en color `violeta` como en el ejemplo `'(b c d e)`. Este color es diferente al color por defecto en el IDE de `DrRacket`, pero se ha decidido este color para diferenciarlo de los identificadores y el lector pueda identificarlo más rapidamente. Las evaluaciones son el resultado de las interacciones y estas aparecen en los ejemplos.
- Los comentarios en el código se escribe en color *café* anteponiendo una marca de /;/, como en: `; Esto es un comentario en el código`.
- En las evaluaciones, ocasionalmente surgen errores. Con el fin de que el programador observe este comportamiento, se han introducido algunas

interacciones equivocadas, donde el error se escribe en rojo, como en el siguiente mensaje car: contract violation.

- La mayoría de las expresiones en DrRacket se encierran entre paréntesis, resaltando el color de los paréntesis, como `(list 4 'q 'p)`.

Los programas en DrRacket se han escrito en apartados como el siguiente:

```
1 ; (mayores-de L μ) ↦ listaDe/numero?
2 ; L : listaDe/numero?
3 ; μ : numero? - umbral
4 (define mayores-de
5   (λ (L μ)
6     (filter (λ (item) (> item μ)) L)))
```

Cada programa cuenta con números de línea para una mejor referencia, así como comentarios para establecer la firma o encabezado de la función.

DrRacket tiene un conjunto relativamente pequeño de palabras reservadas. Cuando se utiliza por primera vez alguna función de DrRacket, se hace un breve comentario explicando la sintaxis y el uso de tal función. Este tipo de comentarios se señalan con el logotipo del lenguaje:

En el IDE de DrRacket es posible introducir algunos símbolos matemáticos tales como μ ó λ. Para instertar un símbolo matemático en el IDE de DrRacket, se escribe `\lambda` seguido del juego de teclas `<Alt>+ <\>`. Para una mejor información consulte la documentación del lenguaje en la sección *LaTeX and TeX inspired keybindings*.

Casi siempre después de un segmento de código, se muestra el uso del concepto mediante una interacción. Al programar en el IDE de DrRacket, para comprobar el correcto funcionamiento de la definición, se hacien interacciones con el sistema. Las interaciones están escritas en un recuadro:

```
> (mayores-de '(2 3 8 4 9) 4)
'(8 9)
>
```

Las interacciones se escriben con la finalidad de que el lector, al escribir su propio código, compare la salida que le ofrece su interacción con la salida ofrecida en este texto. Tanto las interacciones como el código fuente se han tipografiado de la manera más similar que se presenta en el propio IDE de DrRacket, con el fin de que la experiencia de aprendizaje del lenguaje del propio lector, sea lo más amable posible.

En muy pocas ocasiones se hace una nota para relacionar alguna definición escrita en DrRacket, con la misma definición escrita en un modo típico utilizando símbolos y notación matemática convencional como en:

El símbolo $\forall$ se usa en expresiones lógicas de orden superior, se escribe como $\forall(x \in A) : P(x)$, donde: A es un conjunto y P es un predicado.

Por el contrario, en la mayoría de las ocasiones se escriben las expresiones matemáticas al estilo de DrRacket, inclusive dentro del mismo texto, como en (multiplicar 2 30); esto se hace con el propósito de que el lector se familiarice con la notación y expresividad del lenguaje, sin embargo en ocasiones se acompaña de una anotación como la anterior.

Este libro cuenta con toda la colección de definiciones, que se definen particularmente a lo largo del texto y se ofrecen juntas en el apéndice A. El lector puede transcribir el código a su computadora y ejecutarlo a medida que vaya avanzando en su estudio.

Todas las definiciones hechas en DrRacket son probadas mediante ejemplos, que se han enmarcado en un recuadro para simular una interacción, como en el siguiente ejemplo:

■ **Ejemplo 0.1** En este ejemplo se muestra una interacción típica en DrRacket

```
> (car '(a b c d e))
'a
> (cdr '(a b c d e))
'(b c d e)
>
```

■

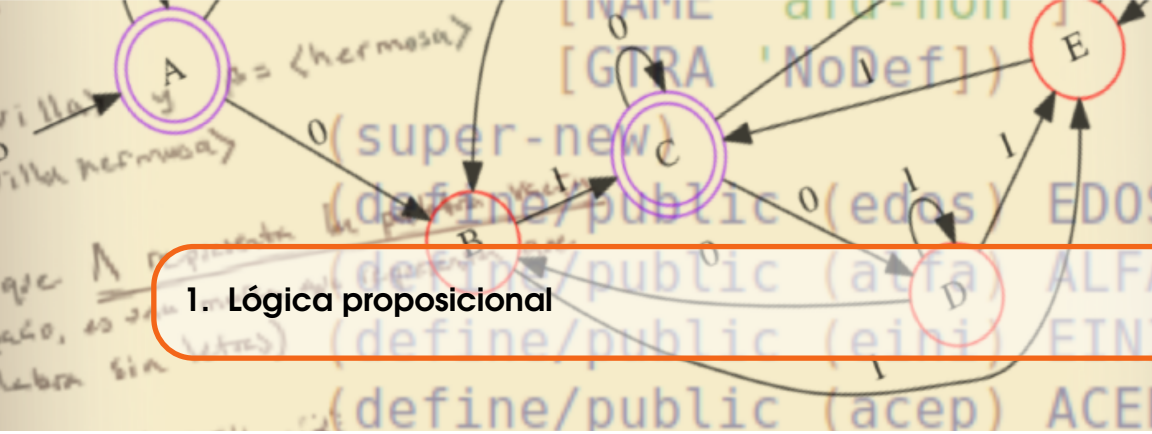
He puesto especial atención en ofrecer una bibliografía que combine textos clásicos y textos de más reciente publicación [hasta el momento de edición], con el fin de complementar el conocimiento, combinando la teoría y la práctica.

En el índice alfabético se enlistan también las palabras clave que han sido definidas como lenguaje? o alfabetoVacio; así como las que son propias de DrRacket, como car, o cdr. El lector el libre de referirse a éste cuando lo considere oportuno.

Este texto se escribió en LaTeXcon un tipo base TimesRoman de 12 puntos, TrueType también de 12 puntos [https://www.latex-project.org/]; el código se escribió en la versión 7.0 de DrRacket [https://racket-lang.org/]; el código de las imágenes para mostrar los diagramas de estados se generaron en DrRacket y las imagenes se realizaron con Graphviz [http://www.graphviz.org/].

Finalmente quiero agradecer particularmente a Nestor José León Hernández y a Viviana Solano González, de Gloria Jean's COFFEES, por su amable trato y por su generoso patrocinio de rico y delicioso café, que me ha acompañado en el desarrollo de este libro.

Fundamentos



1. Lógica proposicional

Para el estudio de los autómatas finitos se requieren algunas herramientas de matemáticas discretas y de programación. De parte de matemáticas discretas, necesitaremos elementos de lógica matemática, de conjuntos, de relaciones y funciones; y de parte de la programación, utilizaremos un enfoque funcional basado en `DrRacket`, porque es un lenguaje sencillo de aprender y es sumamente simple, pronto se dará cuenta que se dedica más tiempo para plasmar las ideas relacionadas a la teoría de lenguajes formales y autómatas, que para aprender la sintaxis y reglas de escritura del propio lenguaje.

1.1 Expresiones simbólicas

`DrRacket` es un lenguaje de programación para hacer cómputo simbólico, esto significa que es encesario otorgar un significado a los símbolos para manipularlos y después de hacerlo, es necesario interpretar los símbolos para leer el nuevo significado.

La dinámica de trabajo en el lenguaje se basa en un ciclo interactivo que se llama `REPL`, que son siglas en inglés y en español significa *Ciclo Leer, Evaluar, Imprimir (Read Evaluate Print Loop)*. Hay principalmente dos tipos de expresiones:

E. Primitivas: Como `#t` o `#f`, también números como `10`, `24.48`, o $\frac{1}{2}$; o las cadenas de caracteres entre comillas como `“hola mundo”` y también los símbolos con un apóstrofo que le antecede como en `'nombre-variable` son considerados como símbolos primitivos.

■ **Ejemplo 1.1** Expresiones primitivas
- `#t ; Es un valor de verdad`
- `“falso” ; Es una cadena de texto`
- `3.14159 ; Es un número`
- `'3+2 ; Es un símbolo`
- `'(w x y z) ; Es una lista de símbolos`

■

E. Compuestas: Las expresiones compuestas se escriben agrupando símblos entre paréntesis utilizando notación prefija.

Como regla general una expresión compuesta **empieza con un operador** y **el resto son sus operandos**, siguiendo el siguiente patrón

(⟨operador⟩ ⟨operando⟩ ...)

El símbolo ... significa *y así sucesivamente, con 0 o más elementos*; de paso hay otro símbolo que utilizaremos de vez en cuando, es ...+ que significa: *y así sucesivamente, con 1 o más elementos*.

■ **Ejemplo 1.2** Expresiones compuestas
- `(+ 8 10); Compuesta con dos primitivas`
- `(= (+ 8 10) (- 20 2)); dos combinaciones`

■

1.2 Valores de verdad

Definimos como valores de verdad los conceptos: **falso** y **cierto**. El valor falso lo escribimos como `#f`y el cierto lo escribimos como `#t`.

Además de `#t` y `#f`, otros símbolos para los valores de verdad son `#T` y `#F` para cierto y falso respectivamente.

Cualquier expresión lógica bien puede ser o bien cierta, o bien falsa, pero no puede ser ambas al mismo tiempo, o bien carecer de algún valor de verdad. En el cálculo de proposiciones el objetivo es combinar proposiciones para generar expresiones lógicas complejas y determinar su valor de verdad; o bien calcular el valor de verdad de expresiones compuestas, a partir del valor de verdad de expresiones más simples.

Después de una adecuada definición, otros símbolos pueden ser utilizados como valores de verdad; como el 1 para `#t`y 0 para `#f`; en circuitos eléctricos y electrónicos suele utilizarse `ON` (encendido) como `#t` y `OFF` (apagado) como `#f`.

■ **Ejemplo 1.3** El valor asociado a los valores de verdad se puede obtener al interactuar con DrRacket.

```
> #t; El símbolo para cierto
#t
> #f; El símbolo para falso
#f
```

■

La lógica matemática es un campo de estudio sumamente extenso, lo que se presenta aquí no es sino un muy breve repositorio de definiciones que deben alentar al lector a ingadar más sobre el tema. Un excelente libro para empezar un estudio diligente en la lógica matemática es «*A Beginner's Guide to Mathematical Logic*» [Smu14], pero sin duda en una búsqueda en internet es posible obtener muchos más títulos.

1.2.1 Proposiciones

Una proposición es una **expresión lógica** que en DrRacket puede ser evaluada o bien como #t o como #f. Las expresiones proposicionales pueden ser primitivas o compuestas.

■ **Ejemplo 1.4** Los siguientes ejemplos son expresiones proposicionales en DrRacket.

1. #t ↦ #t ; *Es primitiva, significa cierto*
2. #f ↦ #f ; *Es primitiva, significa falso*
3. (= 4 4) ↦ #t; $4=4$ *produce verdadero*
4. (= 14 5) ↦ #f; $14=5$ *es evaluado como falso*
5. (> (+ 2 2) (* 2 2)) ↦ #f; $2+2>2*2$ *significa falso*

■

El símbolo /↦ / debe leerse como «*produce*» ; se utiliza cuando se desea obtener el valor de una expresión. El sistema debe leer la solicitud, evaluarla y *producir* un resultado. De manera similar el símbolo /↤/, se utiliza para asignar un valor a un identificador. En este caso, una expresión que involucra el símbolo /↤/ se podría leer como «*en adelande, ... tendrá el valor de ...*». Finalmente el símbolo /=/ es reservado para comparar la igualdad entre números.

1.2.2 Predicados

Una expresión lógica cuyo valor de verdad depende del valor de verdad de uno o más de sus argumentos se llama **predicado**. En DrRacket un predicado se escribe en forma de un procedimiento anónimo, utilizando una λ-expresión. Un predicado anónimo contiene las instrucciones que se deben efectuar para obtener un valor de verdad, pero sin asociar el procedimiento con algún identificador.

Una expresión lambda, escrita también como λ-expresión es una expresión que permite comunicar la existencia de variables que deben ser sustituídas en una expresión simbólica. En DrRacket las λ-expresiones sirven para crear predicados y en general funciones, se escriben con el siguiente formato:

(λ (arg-id ...) ⟨expr⟩ ...)

de manera equivalente

(lambda (arg-id ...) ⟨expr⟩ ...)

Que puede leerse como *El predicado (el procedimiento) que requiere los argumentos llamados* `arg-id` ..., *con los que se debe evaluar las expresiones* `expr` ..., El uso del símbolo λ, como lo señala Abelson en [ASS96], proviene del λ-cálculo, que es un formalismo dado a conocer por Alonzo Church [Chu41], con la finalidad de proporcionar una manera formal y rigurosa de expresar las nociones de *función* y *aplicación de una función*.

■ **Ejemplo 1.5** Las siguientes expresiones son predicados:

$3x = 12$. El valor de verdad depende de x.

`(λ (x) (= (* 3 x) 12))`. El valor de verdad depende del valor de `x`.

Para obtener un valor de verdad de las expresiones anteriores, primero se debe establecer el valor de x en el primer caso, o de `x` en el segundo:

- Si $x = 5$, $3x = 12$ es `#f`, porque $3 \cdot 5 = 15$ y $15 \neq 12$.
- Para interactuar con una expresión lambda, utilizamos el IDE de DrRacket:

```
> ((λ (x) (= (* 3 x) 12)) 5)
#f
>
```

Observe en la interacción, que la expresión lambda ha tomado el lugar que corresponde a un operador, el primero de la lista. ■

Un predicado puede convertirse es una proposición cuando todos los argumentos del predicado hayan sido definidos y valorados. Cuando todas las variables hayan sido instanciadas será posible evaluar el predicado y determinar un valor de verdad. En el momento en que las variables de un predicado se relacionan con valores, deja de ser predicado para convertirse en proposición.

■ **Ejemplo 1.6** También es posible encontrar predicados en en lenguaje natural:

- «Una persona es mi amigo». El valor de verdad depende de quién es esa persona.
- «Todos los de mi equipo son buenas personas». Será cierto una vez que se especifique qué personas pertenecen al equipo en cuestión y luego cada persona de ese equipo, debe confirmar que en verdad «es una buena persona».
- «Mañana será un gran dia». Será cierto cuando llegue el día de mañana y se pueda evaluar la sentencia.

■

Negación

Si `p` es una expresión lógica, `(neg p)` es el predicado que requiere un valor booleano `p`, con el que se obtiene el valor de verdad diferente a `p`.

☞ En notación matemática usual, la negación de p se escribe como $\neg p$ y se define como

$$\sim p = \begin{cases} \texttt{\#t} & \text{si } p \mapsto \texttt{\#f}; \\ \texttt{\#f} & \text{si } p \mapsto \texttt{\#t}; \end{cases}$$

Así podemos crear una definición `DrRacket` para determinar el valor de verdad de la negación de una proposición.

Código 1.1: *La negación*

```
1 ; (neg p) ↦ booleano?
2 ; p : booleano?
3 ; Ej (neg #t)↦ #f : La negación de #t significa #f
4 (define neg
5   (λ (p)
6     (if p #t #f)))
```

Las definiciones de procedimientos en `DrRacket` tienen cuatro partes:

1. La sección de comentarios, que aprovecharemos para escribir el **contrato** de la función, que indica qué elementos se esperan recibir y qué tipo de elemento se devuelve al finalizar el proceso. En el programa escrito en el código 1.1, el contrato es:

   ```
   1 ; (neg p) ↦ booleano?
   2 ; p : booleano?
   3 ; Ej (neg #t) ↦ #f : La negación de #t produce #f
   ```

   El contrato dice que se debe escribir `(neg p)`, donde `p` es una expresión que al ser evaluada se obtiene un valor booleano y resultado de hacer la negación, se obtiene un valor que es de hecho un valor booleano (`booleano?`). Aunque el contrato es opcional, es muy importante escribirlos para que el código sea más fácil de leer para las personas. Un código de programación no solamente debería servir para que las computadoras realicen una tarea, sino para que las personas tengan un mejor conocimiento acerca de los conceptos que estamos definiendo en nuestros programas.
2. La palabra clave `define`, esto indica a la computadora que se va a crear un nuevo concepto.
3. El identificador, es una secuencia de símbolos que identifican el procedimiento que se define, con este nombre podemos saber cuál es la **función** que realiza el procedimiento. En el código 1.1, el identificador es la palabra `neg`.
4. La expresión λ, que es un procedimiento con parámetros o sin ellos, es una expresión que puede incluir más expresiones que se evalúan para realizar la función específica. Nuevamente en el código 1.1, la expresión λ es:

```
5  (λ (p)
6    (if p #f #t))
```

En el código 1.1 de la negación, el formato requiere una expresión lógica asociada con la literal p; luego la expresión condicional if determina el valor resultante en dependencia de la proposición p.

En la expresión de decisión if, si p ↦ #t, el resultado de la evaluación es el tercer término de la expresión, el valor #f; de otro modo, cuando no es el caso que p ↦ #t, el valor de la expresión es el cuarto término de la expresión condicional, el valor #t.

En DrRacket la expresión if , es una S-expresión especial que tiene el siguiente formato:

(if ⟨expr-lógica⟩ ⟨caso-verdadero⟩ ⟨alternativa⟩)

Se utliza una expresión if cuando se desea evaluar una de dos posibles expresiones, aquella expresión que será evaluada, se determina por el valor de verdad de la ⟨expr-lógica⟩; si ⟨expr-lógica⟩ ↦ #t, entonces la expresión condicional if se evalúa con el resultado de la evaluación de ⟨caso-verdadero⟩; en caso contrario, es decir que ⟨expr-lógica⟩ ↦ #f, entonces el valor de la expresión if es el valor obtenido de la expresión ⟨alternativa⟩.

Conjunción

Si p y q son expresiones lógicas, la conjunción de p con q es un predicado (y p q) cuyo valor de verdad se describe en los siguientes casos.

- si p↤#t y q↤#t, entonces (y p q)↦ #t.
- si al menos una de p o q es #f, entonces (y p q)↦ #f.

En notación usual de matemáticas, la conjunción se representa con el símbolo ∧ y se define como:

$$p \wedge q = \begin{cases} \texttt{\#t} & \text{si } p \leftarrow\!\shortmid \texttt{\#t} \text{ y } q \leftarrow\!\shortmid \texttt{\#t} \\ \texttt{\#f} & \text{en otro caso.} \end{cases}$$

Una **tabla de verdad** muestra de manera tabular los valores de verdad de una expresión lógica compuesta. Para la conjunción tenemos los cuatro casos, mostrando cada caso en un renglón. El orden de los renglones no es muy importante, pero se debe respetar el orden elegido. En la siguiente **tabla de verdad** se muestra el comportamiento de la conjunción de dos proposiciones:

	p	q	(y p q)
caso 1:	#t	#t	#t
caso 2:	#t	#f	#f
caso 3:	#f	#t	#f
caso 4:	#f	#f	#f

Solo en el caso 1, la expresión (y p q)↦ #t, porque tanto p como q son #t, en los demás una o ambas proposiciones son #f.

Código 1.2: *La conjunción*

```
; (y p q) ↦ booleano?
; p : booleano?
; q : booleano?
; (y #t #f) ↦ #f
(define y
  (λ (p q)
    (if p q #f)))
```

■ **Ejemplo 1.7** Probar el comportamiento de la conjunción `y`

```
> (y #t #t); caso 1 de la tabla de verdad
#t
> (y #t #f); caso 2 de la tabla de verdad
#f
> (y #f #t); caso 3 de la tabla de verdad
#f
> (y #f #f); caso 4 de la tabla de verdad
#f
```

■

Disyunción

La disyunción es un predicado `(o p q)` que requiere dos expresiones lógicas `p` y `q`, para generar nueva proposición con valor de verdad `#t` si al menos una de las proposiciones es `#t`y `#f` si ambos argumentos son `#f`; de acuerdo con la tabla de verdad de la disyunción.

	p	q	(o p q)
caso 1:	#t	#t	#t
caso 2:	#t	#f	#t
caso 3:	#f	#t	#t
caso 4:	#f	#f	#f

☞ En notación usual de matemáticas, la disyunción se representa con el símbolo $\vee$ y se define como:

$$p \vee q = \begin{cases} \texttt{\#t} & \text{si } p \leftarrow\!\!\!\mapsto \texttt{\#t} \text{ o } q \leftarrow\!\!\!\mapsto \texttt{\#t} \\ \texttt{\#f} & \text{en otro caso.} \end{cases}$$

Código 1.3: *La disyunción*

```
; (o p q) ↦ booleano?
; p : booleano?
; q : booleano?
(define o
  (λ (p q)
    (if p #t q)))
```

■ **Ejemplo 1.8** Probar el comportamiento de la disyunción `o`:

```
> (o #t #t); caso 1
#t
> (o #t #f); caso 2
#t
> (o #f #t); caso 3
#t
> (o #f #f); caso 4
#f
```

■

Extensión de la conjunción y la disyunción

Se extiende el concepto de conjunción (y disyunción), para aplicar el mismo predicado a una secuencia de expresiones lógicas. En la conjunción extendida el predicado es `#t`, si todas las expresiones lógicas son `#t`y `#f` si al menos una es `#f`, observe la tabla de verdad.

...	p3	p2	p1	(y* p1 p2 p3 ...)
...	#t	#t	#t	#t
...	#t	#t	#f	#f
⋮	⋮	⋮	⋮	⋮
...	#f	#f	#f	#f

Código 1.4: *La conjunción extendida*

```
; (y* LPred) ↦ booleano?
; LPred : booleano? ...
(define y*
  (λ LPred
    (define y*aux
      (λ (LP)
        (cond ((empty? LP) #t)
              ((car LP) (y*aux (cdr LP)))
              (else #f))))
    (if (empty? LPred)
        #t
        (y*aux LPred))))
```

■ **Ejemplo 1.9** La conjunción extendida puede recibir `0` o más argumentos:

```
> (y* #t #t #t #t)
#t
> (y* #t #t #f #t)
#f
>
```

■

☞ En la notación convencional, la conjunción extendida se puede escribir como:

$$P(x_1) \wedge P(x_2) \wedge \cdots \wedge P(x_k)$$

Con P un predicado y $x_1, x_2, \ldots, x_k$ son valores con los que se evaúa cada predicado.

La disyunción extendida se define de manera similar. Se determina `#t` cuando al menos una expresión lógica es `#t`.

...	p3	p2	p1	(o* p1 p2 p3 ...)
...	#t	#t	#t	#t
...	#t	#t	#f	#t
⋮	⋮	⋮	⋮	⋮
...	#f	#f	#f	#f

☞ En la notación convencional de matemáticas, la disyunción extendida de una secuencia de proposiciones se puede escribir como:

$$P_1(x_1) \vee P_2(x_2) \vee \cdots \vee P_k(x_k)$$

Cuando $P_1, P_2, \ldots, P_k$ son predicados no necesariamente iguales; $x_1, x_2, \ldots, x_k$ son una colección de valores con los que se evaúa cada predicado;

Código 1.5: *La disyunción extendida*

```
; (o* LPred) ↦ booleano?
; LPred : booleano? ...
(define o*
  (λ LPred
    (define o*aux
      (λ (LP)
        (cond ((empty? LP) #f)
              ((car LP) #t)
              (else (o*aux (cdr LP))))))
    (if (empty? LPred)
        #f
        (o*aux LPred))))
```

■ **Ejemplo 1.10** Interacciones para observar la disyunción extendida.

```
> (o*)
#f
> (o* #t)
#t
> (o* #f)
#f
> (o* #t #t)
#t
> (o* #t #t #f)
#t
>
```

■

La disyunción exclusiva

La **disyunción exclusiva** se puede escribir como un predicado de nombre `ox`, que recibe dos valores booleanos como argumentos y genera un valor `#t` si exactamente uno de los argumentos es `#t`, mientras que el otro es `#f`; y devuelve un valor `#f` cuando ambas proposiciones son o bien `#t`, o ambas `#f`.

	p	q	(ox p q)
caso 1:	#t	#t	#f
caso 2:	#t	#f	#t
caso 3:	#f	#t	#t
caso 4:	#f	#f	#f

Observe en la siguiente definición, que cuando el argumento `p` es `#f`, el valor de `q` se devuelve sin ser afectado.

Código 1.6: *Disyunción exclusiva*

```
; (ox p q) ↦ booleano?
; p : booleano?
; q : booleano?
(define ox
  (λ (p q)
    (if p (neg q) q)))
```

■ **Ejemplo 1.11** Comportamiento de la función `ox`.

```
> (ox #t #t); Caso 1 de la tabla de verdad.
#f
> (ox #t #f); Caso 2 de la tabla de verdad.
#t
> (ox #f #t); Caso 3 de la tabla de verdad.
#t
> (ox #f #f); Caso 4 de la tabla de verdad.
#f
```

■

La condicional

Si `p` y `q` son expresiones lógicas, la **condicional** escrita como `(-> p q)` es una expresión con valor de verdad dependiente de `p` y `q`, como aparecen en la tabla de verdad:

	p	q	(-> p q)
caso 1:	#t	#t	#t
caso 2:	#t	#f	#f
caso 3:	#f	#t	#t
caso 4:	#f	#f	#t

1. El operando `p`, conocido como el **antecedente** o hipótesis de la implicación.

2. El operando `q`, llamado el **consecuente** o tesis de la implicación.
3. La expresión combinada `(-> p q)` es la condicional o **implicación**. La condicional es `#f`, cuando el antecedente es `#t` y el consecuente es `#f`.

En notación matemática convencional, la implicación se denota con el símbolo ⇒, como en $p \Rightarrow q$. La implicación es un predicado muy importante porque establece que una proposición (`q`) es una consecuencia lógica de otra (`p`). Algunos autores han dedicado suficientes páginas para el estudio de este predicado [RM04].

La condicional o implicación, ha sido de mucha utilidad en los lenguajes de programación, ya que se puede modelar una expresión condicional para ejecutar código dependiendo de la evaluación de una expresión booleana. Cuando esa expresión booleana resulta `#t`, entonces se ejecuta una parte del código; en cambio, cuando la expresión booleana es `#f` se ejecuta un código alternativo.

Código 1.7: *La implicación*

```
; (-> p q) ↦ booleano?
; p : booleano?
; q : booleano?
; Ej. (-> (= 4 4) (= 4 3)) ↦ #t.
(define ->
  (λ (p q)
    (if p q #t)))
```

■ **Ejemplo 1.12** Modele en `DrRacket` el siguiente enunciado y evalúe la expresión: ¿Qué valor de verdad tiene el enunciado «Si la manzana es fruta, entonces es saludable»? Considerando que es verdad que la manzana es una fruta y que también es verdad que las manzanas son alimentos saludables:

```
> (define manzana-es-fruta #t)
> (define manzana-es-saludable #t)
> (-> manzana-es-fruta manzana-es-saludable)
#t
>
```

■

1.3 Cuantificadores

Los cuantificadores universal y existencial son predicados especiales. Estos predicados requieren evaluar otro predicado con cada instancia de un conjunto. Para determinar el valor de verdad final, se requiere considerar el valor de verdad obtenido en cada evaluación.

1.3.1 Cuantificador universal

El cuantificador universal es equivalente a la conjunción extendida con un mismo predicado aplicado a una colección de operandos. Si se tiene un predicado

`(λ (i) (pred? i))` y una colección de operandos `(list` i_1 i_2 `...)`. El valor de verdad del cuantificador universal se obtiene al hacer la conjunción extendida de `pred?` en cada operando de la colección.

`(y*` `(pred?` i_1`)` `(pred?` i_2`)` ...`)`

El cuantificador universal también se conoce como *para-todo*, ya que la idea central es que un mismo predicado sea cierto para todo operando en la colección. La colección de operandos se conoce como **dominio**.

En la notación convencional de matemáticas, el cuantificador universal se denota con el símbolo / ∀/ y se utiliza con un predicado y un dominio, como se muestra en el siguiente patrón

$$\forall x \in D : P(x)$$

El dominio D contiene todos los valores con los que x debe ser instanciada en la expresión $P(x)$, donde P es un predicado.

En `DrRacket` la expresión primitiva `andmap` aplica la primitiva `map` a una lista de proposiciones y devuelve `#t` si todas las proposiciones son `#t`, de otro modo devuelve `#f`, pero lo hace de manera más eficiente, de modo que es suficiente definir un sinónimo más adecuado al lenguaje español.

Código 1.8: *Cuantificador universal*

```
1 ; (paraTodo proc lst ...+) ↦ lista?
2 ; proc : procedure? ; es el predicado
3 ; lst : lista? ; es la lista de dominios
4 (define paraTodo andmap)
```

El contrato de `paraTodo` es el mismo que el contrato de `andmap`:

```
(andmap proc lst ...+) ↦  cualquier
; proc : predicado?
; lst : lista?
```

Una restricción en el uso de `andmap` y `paraTodo`, así como para `ormap` y `existeUn` (siguiente tema), es que todos los dominios `lst ...+` deben tener la misma cantidad de elementos y el número de dominios debe coincidir con la aridad del predicado `pred`, que es precisamente número de literales utilizadas en el predicado. La razón es simple, si tuvieran diferente cardinalidad, ocurriría el caso de intentar aplicar un predicado con valores no definidos, ocasionando un error.

■ **Ejemplo 1.13** La siguiente interacción muestra cómo se utiliza el cuantificador universal con un dominio, luego con más de un dominio y también con un dominio vacío.

1. $\forall x \in \{10,20,30,40\} : x \geq 10 \mapsto$ `#f`.
2. $\forall x \in \{5,6,7,8\}, y \in \{5,4,3,2\} : x+y = 10 \mapsto$ `#t`.
3. $\forall x \in \{\} : x = 10 \mapsto$ `#t`.

Las siguientes interacciones corresponden a los mismos ejercicios de la lista anterior, traducidos al lenguaje `DrRacket`:

```
> (paraTodo (λ (x) (>= x 10)) '(10 20 30 40))
#t
> (paraTodo (λ (x y) (= (+ x y) 10)) '(5 6 7 8) '(5 4 3 2))
#t
> (paraTodo (λ (x) (= x 10)) '())
#t
```

■

1.3.2 Cuantificador existencial

El cuantificador existencial es un predicado que requiere dos elementos para determinar su valor de verdad, un predicado y un dominio de operandos. El predicado que es pasado como argumento se evalúa con cada operando del dominio y el valor de verdad será `#t` si existe al menos un operando del dominio que verifica el predicado; y será `#f` si no hay tal operando en el dominio.

La idea general del cuantificador existencial es determinar el valor de verdad mediante una disyunción extendida, con los operandos del dominio. Supongamos que `(λ (x) (pred? x))` es el predicado; y `(list` i_1 i_2...`)` es la colección de operandos, el cuantificadoe existencial tiene el mismo comportamiento que la disyunción extendida:

`(o* (pred?` i_1`) (pred?` i_2`) ...)`

En `DrRacket` se ha implementado la primitiva `ormap` que realiza esta función, que es un predicado que es capaz de trabajar con predicados de más de una variable.

Código 1.9: *El cuantificador existencial*

```
; (existeUn pred lst ...+) ↦ booleano?
; pred : predicado?
; lst : lista? ; requiere una o más listas de la misma cardinalidad
(define existeUn ormap)
```

`ormap` es similar a `map`, pero devuelve el resultado de evaluar `or` con cada expresión del dominio.

La notación matemática convencional para el cuantificador existencial es $\exists x \in D : P(x)$, con D el dominio y P un predicado. La lectura de esta notación es que el cuantificador existencial será cierto, si existe un operando x en el dominio D, tal que se verifique el predicado P en tal operando x.

El cuantificador existencial es equivalente a una disyunción extendida, pero aplicando el mismo predicado a cada operando del dominio.

$$\bigvee_{x \in D} P(x) = P(x_1) \vee P(x_2) \vee \ldots.$$

■ **Ejemplo 1.14** La siguiente interacción muestra cómo utilizar el cuantificador existencial con un dominio, con más de un dominio y con un dominio vacío.

1. $\exists x \in \{10,20,30,40\} : x \geq 10 \mapsto$ `#t`.
2. $\exists x \in \{5,6,7,8\}, y \in \{2,3,4,5\} : (x+y) = 10 \mapsto$ `#f`.
3. $\exists x \in \emptyset : x = 10 \mapsto$ `#f`.

Las siguientes interacciones corresponden a los mismos ejercicios de la lista, traducidos al lenguaje DrRacket:

```
> (existeUn (λ (x) (>= x 10)) '(10 20 30 40))
#t
> (existeUn (λ (x y) (= (+ x y) 10) '(5 6 7 8) '(2 3 4 5)))
#f
> (existeUn (λ (x) (= x 10)) '())
#f
```

■

Cuantificador existencial único

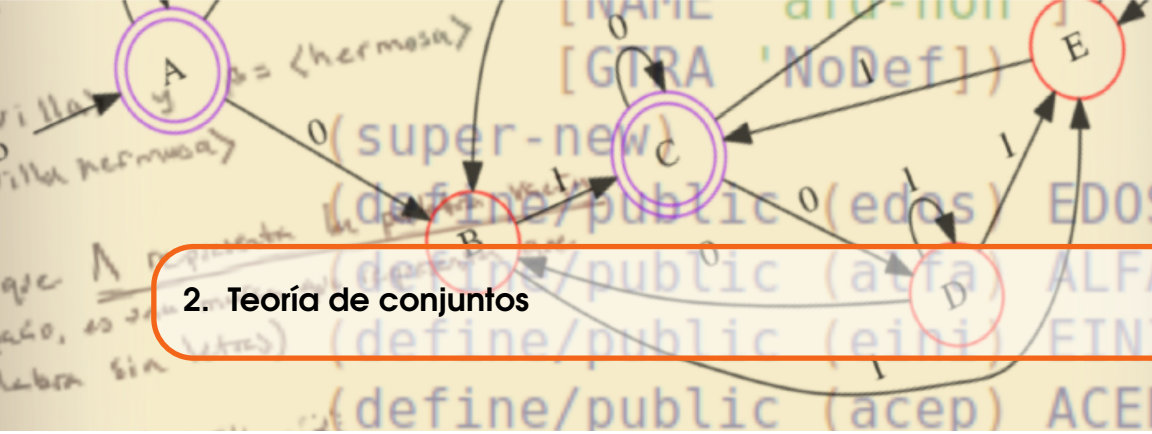
Este cuantificador es un caso especial del cuantificador existencial. El comportamiento es como el cuantificador existencial, pero la cantidad de elementos que hacen `#t` el predicado es exactamente 1.

■ **Ejemplo 1.15** Existe una única persona `p` en el dominio `'(damian eugenia arturo maria jose alejandro aurora)` que se llama `'arturo`. Se observa que además de que el cuantificador existencial es `#t`, el cuantificador existencial único también es `#t`. ■

La implementación de una función en DrRacket que modele el comportamiento del cuantificador existencial único debe considerar ambos requisitos:

1. Que el cuantificador existencial sea `#t`.
2. Que la cantidad de elementos que hacen `#t` el predicado, sea exactamente uno.

El código de este caso especial de cuantificador existencial se deja como ejercicio para el lector.



2. Teoría de conjuntos

Un conjunto es una agrupación de objetos. Los objetos que son considerados parte del conjunto se dice que **pertenecen** al conjunto y que son **elementos** del conjunto.

Para nombrar los conjuntos se utilizan identificadores que inician con letra mayúscula, como `A`, `B`, `S`, `Alfabeto`, `S1`, `Q3`; las letras minúsculas con apóstrofo son los elementos de los conjuntos, como `'e`, `'x`, `'arturo`, `'sopa`. Cuando se requiera crear un identificador que adquiera como significado un elemento de un conjunto, se escribirá sin apóstrofo, porque se trata de un símbolo de valor variable, como en `s↤'s`, donde se entiende que una variable identificada con el símbolo `s`, tomará el valor `'s`. La escritura `'s↤'s` no tiene sentido, pues `'s` no es una variable a la que se le pueda asignar valor alguno. Por otro lado la notación `s↤s`, causará un error si es la primera vez que se define `s` ya que un identificador no puede ser definido más de una vez.

2.1 Creación de conjuntos

Los conjuntos pueden ser creados de manera **implícita**, al dar una regla que determine qué elementos contiene el conjunto; o bien de manera **explícita** al enlistar a cada elemento del conjunto.

La notación convencional en matemáticas para los conjuntos, es encerrar los elementos entre llaves, /{/ y /}/. Dentro del ámbito del conjunto se escribe, en el caso de un conjunto implícito, una variable y un predicado, separados por un símbolo /|/ o /:/; y en el caso de los conjuntos explícitos, una lista con cada uno de los elementos que deben ser considerados en el conjunto, como:

$$\{x|1 \leq x \leq 5\}\text{; De manera implícita.}$$
$$\{1,2,3,4,5\}\text{; De manera explícita.}$$

■ **Ejemplo 2.1** Supongamos que deseamos crear un conjunto...:

1. ... Llamado `C1`, con los números no negativos menores que o iguales a `100` menores que `20`.

```
(define C1 (filter (λ (n) (y (esPar? n) (< n 20)))
                   (build-list 101 values)))
```

Compare esta definición con la expresión:

$$C_1 = \{x \in \mathbb{Z}_0^{100} | esPar(x) \wedge x < 20\}$$

`filter` selecciona elementos de una lista que verifican un predicado:

```
(filter proc lst)
; proc : procedimiento
; lst : (listade cualquier)
```

El predicado `esPar?` es un sinónimo de `even?`, devuelve `#t` si su argumento es un número par:

```
(define esPar? even?)
```

2. ... Que se llame `C2` y que contenga los números 0, 2, 4, 6, 8, 10, 12, 14, 16, 18. Esta es una forma explícita. En `DrRacket` escribimos

```
(define C2 '(0 2 4 6 8 10 12 14 16 18))
```

Compare una vez más con la definición convencional.

$$C_2 = \{0,2,4,6,8,10,12,14,16,18\}$$

3. ... Que se llame `C3` y que no tenga elementos. Esto es el conjunto vacío.

```
(define C3 vacio)
```

Y su definición convencional para su comparación:

$$C_3 = \emptyset$$

```
> (define C1  (filter (λ (n) (y (esPar? n) (< n 20)))
                      (build-list 101 values)))
;; (λ (n) (y (even? n) (<n 20)))
;; Es una expresión lambda que es un predicado.
;; Verifica que un número n sea par y menor que 20.
;; (build-list 101 values)
;; crea una lista de 101 números, desde 0 hasta 100.
>  C1
'(0 2 4 6 8 10 12 14 16 18)
> (define C2  '(0 2 4 6 8 10 12 14 16 18))
>  C2
'(0 2 4 6 8 10 12 14 16 18)
> (define vacio '())
> (define C3  vacio)
>  C3
'()
```

■

Para crear un conjunto con cualquier cantidad de elementos de cualquier tipo, se utilizará la primitiva `list`.

Código 2.1: *Definición de conjuntos*

```
1 ;; Se utiliza list como generador de conjuntos.
2 ;; Se crea un nombre adecuado para llamarlo.
3 (define conj list); conj es un sinónimo de list.
```

■ **Ejemplo 2.2** Sea A el conjunto $\{2,4,6,8,10\}$. El conjunto A puede crearse en `DrRacket` utilizando el nuevo procedimiento `conj`:

```
> (define A (conj 2 4 6 8 10))
> A
'(2 4 6 8 10)
```

■

2.2 Conjunto vacío

Un conjunto que no tiene elementos es un conjunto vacío. En `DrRacket`, el conjunto vacío se modela como una lista sin elementos:

Código 2.2: *Definición del conjunto vacío*

```
1 ; Se utiliza la lista vacía como el conjunto vacío.
2 ; Se crea un nombre adecuado para llamarlo.
3 (define vacio '()); Se define el conjunto vacío.
```

Junto con `vacio`, definiremos el predicado `vacio?` para determinar si un conjunto es vacío o no. Si `A` es el conjunto vacío, la evaluación del predicado `(vacio? A)` $\mapsto$ `#t`; por el contrario, si `(vacio? A)` $\mapsto$ `#f` entonces `A` no es vacío

y tiene al menos un elemento. DrRacket cuenta con el predicado empty? que verifica que una lista no tenga elementos.

Código 2.3: *Predicado para verificar el conjunto vacío*

```
1 ; Se define un sinónimo para verificar el conjunto vacío.
2 ; (vacio? A) ↦ booleano?
3 ; A : conjunto?
4 (define vacio? empty?)
```

■ **Ejemplo 2.3** Verifique que '() es vacío; y que el conjunto '(2 i w 8 4) no lo es.

```
> (vacio? '())
#t
> (vacio? (conj 2 i w 8 4))
#f
>
```

■

Convencionalmente se usa el símbolo ∅ para denotar el conjunto vacío, también suele escribirse {}.

2.3 Pertenencia

Un elemento **pertenece** a un conjunto si se cumple una de las siguientes condiciones:

1. Está entre los elementos enlistados en el conjunto explícitamente definido.
2. Cumple con las características especificadas en un conjunto implícitamente definido.

En la notación convencional se usa /∈/ para denotar la pertenencia y se utiliza con dos diferentes significados que difieren por el contexto. Una expresión como "*dado que* $a \in A$ ..." se interpreta como proposición; en cambio, en una expresión como "*si* $a \in A$, *entonces* ...", el valor de la pertenencia aún no está determinado.

Se requiere ahora un procedimiento para verifiar la pertenencia de un elemento a un conjunto (modelado como una lista), recordemos que las listas:

(a) Si no tienen elementos, son de la forma '().

(b) Si tiene al menos un elemento, una lista siempre tiene dos partes:

(I) El primer elemento de la lista, que es el car de la lista,

(II) El resto de los elementos de la lista, que es el cdr de la lista.

En DrRacket hay primitivas que son más cercanas al significado de car y cdr, son first para obtener el primer elemento de una lista y el rest para el resto excepto el primero. Se conserva car y cdr por razones históricas [McC78].

En este texto se denota un conjunto no vacío de dos formas:

1. (a$_1$|A'), donde a$_1$ es el car del conjunto y A' es el cdr del conjunto. Esta notación hace énfasis en que el conjunto tiene un primer elemento y una lista que contiene al resto de los elementos.
2. '(a$_1$ ... a$_n$), indica que el conjunto tiene n elementos, enumerados desde 1 hasta el número asociado con n.

Código 2.4: *La pertenencia de un elemento a un conjunto*

```
; (en? a A) ↦ booleano?
; a : cualquier
; A : conjunto?
; (en? 'a '(e d s a w)) ↦ #t
(define en?
  (λ (a A)
    (existeUn (λ (x) (equal? a x)) A)
```

■ **Ejemplo 2.4** Sea C el conjunto $\{5,4,8,7,0,9\}$. Determine si 5 y 2 son elementos de C.

```
> (define C (conj 5 4 8 7 0 9))
> (en? 5 C)
#t
> (en? 2 C)
#f
>
```

■

DrRacket tiene une sistema de índices basado en 0 para los elementos de las listas, la primitiva list-ref permite obtener el elemento de una lista que se encuentra en una posición específica.

```
(list-ref lst pos) ↦  cualquier
; lst : lista?
; pos : numero-entero-no-negativo?
```

2.3.1 Cardinalidad

La cardinalidad de un conjunto es justamente el número de elementos que contiene. El conjunto vacio tiene cardinalidad 0, porque no tiene elementos.

En notación convencional se establece $|A|$ para denotar al número entero no negativo que representa la cardinalidad de un conjunto llamado A. El operador $|\cdot|$ sirve para otras operaciones comunes en matemáticas, de modo que es necesario considerar el tipo del operando para determinar la naturaleza de la operación $|\cdot|$.

Si A ↦ '(a$_1$ ... a$_n$), el conjunto A es **finito** y contiene n diferentes elementos, con n un número entero no negativo. También A es de orden n. El vacio es de orden 0. Si A no es finito, entonces es **infinito**. [MH81]

Ya que los autómatas finitos tratan con conjuntos finitos, se creará un porcedimiento para contar los elementos de un conjunto finito y explícitamente definido:

1. Un conjunto definido de manera explícita, que podría o no, ser el conjunto vacío.
2. De manera auxiliar, se requiere una variable que contenga la cardinalidad inicial del conjunto, cuyo valor debe ser 0 al inicio del procedimiento. Este valor se utilizará como repositorio del resultado. Tener parámetros con la finalidad de construir progresivamente la respuesta es un modo de crear procesos iterativos dentro de funciones recursivas [ASS96].

Código 2.5: *La cardinalidad de un conjunto*

```
; (card A [res]) ↦ numero?
; A : conjunto?
; res : numero? = 0 -- es un parámetro opcional, con valor 0 en omisión
(define card
  (λ (A [res 0])
    (if (vacio? A)
        res
        (card (cdr A) (+ res 1)))))
```

En DrRacket se pueden definir **argumentos opcionales** que consisten de un identificador con un valor inicial. Se definen dentro de una λ-expresión:

```
(λ (arg ... [arg-opc val-ini] ....)
  cuerpo ...+)
```

Esto significa que al definir la lista de argumentos formales, es posible declarar 0 o más argumentos obligatorios, seguidos de 0 o más argumentos opcionales, cada uno de ellos con su valor inicial. Cuando se invoca el procedimiento omitiendo los argumentos opcionales, se utiliza el valor por defecto. Pero cuando se invoca el procedimiento utilizando explícitamente los argumentos opcionales, se utiliza el valor dado en la llamada.

■ **Ejemplo 2.5** Uso de argumentos opcionales dentro de la función `card`:

```
> (card (conj 1 2 3 x y z)) ; La variable res vale 0.
6
> (card (conj 1 2 3 x y z) 0) ; El valor de res es 0.
6
> (card (conj 1 2 3 x y z) 1) ; El valor de res ahora es 1.
7
>
```

■

Cuando `(card A)` ↦ `1`, es decir que el conjunto `A` tiene solo un elemento, `A` es un **conjunto unitario**, también se puede decir que es de **orden** 1. El conjunto `(conj (conj ))` es unitario, se le invita a comprobarlo mediante una interacción.

Código 2.6: *Predicado para determinar si un conjunto es unitario*

```
; (unitario? C) ↦ booleano?
; A : conjunto?
(define unitario?
  (λ (A)
    (= (card A) 1)))
```

■ **Ejemplo 2.6** Verifique que `'(0)`, `'(s)` y `'(())` sean conjuntos unitarios.

```
> (unitario? (conj 0))
#t
> (unitario? (conj s))
#t
> (unitario? (conj '()))
#t
```

■

2.4 Operaciones con conjuntos

En esta sección se muestra cómo comparar, crear, modificar y cómo combinar conjuntos para formar nuevos conjuntos con características particulares.

2.4.1 Subconjuntos

Si `A` y `B` son conjuntos, decimos que el conjunto `A` es subconjunto del conjunto `B`, cuando todos los elementos de `A` pertenecen al conjunto `B`.

En la notación matemática usual, podemos escribir $A \subseteq B$ para decir que es cierto que A es un subconjunto de B. Es decir, que la notación $\subseteq$ sirve como una función booleana, ya que el valor devuelto es `#t` o `#f`.

Esta definición puede ser implementada en `DrRacket` considerando la pertenencia de un elemento a un conjunto (ver la página 35) y un cuantificador universal, ya que se debe verificar la pertenencia para todo elemento de `A` al conjunto `B`.

Código 2.7: *Subconjunto de un conjunto*

```
; (subc? A B) ↦ booleano?
; A : conjunto?
; B : conjunto?
(define subc?
  (λ (A B)
    (paraTodo (λ (a) (en? a B)) A))) ;∀(a ∈ A) : a ∈ B
```

■ **Ejemplo 2.7** Si `A` ↦ `'(1 2 3)` y `B` ↦ `'(3 8 2 9 1)` son conjuntos, `A` es subconjunto de `B` porque todos sus elementos (los de `A`), pertenecen también a `B`.

Si ahora `A` ↦ `'(1 2 3)` y `B` ↦ `'(1 2 7 8 9)`, `A` no es subconjunto de `B`, porque no todos los elementos de `A` pertenecen también a `B`.

```
> (subc? '(1 2 3) '(3 8 2 9 1))
#t
> (subc? '(1 2 3) '(7 8 2 9 1))
#f
> (subc? '() '(3 8 2 9 1))
#t ;
> (subc? '() '())
#t ;
```

■

Dos resultados importantes son:

1. El conjunto vacío es subconjunto de cualquier conjunto.
2. Cualquier conjunto es subconjunto de él mismo.

Subconjunto propio

Si `A` y `B` son conjuntos y `(subc? A B)` $\mapsto$ `#t`, es decir que es cierto que `A` es subconjunto de `B`, pero también sucede que `B` tiene al menos un elemento que no pertenece al conjunto `A`, se dice entonces que `A` es un **subconjunto propio** del conjunto `B`.

En otras palabras, `A` es un subconjunto propio de `B` si `A` es un subconjunto de `B` y existe al menos un elemento de `B` que no esté en `A`. Una clara consecuencia de esta definición, es que la cardinalidad de `B` es estrictamente mayor que la cardinalidad de `A`.

☞ En la notación convencional, si A es un subconjunto propio de B, se denota como $A \subset B$, note la diferencia con la notación de subconjunto, $A \subseteq B$, donde pudiera ocurrir que $|A| = |B|$.

Código 2.8: *Subconjunto propio de un conjunto*

```
; (subcPropio? A B) ↦ booleano?
; A : conjunto?
; B : conjunto?
(define subcPropio?
  (λ (A B) ;<-- dos conjuntos en forma de lista.
    (y (subc? A B)
       (existeUn (λ (b) (neg (en? b A))) B))))
```

■ **Ejemplo 2.8** Verifique que el conjunto `(conj 1 2 3)` sea un subconjunto propio del conjunto `(conj 7 3 8 2 9 1)`:

```
> (subcPropio? (conj 1 2 3) (conj 7 3 8 2 9 1))
#t
>
```

■

2.4.2 Igualdad de conjuntos

Sean `A` y `B` dos conjuntos. Decimos que `A` y `B` son **conjuntos iguales** si ambos contienen exactamente los mismos elementos. Para asegurar que ambos tienen los mismos elementos, `A` debe ser un subconjunto de `B` y `B` no debe tener elementos que no pertenezcan a `A`. Para determinar la igualdad entre dos conjuntos (definidos extensionalmente), debemos hacer dos cosas:

1. Verificar que `(subc? A B)` $\mapsto$ `#t`. Esto nos dirá que todos los elementos de `A` pertenecen a `B`.
2. Asegurar que «no es cierto que exista un elemento en `B`, que no pertenezca al conjunto `A`»; de manera equivalente se puede expresar como «es verdad que todos los elementos de `B` pertenecen al conjunto `A`», lo que en términos convencionales se escribe como:

$$\forall x \in B : x \in A,$$

y en `DrRacket` se escribe:

`(paraTodo (λ (x) (en? x A)) B)` $\mapsto$ `#t`,

es decir, `(subc? B A)` $\mapsto$ `#t`.

El símbolo `c=?` es el nombre asignado al procedimiento para determinar si dos conjuntos son iguales.

Código 2.9: *Conjuntos iguales*

```
; (c=? A B) ↦ booleano?
; A : conjunto?
; B : conjunto?
(define c=?
  (λ (A B)
    (y (subc? A B) (subc? B A))))
```

■ **Ejemplo 2.9** Determinar el valor de verdad de las siguientes comparaciones:

1. $\{2,1,3\} = \{2,3,1\}$
2. $\{2,1,3\} = \{2,3,1,4\}$
3. $\{2,1,3,4\} = \{2,3,1\}$
4. $\emptyset = \emptyset$

```
> (c=? '(2 1 3) '(2 3 1))
#t
> (c=? '(2 1 3) '(2 3 1 4))
#f
> (c=? '(2 1 3 4) '(2 3 1))
#f
> (c=? '() vacio) ; El identificador vacio es definido en la p. 33.
#t
>
```

■

2.4.3 Agregar un elemento a un conjunto

Si `A` $\mapsto$ `vacio`. Para agregar un elemento `a` al conjunto, se hace un procedimiento que genera un nuevo conjunto con todos los elementos de `A`, junto con el nuevo elemento `a` si es que no pertenecía ya al conjunto.

La definición de un procedimiento para agregar un elemento `a` a un conjunto `A`, llamado `agregar`, tiene las siguientes condiciones de entrada y salida.

- **Entrada:** El procedimiento `agregar` recibe un elemento `a` de cualquier tipo; y un conjunto `A` que puede o no ser `vacio`.
- **Salida:** Debe devolver un conjunto (en forma de una lista), que contenga como primer elemento al candidato `a` siempre y cuando el elemento no pertenezca ya al conjunto `A`; y el resto del conjunto `A`.

Antes de agregar `a` al conjunto `A`, se debe verificar `(en? a A)`; sin embargo hay situaciones en las que se conoce de antemano que el elemento no pertenece al conjunto, vale el esfuerzo de permitir no hacer la verificación; el no hacerlo permitirá que el tiempo de ejecución del procedimiento disminuya considerablemente.

Código 2.10: *Agregar un elemento a un conjunto*

```
; (agregar e A) ↦ conjunto?
; e : cualquier
; A : conjunto?
(define agregar
  (λ (e A #:chk [chk #t])
    (if (y chk (en? e A))
        A
        (cons e A))))
```

`cons` recibe dos argumentos de cualquier tipo `e` y `L`; y produce un par de elementos, si `L` es una lista, entonces el nuevo elemento `e` se agrega a la lista como `car` y los elementos de `L` como el `cdr` de la nueva lista.

```
(cons e L) ↦ par?
; e : cualquier
; L : cualquier
```

`DrRacket` permite también definir argumentos en forma de *palabras clave*. Una palabra clave permite condicionar el comportamiento del procedimiento en dependencia del valor de la palabra clave. A diferencia de los argumentos convencionales, las palabras clave se pueden colocar en cualquier orden, ya que en la invocación se requiere indicar el nombre de la palabra clave. Una palabra clave se declara en la λ-expresión, generalmente después de los argumentos obligatorios y optativos, tiene el siguiente formato:

```
(λ ( ... arg-palclv [arg-id val-ini] ...)
   cuerpo ...+ )
```

donde `arg-palclv` es de la forma `#:`, seguido de un identificador; entre los corchetes, el `arg-palclave` es el identificador opcional dentro del procedimiento y que tiene un valor inicial dado por `val-ini`.

■ **Ejemplo 2.10** Las siguientes interacciones ejemplifican el uso de la función `agregar`, utilizando el argumento clave `#:chk`, que sirve para determinar si se hace una verificación de pertenencia o no se hace. No hacer la verificación acelera el proceso, pero si el elemento pertenecía ya al conjunto, entonces se duplicará.

```
> (agregar 'a (conj 2 k s o a 3) #:chk #f)
'(a 2 k s o a 3)
> (agregar 'a (conj 2 k s o a 3) #:chk #t)
'(2 k s o a 3)
> (agregar 'a (conj 2 k s o a 3))
'(2 k s o a 3)
>
```

■

2.4.4 Unión de conjuntos

Si `A` y `B` son conjuntos, la unión de `A` con `B` genera un nuevo conjunto con los elementos de `A` y `B`. El nuevo conjunto contendrá a todos los elementos `B` junto con aquellos elementos de `A` que no pertenezcan a `B`.

Convencionalmente se define la unión de los conjuntos A y B como

$$A \cup B = \{x | x \in A \vee x \in B\}$$

Esta definición es un claro ejemplo de *definición declarativa* ya que declara qué características debe cumplir cada elemento del conjunto; esto es a diferencia de las *definiciones efectivas* que suelen escribirse en computación, donde se menciona cómo se obtiene cada elemento.

El procedimiento `union` debe considerar tres posibles casos:

1. Si `(vacio? A)` $\mapsto$ `#t`, entonces se devuelve el conjunto `B`.
2. Si el primer elemento de `A` pertenece a `B`, entonces no hay necesidad de agregarlo y debe continuar el proceso con el resto de los elementos de `A`.
3. De otro modo se agrega el primer elemento de `A` al conjunto `B` sin verificar la pertenencia.

Código 2.11: *La unión de dos conjuntos*

```
1 ;; Unión de dos conjuntos
2 ; (union A B) ↦ conjunto?
3 ; A : conjunto?
4 ; B : conjunto?
5 (define union
6   (λ (A B)
7     (cond ((empty? A) B) ; Caso 1.
8           ((en? (car A) B) (union (cdr A) B)) ; Caso 2.
9           (else (union (cdr A)
10                       (agregar (car A) B
11                                #:chk #f))))))); Caso 3.
```

■ **Ejemplo 2.11** Se muestra el uso de `union` con dos conjuntos en diferentes circunstancias:

```
> (union '(2 4 6 8 10) '(6 7 8 9 10))
'(4 2 6 7 8 9 10)
> (union '() '(6 7 8 9 10))
'(6 7 8 9 10)
> (union '(2 4 6 8 10) '())
'(10 8 6 4 2)
> (union '(2 4 6 8 10) '(2 4 6 8 10))
'(2 4 6 8 10)
>
```

■

La forma `cond` enlista una serie de casos; cada caso se compone de una expresión lógica, con una expresión. La forma `cond` es evaluada con el valor de la expresión cuya expresión booleana haya sido la primera en verificarse.

```
(cond (expr-log expr) ...)
; expr-log : expresión lógica
; expr : S-expresión
```

Cada expresión lógica se evalúa en secuencia desde la más cercana a la palabra `cond`; si el resulta `#f`, se continúa con la siguiente expresión lógica; si el resultado fue `#t`, entonces se evalúa la expresión que la acompaña y termina el proceso de `cond`. Si ninguna de las expresiones lógicas resultó ser `#t`, entonces el resultado es un valor especial llamado `#<void>`. La expresión lógica `else`, debe estar en el último par, ya que tiene valor `#t`, sirve para considerar "cualquier otro caso"; de no ser la última ocasiona un error.

2.4.5 Union extendida de una lista de conjuntos

Es posible extender el concepto de unión de conjuntos al considerar la unión de una lista de conjuntos. Si A_1, A_2, ... , A_n son conjuntos, podemos definir un procedimiento que nos permita obtener la unión de todos ellos, lo que se denominará la **unión extendida** y denotaremos como `union*`

Nótese que la unión extendida de ningún conjunto es el `vacio`, y progresivamente se unirán uno a uno los conjuntos de la lista de argumentos, construyendo el resultado en un conjunto temporal llamado `res`.

En la notación convencional de matemáticas se suele utilizar un símbolo de unión más grande, especificando con un subíndice los conjuntos a ser unidos y con un superíndice el alcance de la unión extendida.

$$\bigcup_{i=1}^{n} A_i = A_1 \cup A_2 \cup \cdots \cup A_n = (A_n \cup (\ldots (A_2 \cup (A_1 \cup \emptyset)) \ldots))$$

Código 2.12: *La unión extendida*

```
; Unión extendida.
; (union* conj ...) ↦ conjunto?
; conj : (listade lista?)
(define union*
  (λ LC ;<-- una lista indefinida
    (if (vacio? LC)
        '()
        (union*aux LC))))
; (union*aux LC [res]) ↦ conjunto?
; LC : (listade lista?)
; res : conjunto? = '()
(define union*aux
  (λ (LC [res '()])
    (if (vacio? LC)
        res
        (union*aux (cdr LC) (union res (car LC))))))
```

■ **Ejemplo 2.12** Calcule la unión $\{a\} \cup \{f,a,g,h\} \cup \{c,d,f,a\}$.

```
> (union* '(a) '(f a g h) '(c d f a))
'(a c d f g h)
>
```

■

La unión extendida de ningún conjunto produce el conjunto vacío.

■ **Ejemplo 2.13**

```
> (union*)
'()
>
```

■

2.4.6 Intersección de conjuntos

De la intersección de los conjuntos `A` y `B` resulta un nuevo conjunto con los elementos que pertenecen tanto al conjunto `A` como al conjunto `B` al mismo tiempo.

La notación convencional en matemáticas se basa en el predicado:

$$A \cap B = \{x | x \in A \wedge x \in B\}$$

Si alguno de los conjuntos es vacío, ya sea que $A = \emptyset$ o $B = \emptyset$, es claro que no hay elementos compartidos, por lo que $A \cap B = \emptyset$. Por otro lado, si $A = B$, la intersección de ambos conjuntos tiene los mismos elementos que A [y que B].

Con el propósito de crear un procedimiento efectivo para calcular la intersección de los conjuntos `A` y `B`, si elegimos uno de los conjuntos, digamos el conjunto `B`, los elementos de la intersección serán solamente aquellos elementos de `B` que pertenezcan al conjunto `A`. De este razonamiento podemos crear un procedimiento que seleccione solamente tales elementos del conjunto `B`.

Código 2.13: *La intersección de dos conjuntos*

```
; (intersec A B) ↦ conjunto?
; A : conjunto?
; B : conjunto?
(define intersec
  (λ (A B)
    (filter (λ (b) (en? b A)) B)))
```

■ **Ejemplo 2.14** Calcular las siguientes intersecciones:

1. $\emptyset \cap \{1,2,3,4,5\}$
2. $\cap\{1,2,3,4,5\} \cap \emptyset$
3. $\{1,2,3,4,5\} \cap \{1,3,5,7,9\}$

```
> (intersec '() '(1 2 3 4 5))
'()
> (intersec '(1 2 3 4 5) '())
'()
> (intersec '(1 2 3 4 5) '(1 3 5 7 9))
'(5 3 1)
>
```

■

2.4.7 Diferencia de un conjunto respecto de otro

La diferencia del conjunto `A` respecto del conjunto `B` es un nuevo conjunto que contiene a los elementos de `A` que no pertenecen al conjunto `B`.

La notación convencional en matemáticas la diferencia de conjuntos se define:

$$A \backslash B = \{x | x \in A \wedge x \notin B\}.$$

Generalmente la diferencia de conjuntos no es una operación conmutativa.

La diferencia de un conjunto `A` respecto a otro conjunto `B`, también se le puede llamar el **complemento** de `A` respecto a `B`. Frecuentemente se hace mención de el complemento de un conjunto (sin mencionar respecto de qué otro conjunto), suponiendo la existencia de un conjunto universal $\mathbb{U}$ previamente definido, que contiene todos los elementos que pudieran ser considerados. El procedimiento `difc` selecciona los elementos de `A` que no pertenecen a `B`:

Código 2.14: *La diferencia de conjuntos*

```
;; Diferencia de dos conjuntos
; (difc A B) ↦ conjunto?
; A : conjunto?
; B : conjunto?
(define difc
  (λ (A B)
    (filter-not (λ (a) (en? a B)) A)))
```

■ **Ejemplo 2.15** Calcule las siguientes diferencias entre conjuntos:

1. $\{a,b,c,d,e,f\}\backslash\{k,w,s,n,d,a\}$
2. $\{k,w,s,n,d,a\}\backslash\{a,b,c,d,e,f\}$

```
> (difc '(a b c d e f) '(k w s n d a))
'(f e c b)
> (difc '(k w s n d a) '(a b c d e f))
'(k w s n)
>
```

■

2.5 Conjunto potencia

El conjunto potencia del conjunto `A` es el conjunto que contiene todos los subconjuntos de `A`. Un procedimiento para obtener el conjunto potencia puede identificarse con `cPot`.

☞ La notación convencional para el conjunto potencia del conjunto A, suele escribirse 2^A, también como $\mathbb{P}(A)$ y se define como

$$\mathbb{P}(A) = \{A' | A' \subseteq A\}$$

Si `A` $\mapsto$ `'()`; `(cPot A)` $\mapsto$ `(cPot '())` $\mapsto$ `'(())`.

Con el fin de vislumbrar la manera de obtener el conjunto potencia de un conjunto, supóngase que `A` $\mapsto$ `'(1)`. Los únicos subconjuntos de `A`, son el `vacio` y el conjunto unitario `'(1)`, así

`A` $\mapsto$ `'(1)`
`(cPot A)` $\mapsto$ `(cPot '(1))` $\mapsto$ `'(() (1))`.

Ahora se agrega un nuevo elemento al conjunto: `A` $\mapsto$ `'(2 1)`:

`A` $\mapsto$ `'(2 1)`
`(cPot A)` $\mapsto$ `'(() (2) (1) (2 1))`.

Se crea el doble de subconjuntos respecto del conjunto potencia del conjunto anterior. Esta característica se mantiene con cada nuevo elemento.

`A` $\mapsto$ `'(3 2 1)`
`(cPot A)` $\mapsto$ `'(() (3) (2) (3 2) (1) (3 1) (2 1) (3 2 1))`

La figura 2.1 muestra cómo cada nuevo elemento genera un par de nuevos subconjuntos.

El conjunto potencia divide los conjuntos en dos. Hay un elemento que aparece en la mitad de los subconjuntos y no aparece en la otra mitad (observa la figura 2.2).

Si `A` $\mapsto$ `(`a_0`|A')` es un conjunto no vacío, `(cPot A)` se obtiene al unir los conjuntos de `(cPot A')` con los conjuntos que resultan de agregar a_0 en cada conjunto de `(cPot A')`.

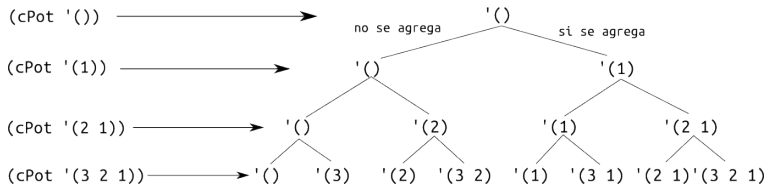


Figura 2.1: *El conjunto potencia de un conjunto con* `n` *elementos, tiene el doble de subconjuntos que el conjunto potencia de un conjunto con* `(- n 1)` *elementos.*

```
'() '(2) '(1) '(2 1)      '(3) '(3 2) '(3 1) '(3 2 1)
```

Figura 2.2: *Solo la mitad de los subconjuntos tienen al último elemento agregado.*

Código 2.15: *Función auxiliar para el conjunto potencia*

```
; (agregaEnCada e LC) ↦ (listade lista?)
; e : cualquier
; LC : (listade lista?)
(define agregaEnCada
  (λ (e LC)
    (map (λ (C) (agregar e C)) LC)))
```

■ **Ejemplo 2.16**

```
>(agregaEnCada 'a '((1) (2 3) (4 5 6)))
'((a 1) (a 2 3) (a 4 5 6))>
```

■

`map` es una primitiva que aplica un procedimiento a una o varias listas de argumentos. El contrato de `map` es

```
(map proc lst ...+) ↦ lista?
; proc : procedimiento?
; lst : lista?
```

El procedimiento para construir iterativamente el conjunto potencia de un conjunto `A`, considera dos casos:

1. Si `(vacio? A)` ↦ `#t`, entonces el conjunto potencia es el conjunto unitario, que contiene al `vacio`.
2. Si el anterior no es el caso, entonces el conjunto `A` es de la forma `(a1|A')` ; esto permitirá construir el conjunto potencia de `A` como la unión del conjunto potencia de `A'`, con lo que resulte de haber agregado en cada subconjunto del conjunto potencia de `A'`, el nuevo elemento a_1.

Código 2.16: *El conjunto potencia*

```
; (cPot A) ↦ (listade lista?)
; A : lista
(define cPot
  (λ (A [res '(())])
    (if (empty? A)
        res
        (cPot (cdr A) (append res (agregaEnCada (car A) res))))))
```

El procedimiento `append` anexa dos o más listas. El contrato es como sigue:

```
(append lst ...) ↦  lista?
; lst : lista?
```

■ **Ejemplo 2.17** Calcule el conjunto potencia del conjunto `'(1 2 3 4)`:

```
> (cPot '(1 2 3 4))
'(() (1) (2) (2 1) (3) (3 1) (3 2) (3 2 1) (4) (4 1) (4 2) (4 2 1) (4 3)
  (4 3 1) (4 3 2) (4 3 2 1))
>
```

■

2.6 Tuplas

Una **tupla** es una secuencia ordenada de elementos, no necesariamente únicos. Conceptualmente tupla es lo mismo que una **lista**, o **palabra**. Las tuplas reúnen valores de diferente tipo en una secuencia. Así por ejemplo, un objeto que sea de `10` unidades de largo, que sea de color `'azul` y tenga un modo «encendido» que esté activado, puede representarse mediante una tupla ya sea como ⟨`10`,`'azul`,`#t`⟩, o bien utilizando una notación más parecida a `DrRacket` como `(tupla 10 'azul #t)`. El siguiente código muestra una manera efectiva de crear tuplas con una lista no definida de valores.

Código 2.17: *Crea una tupla con cualquier número de elementos*

```
;; (tupla lst ... ) ↦ lista?
; lst : cualquier ...
(define tupla
  (λ lst ; Es una lista indeterminada de elementos.
    lst)) ; Devuelve los valores en una lista determinada.
```

Con el predicado `list?` que es una primitiva de `DrRacket`, se verifica también si un objeto es una tupla. En `DrRacket` es un hecho que `'()` es una tupla, se trata de la tupla vacía. Si una tupla no es la tupla vacía, entonces tiene al menos un primer elemento y una tupla que tiene al resto de los elementos.

Código 2.18: *Es tupla?*

```
; (tupla? t) ↦ booleano?
; t : cualquier
(define tupla?
  (λ (t)
    (list? t)))
```

En DrRacket, las tuplas, listas y vectores son tratados como secuencias. Hay una gran variedad de secuencias que se utilizan para fines particulares. El lector debe revisar en la referencia del lenguaje la entrada «*sequences*» `https://docs.racket-lang.org/reference/sequences.html`.

■ **Ejemplo 2.18** Cree la tupla ⟨a,b,b,a,a,a,c⟩ y verifique que es una tupla.

```
> (define t (tupla a b b a a a c))
> (tupla? t)
#t
>
```

■

La **longitud** de una tupla `t` es la cantidad de elementos que contiene, sin importar si hay elementos repetidos o no.

Código 2.19: *Longitud de una tupla*

```
; (tlong t) ↦ numero-entero-no-negativo?
; t : tupla
(define tlong
  (λ (t)
    (card t)))
```

Se puede verificar que un objeto que presumiblemente sea una tupla no tenga elementos, verificando que el objeto sea una tupla y sea la tupla vacía.

Código 2.20: *La tupla vacía*

```
; (tvacia? t) ↦ booleano?
; t : (listade cualquier)
(define tvacia?
  (λ (t)
    (and (tupla? t) (tvacia? t))))
```

La tupla `t` no tiene elementos si `(long t)` ↦ `0`. Cuando `(long t)` ↦ `1`, se dice que es una **tupla unitaria**.

Código 2.21: *La tupla unitaria*

```
; (tunit? t) ↦ booleano?
; t : (listade cualquier)
(define tunit?
  (λ (t)
    (and (tupla? t) (= (tlong t) 1))))
```

Si S es un conjunto, una **palabra en** S, o **S-palabra** es una tupla que contiene únicamente elementos de S. El término «palabra» es frecuentemente utilizado en la teoría de lenguajes formales, que es uno de los campos de aplicación de los autómatas finitos.

☞ En notación convencional de matemáticas, se suelen utilizar diferentes delimitadores para las tuplas, como (), [], o $\langle\rangle$. Las tuplas suelen utilizarse para crear definiciones que enlistan rasgos de los objetos definidos. Por ejemplo una tupla se utiliza en la definición de un semigrupo como $\langle S, \times \rangle$, donde S es un conjunto y $\times$ es una operación asociativa sobre los elementos de S.

En una tupla no vacía, el primer elemento de la tupla es el que se encuentra en el extremo izquierdo y se le conoce como el «primero de» la tupla, el segundo de la tupla es el elemento que se encuentra a la derecha del primero de la tupla y así sucesivamente. Una tupla de dos elementos es un par.

■ **Ejemplo 2.19** Construya las siguientes tuplas:

1. La tupla vacía $\langle\rangle$.
2. La tupla unitaria $\langle x \rangle$.
3. El par $\langle q,\ 0 \rangle$.
4. La 3-tupla $\langle$'a, 5, "México"$\rangle$.

```
> (tupla) ; Crea una tupla sin argumentos, la tupla vacía.
'()
> (tupla 'x) ; Una tupla unitaria.
'(x)
> (tupla 'q 0) ; Un par.
'(q 0)
> (tupla 'a 5 "México") ; Una 3-tupla.
'(a 5 "México")
>
```

■

2.7 Producto cartesiano

Si A y B son conjuntos no vacíos, el **producto cartesiano** de A con B es el nuevo conjunto formado por todas las tuplas de longitud 2, que tienen como primer elemento un miembro de A y como último elemento, un miembro de B.

☞ La operación de producto cartesiano se denota usualmente por un símbolo $\times$ que en notación infija se escribe entre el nombre de dos conjuntos. La definición en la notación convencional de matemáticas es

$$A \times B = \{(a,b) | a \in A \wedge b \in B\}$$

Podemos crear el concepto pCart como el producto cartesiano. Cuando A↤'() o B↤'(), (pCart A B)↦ '().

Código 2.22: *El producto cartesiano de dos conjuntos*

```
; (pCart A B) ↦ conjuntoDe/par?
; A : conjunto?
; B : conjunto?
(define pCart
  (λ (A B)
    (append* (map (λ (a)
                    (map (λ (b)
                           (append (if (tupla? a) a (tupla a))
                                   (tupla b))) B)) A))))
```

Con `append*` se anexan dos o más listas:

```
(append* lst ... lsts) ↦ lista?
; lst : lista?
; lsts : (listade lista?)
```

■ **Ejemplo 2.20** Calcular el producto cartesiano de `(conj 1 2 3)` con `(conj 'a 'b 'c)` y comparar la cardinalidad del producto potencia con el producto de la cardinalidad de ambos conjuntos.

```
> (pCart (conj 1 2 3) (conj 'a 'b 'c))
'((1 a) (1 b) (1 c) (2 a) (2 b) (2 c) (3 a) (3 b) (3 c)
> (card (pCart (conj 1 2 3) (conj 'a 'b 'c)))
9
> (* (card (conj 1 2 3)) (card (conj 'a 'b 'c)))
9
>
```

■

La cardinalidad del producto cartesiano de dos conjuntos es el producto de la cardinalidad de los conjuntos participantes, esto es: $|A \times B| = |A| \cdot |B|$

2.7.1 El producto cartesiano extendido

El producto cartesiano de conjuntos puede extenderse para calcular el producto cartesiano de una colección indeterminada de conjuntos, el procedimiento permite operar con cero o más conjuntos.

La idea es tomar uno a uno los conjuntos de la lista de conjuntos que se deben operar y crear progresivamente el producto cartesiano. Es necesario tener un conjunto base que es el conjunto unitario que contiene al `vacio`; y con el primer conjunto crear el producto cartesiano, de lo que resulta un conjunto de tuplas unitarias; luego hacer el producto cartesiano del siguiente conjunto con el resultado anterior, de lo que resulta un conjunto de pares, el proceso continúa hasta hacer el producto cartesiano del último conjunto de la lista con el conjunto de tuplas que ha sido resultado del proceso anterior, para formar las tuplas finales.

Código 2.23: *Producto cartesiano extendido de conjuntos*

```
; (pCart* lst ...) ↦ (listade lista?)
; lst : (listade cualquier)
(define pCart*
  (λ LC
    ;--------------------------
    (define pCart*aux
      (λ (LC [res '(())])
        (if (vacio? LC)
            res
            (pCart*aux (cdr LC) (pCart res (car LC))))))
    ;--------------------------
    (if (vacio? LC)
        '()
        (pCart*aux LC))))
```

Observe en el código, que se ha definido la función `pCart*aux` dentro del ámbito de la función `pCart*`, esto tiene el efecto de «esconder» la función `pCart*aux`. El uso de `pCart*aux` será válido únicamente dentro del ámbito de `pCart*`, si se intenta utilizar la función fuera del ámbito de `pCart*` ocasionará un error.

■ **Ejemplo 2.21** Algunos ejemplos en el uso de la función `pCart*` con cero o más conjuntos:

```
> (pCart*)
'()
> (pCart* '())
'()
> (pCart* '() '(w))
'()
> (pCart* '(w))
'((w))
> (pCart* '(w) '(s d))
'((w s) (w d))
> (pCart* '(w) '(s d) '(2 5 7))
'((w s 2) (w s 5) (w s 7) (w d 2) (w d 5) (w d 7))
>
```

■

2.8 Relaciones

Si `A` y `B` son conjuntos, una relación (binaria) `R` de elementos del conjunto `A` con elementos del conjunto `B` es un subconjunto del `(pCart A B)`; es decir, es un subconjunto tuplas ⟨`a b`⟩, donde `(en? a A)`↦ `#t` y `(en? b B)`↦ `#t`. Si una tupla ⟨`a b`⟩ está en la relación, se dice que `a` está relacionado con `b`.

Para expresar que existe una relación llamada `R`, que toma elementos del conjunto `A` y los relaciona con cero, uno o más elementos del conjunto `B`, frecuentemente se utiliza la notación `R:A→B` «esto es un nombre, no una operación».

Cuando es absolutamente claro qué conjuntos son `A` y `B`, es posible referirse a la relación `R: A→B`, simplemente como `R`.

☞ En notación convencional de matemáticas, la relación R entre elementos del conjunto A con elementos del conjunto B, se escribe como

$$R : A \to B = \{(a,b) | a \in A \wedge b \in B\};\ R : A \to B \subseteq A \times B.$$

En términos de `DrRacket`, la relación `R` es simplemente una lista de tuplas, donde el primer elemento de cada tupla pertenece al conjunto `A` y el segundo elemento de cada tupla pertenece a `B`.

Código 2.24: *Verifica la validez de una relación binaria*

```
; (relacion? R A [B A]) ↦ booleano?
; R : (listade tupla?)
; A : conjunto?
; B : conjunto? = A
(define relacion?
  (λ (R A [B A]) ; R : A → B
    (subc? R (pCart A B)))) ; si R ⊆ A × B
```

■ **Ejemplo 2.22** Si `A↤(conj 1 4 6)` y `B↤(conj 'x 'y 'z)` son conjuntos, los siguientes conjuntos de tuplas son relaciones del conjunto `A` al conjunto `B`:

```
> (define A (conj 1 4 6))
> (define B (conj 'x 'y 'z))
> (relacion? '((1 x) (4 x) (4 z) (6 y)) A B)
#t
> (relacion? '((1 x) (4 x) (4 y) (6 y) (6 x) (1 z)) A B)
#t
>
```

La siguiente no es una relación de `A` a `B`.

```
> (relacion? '((x 6) (y 1) (z 1) (z 4) (z 6)) A B)
#f
>
```

■

La relación vacía ocurre cuando ningún elemento de un conjunto está relacionado con elementos de otro conjunto, o bien cuando no hay elementos con los que se puedan crear relaciones, de modo que es un conjunto vacío y se representa también como `'()`.

En este libro se usará del producto cartesiano de dos conjuntos para crear relaciones con un tercer conjunto. Si A_1 y A_2 son conjuntos, una relación $R : A_1 \times A_2 \to B$ es una colección de tuplas de longitud 3 `'((`a_1 a_2 `b) ...+)`, en las que los dos primeros elementos forman un elemento de `(pCart A1 A2)`.

En general, si $A_1 \ldots +$ son conjuntos (vea la página 18 para la notación $\ldots +$), una relación $R : A_1 \times \ldots \rightarrow B$, es un conjunto de tuplas donde los primeros elementos, excepto el último, pertenecen al producto cartesiano extendido (pCart* $A_1 \ldots +$) y el último elemento (el de más a la derecha) es un elemento de B.

Una relación que involucra n conjuntos es una relación n-aria. En el caso de una relación $R : A \rightarrow B$, que involucra solamente dos conjuntos, se llama relación binaria. Sin embargo, una relación n-aria $R : A_1 \times \ldots \rightarrow B$ puede verse como una relación binaria cuando (pCart* A_1 ...) se considera como un solo conjunto, todo es cuestión de conveniencia para los propósitos que se utilizan.

2.8.1 Dominio, codominio y rango

En una relación $R : A \rightarrow B$, el **dominio** de R es el conjunto de elementos de A que se relacionan con al menos un elemento de B.

En relaciones n-arias definidas extensionalmente, la relación es un conjunto de *n*-tuplas definidas explícitamente, esto significa para obtener el dominio de la relación, de cada tupla de elementos, se omite el último componente; consecuentemente quedan tuplas de longitud (- n 1). Como el resultado debe ser un conjunto, es necesario remover los elementos duplicados.

Código 2.25: *Domino de una relación $R : A \rightarrow B$*

```
; (Dom R) ↦ conjunto?
; R : (listade tupla?)
(define Dom
  (λ (R)
    (let ((dom (remove-duplicates
      (map (λ (t) (drop-right t 1)) R))))
        (if (paraTodo (λ (d) (tunit? d)) dom)
            (map (λ (x) (car x)) dom)
            dom))))
```

let es una S-expresión que permite crear ámbitos de nombres para variables, asociando un valor a una literal dentro del ámbito.

```
(let ([id val-expr] ...) cuerpo ...+)
; id : identificador
; val-expr : Expresión que otorga un valor
```

■ **Ejemplo 2.23** Si S1↤'((4 s) (1 s) (2 t) (2 s) (3 r)) es una relación, se calcula el dominio de S1:

```
> (define S1 '((4 s) (1 s) (2 t) (2 s) (3 r)))
> (Dom S1)
'(4 1 2 3)
```

■

`drop-right` toma una lista, deja unos elementos de la derecha de la lista y se queda con los demás. Se especifica la cantidad de elementos que son dejados:

```
(drop-right lst pos) ↦ lista?
; lst : (listade cualquier)
; pos : numero-entero-no-negativo?
```

`remove-duplicates` toma una lista y devuelve otra lista sin elementos duplicados. En su forma más simple tiene la sintaxis:

```
(remove-duplicates lst) ↦  lista?
; lst : (listade cualquier)
```

El **codominio** de la relación es el conjunto de elementos que aportan elementos para que algunas tuplas puedan relacionarse; si $R: A_1 \times A_2 \times \cdots \times A_n \rightarrow B$ es una relación, el conjunto B es el codominio.

El **rango** es un subconjunto del codominio de la relación. El rango contiene únicamente a los elementos que tienen algún elemento del dominio con el que han sido relacionados.

De una relación $R : A \rightarrow B$, se obtienen las siguientes definiciones:
El dominio[Gin68]:$\text{Dom}(R) = \{a \in A | \exists b \in B : (a,b) \in R\}$
El codominio: $\text{Cod}(R) = B$
El rango [Gin68]:$\text{Ran}(R) = \{b \in B | \exists a \in A : (a,b) \in R\}$

Código 2.26: *Rango de una relación* $R : A \rightarrow B$

```
1 ; (Ran R) ↦ conjunto?
2 ; R : (listade tupla?)
3 (define Ran
4   (λ (R)
5     (remove-duplicates
6        (map (λ (t) (car (take-right t 1))) R))))
```

`take-right` de una lista toma desde el extremo de la derecha un número determinado de elementos y los devuelve como una lista, el resto de elementos son omitidos. La sintaxis es:

```
(take-right lst pos) ↦  lista?
; lst : (listade cualquier)
; pos : numero-entero-no-negativo?
```

■ **Ejemplo 2.24** El conjunto de tuplas `S2` ↦ `'((4 M s) (1 F s) (2 F t) (2 F s) (3 M r))` puede verse como una relación `S2:'(1 2 3 4)×'(F M)→'(r s t)`, se calcula el rango de `S2`:

```
> (define S2 '((4 M s) (1 F s) (2 F t) (2 F s) (3 M r)))
> (Ran S2 )
'(s t r)
```

■

2.8.2 Imagen de un elemento del dominio

Si `R:A→B` es una relación y `a` es un elemento del dominio, la **imagen** de `a` bajo la relación `R`, es el subconjunto de elementos `b` del rango `(Ran R)` tales que `(en? (tupla a b) R)` $\mapsto$ `#t`.

☞ En la notación convencional, si $R : A \to B$ y $a \in A$, la imagen de a bajo la relación R se define como

$$R(a) = \{b \in B | (a,b) \in R\}$$

Código 2.27: *Imagen de un elemento del dominio de una relación*

```
; (Im a R) ↦ conjunto?
; a : cualquier
; R : conjuntoDe/tupla?
(define Im
  (λ (a R)
    (let ((eDom (if (list? a) a (list a))))
      (filter (λ (c) (en? (append eDom (list c)) R))
              (Ran R)))))
```

```
> (define R '((2 3 8 0) (2 4 8 1) (1 6 9 1) (2 3 9 0) (2 3 8 1) (2 4 9 1)))
> (Im '(2 3 8) R) ; R((2,3,8)) = {0,1}
'(0 1)
> (define S '((2 6) (3 6) (2 8) (1 4) (1 1) (4 3)))
> (Im 4 S) ; S(4) = {3}
'(3)
>
```

El **argumento** de una relación, es aquel elemento del dominio de la relación, del cual se desea saber su imagen. El argumento de la relación puede ser también una tupla, esto cuando la relación está definida con el producto cartesiano de conjuntos.

Se puede extender la idea de la imagen para poder aplicarla a un subconjunto de elementos del dominio. La **imagen extendida** toma como argumento 0 o más elementos del dominio. La imagen de un subconjunto del dominio se obtiene al aplicar la unión extendida a todas las imágenes obtenidas al calcular la imagen de cada elemento del subconjunto solicitado.

☞ En la notación convencional, si $R : A \to B$ y $C \subseteq A$, la imagen extendida se refiere a $R[C]$ que es la imagen del conjunto C bajo la relación R y se define como

$$R[C] = \bigcup_{c \in C} R(c) = \{b \in B | \exists c \in C : b \in R(c)\}$$

Tanto la imagen de un elemento del dominio, como la imagen extendida de un subconjunto de elementos del dominio, generan un conjunto de elementos en el rango de la relación.

Código 2.28: *Imagen de un subconjunto del dominio*

```
; (Im* SD R) ↦   conjunto?
; SD : conjunto?
; R : (listade tupla?)
(define Im*
  (λ (SD R) ; SD subconjunto del dominio
    (apply union* (map (λ (e) (Im e R)) SD))))
```

■ **Ejemplo 2.25** Si S ↦ '((2 3 8 0) (2 4 8 1) (1 6 9 2) (2 4 9 1) (2 3 9 0) (2 3 8 1)) es una relación, la imagen extendida del conjunto '((2 3 8) (1 6 9)) es '(0 1 2), porque la imagen de '(2 3 8) es '(0 1); mientras que la imagen de '(1 6 9) es '(2) y (union* '(0 1) '(2)) ↦ '(0 1 2):

```
> (define S '((2 3 8 0) (2 4 8 1) (1 6 9 2) (2 4 9 1) (2 3 9 0) (2 3 8 1)))
> (Im* '((2 3 8) (1 6 9)) S)
'(0 1 2)
```

■

2.9 Funciones

Una relación F:A⟶B es una **función** cuando cada elemento del dominio de la relación, está asociado con exactamente un elemento del rango. En otras palabras, la imagen de cada elemento del dominio es un conjunto unitario.

Las funciones son especialmente importantes en computación, porque se elimina el indeterminismo al tratar de conocer qué elemento del dominio es relacionado con qué elemento del rango y si la función es vista como una transformación, cada elemento del dominio produce un único resultado.

☞ En la notación convencional, una relación $f : A \to B$ es una función cuando

$$\forall a \in A : [\exists! b \in B : \langle a, b \rangle \in f].$$

Código 2.29: *Determina si una relación es una función*

```
; (funcion? lsts) ↦ booleano?
; lsts : (listade lista?)
(define funcion?
  (λ (R)
    (let ((imgns (map (λ (e) (Im e R)) (Dom R))))
      (paraTodo (λ (C) (unitario? C)) imgns))))
```

■ **Ejemplo 2.26** Si R ↦ '((2 3 8 0) (2 4 8 1) (1 6 9 1) (2 4 9 1) (2 3 9 0) (2 3 8 1)) y Fa ↦ '((2 3 8 0) (2 4 8 1) (1 6 9 1) (2 4 9 1) (2 3 9 0)) son relaciones, determine cuál de ellas es una función.

```
> (define R '((2 3 8 0) (2 4 8 1) (1 6 9 1) (2 4 9 1) (2 3 9 0) (2 3 8 1)))
> (define Fa '((2 3 8 0) (2 4 8 1) (1 6 9 1) (2 4 9 1) (2 3 9 0)))
> (funcion? R)
#f ; R(⟨2,3,8⟩) = {0,1}
> (funcion? Fa)
#t
```

■

Para calcular la imagen de un elemento del dominio de una función, es posible reutilizar el código para la imágen de un elemento del dominio de una relación (página 55) y en el mismo sentido, la imagen extendida de un subconjunto de elementos del dominio de una relación (página 56).

2.9.1 Función binaria

Si A, B y C son conjuntos, una **función binaria** se puede representar como F:A×B→C y es un subconjunto de (pCart* A B C), es decir, es una colección de 3-tuplas de la forma '(a b c), donde (en? a A) ↦ #t, (en? b B) ↦ #t y (en? c C) ↦ #t.

En general, si A ...+ y C son conjuntos, una función n-aria , es un subconjunto de (pCart* A ...+ C). En una función F: $A_1 \times A_2 \times \cdots \times A_n \to C$, el número n de conjuntos que aportan elementos para asociarlos con algún elemento de C, es la **aridad** de la función.

■ **Ejemplo 2.27** La función G que contiene las tuplas '((2 3 8 0) (2 4 8 1) (1 6 9 1) (2 4 9 1) (2 3 9 0)) es una función de aridad 3. Los conjuntos que aportan elementos para asignar son $A_1 \mapsto$ '(2 1), $A_2 \mapsto$ '(3 4 6) $A_3 \mapsto$ '(8 9); y el codominio de la función es C ↦ '(0 1).

(Dom G)	(Im _ G)
'(2 3 8)	'(0)
'(2 4 8)	'(1)
'(1 6 9)	'(1)
'(2 4 9)	'(1)
'(2 3 9)	'(0)

El símbolo /_ / significa que debe ser reemplazado por algún elemento del dominio, como en (Im '(2 3 8) G).

```
> (define G '((2 3 8 0) (2 4 8 1) (1 6 9 1) (2 4 9 1) (2 3 9 0)))
> (Dom G)
'((2 3 8) (2 4 8) (1 6 9) (2 4 9) (2 3 9))
> (Im '(2 3 8) G)
'(0)
> (Im '(2 4 9) G)
'(1)
```

■

2.9.2 Evaluación de una función

Sabiendo que una función F:A→B, tiene imágenes unitarias para cada elemento a de su dominio, se define ahora el procedimiento evalf, que obtiene el único elemento que pertenece a la (Im e F).

Así cuando (en? e (Dom F)) ↦ #t y (funcion? F) ↦ #t, la (Im e F) se refiere a un conjunto unitario (F es una función); mientras que (evalf e F), que es la evaluación del elemento e bajo la función F se refiere a un elemento del rango de la función.

Código 2.30: *Evaluación de un elemento bajo una función*

```
; (evalf e F) ↦ cualquier
; e : cualquier
; F : (listade tupla?)
(define evalf
  (λ (e F)
    (let ((ev (Im e F)))
      (if (vacio? ev)
          (error "Error en los argumentos de la funcion")
          (car (Im e F))))))
```

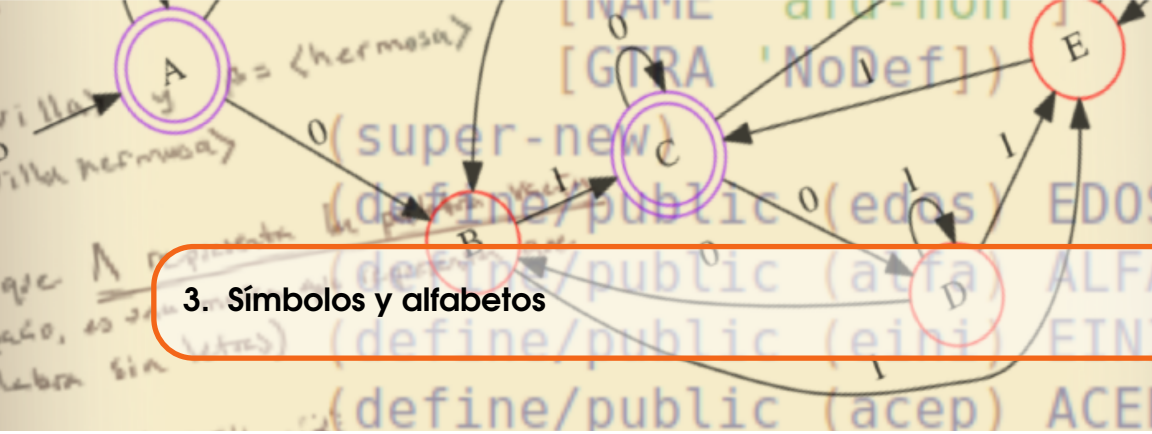
Si (evalf a F) ↦ b, a es el argumento y el elemento b del rango de la función es el valor de la función aplicada al elemento a del dominio.

■ **Ejemplo 2.28**

```
> (define G '((2 3 8 0) (2 4 8 1) (1 6 9 1) (2 4 9 1) (2 3 9 0)))
> (evalf '(1 6 9) G)
1
>
```

■

II De símbolos a lenguajes



3. Símbolos y alfabetos

3.1 Símbolos

Los símbolos son representaciones perceptibles de conceptos, que nos permiten transmitir y comunicar ideas o pensamientos. Hay símbolos que se hacen con las manos; con figuras humanas en ciertas posiciones, por ejemplo cuando se coloca un letrero con una imagen de una persona andando, es un símbolo que significa «Es permitido andar por aquí».

Los símbolos con los que se trabaja en este libro, son los caracteres imprimibles del teclado, como los símbolos del código `ASCII`[1] como las letras `/a/, /b/, /c/, ..., /A/, /B/, /C/,... /$/, /%/,..`, o los símbolos que convencionalmente se escriben en textos de matemáticas, como $/\alpha/$, $/\beta/$, $/\gamma/$. En ocasiones podemos crear nuestros propios símbolos como $\mapsto$, que en este texto se ha especificado que se debe leer como «produce» (página 19).

Frecuentemente podemos llamar a los símbolos, **letras** o **señales**. Este libro se trata de cómo construir máquinas computacionales que reaccionen en presencia de símbolos específicos.

[1] `ASCII`: *American Standard Code for Information Interchange*, código estandar creado a principios de la década de 1960 para ser utilizado en dispositivos electrónicos con el fin de intercambiar datos.

3.1.1 Símbolos en DrRacket

Aunque ya se han utilizado anteriormente los símbolos, en esta sección daremos un enfoque orientado a la construcción de palabras y lenguajes. En DrRacket los símbolos se escriben de dos formas. Con un apóstrofo que antecede al símbolo, como 'a, 'A, 'estoEsUnSimbolo, o incluso '12*+=. Los símbolos expresados de esta forma se llaman **símbolos atómicos**, o simplemente **átomos**. En DrRacket cada símbolo debe tener un significado. El significado de un símbolo atómico es él mismo. Se escriben con un apóstrofo porque de otro modo el símbolo debe tener un significado diferente.

```
> 'estoEsUnSimbolo
'estoEsUnSimbolo
> 'a
'a
> '12*+=
'12*+=
> '(2 3 4)
'(2 3 4)
> pi ; constante predefinida.
3.141592653589793
>
```

La otra forma de escribir símbolos en DrRacket es omitiendo el apóstrofo, como en a, A, raizCuadrada. Sin embargo, en el uso del lenguaje DrRacket se exige que esta clase de símbolos esten asociados con un significado previamente establecido, ya sea en forma de primitiva del lenguaje, o definido arbitrariamente por el programador mediante una forma especial define, let o alguna variante de ésta.

Cuando el símbolo se escribe sin apóstrofo y no tiene alguna definición asociada, entonces al utilizarlo se observa un mensaje de error.

```
> a
a: undefined;
cannot reference an identifier before its definition
> (define a 15)
> a
15
>
```

Esencialmente el mensaje de error dice que el símbolo a no ha sido definido y que no se puede hacer referencia a un identificador (se refiere a la a) antes de su definición.

Esto sucede por alguna de las siguientes razones:

1. Se escribió el identificador a, en lugar del símbolo 'a.
2. No se hizo una definición del concepto a. Esto se resuelve al definir el significado para el símbolo desconocido (define a ...) antes de utilizar el símbolo a, que ahora tendrá un significado específico.

3. En realidad no se quería escribir `a`, sino otro, posiblemente `A`. Aquí se supone que el usuario cometió un error al escribir un símbolo que no esta definido por otro que sí lo está.

En `DrRacket` hay que tomar en cuenta que los símbolos de las letras minúsculas son diferentes a los símbolos de las letras mayúsculas y pueden significar cosas diferentes:

```
> (define a 10)
> (define A 20)
> (+ a A)
30
>
```

Un símbolo especial que se requiere y es necesario introducir, es un símbolo para denotar la *nulidad de símbolos*, lo representaremos con la letra `'`ε y representa la no existencia de símbolos del alfabeto. Este símbolo `'`ε se requiere como base para la construcción y análisis de todas las palabras.

Código 3.1: *El símbolo nulo*

```
1 ; se define nulo como el símbolo 'ϵ
2 (define nulo 'ϵ)
```

■ Ejemplo 3.1

```
> nulo
'ϵ
>
```

■

Para determinar si algún símbolo es precisamente el símbolo nulo, esto se logra comparándolo con `nulo` con el siguiente predicado.

Código 3.2: *Se verifica si el argumento es el símbolo nulo*

```
1 ; (nulo? s) ↦ booleano?
2 ; s : cualquiera?
3 (define nulo?
4   (λ (s)
5     (equal? s nulo)))
```

■ Ejemplo 3.2

```
> (nulo? "nada")
#f
> (nulo? nulo)
#t
>
```

■

Otros símbolos son los números, las palabras encerradas entre comillas inglesas y las cadenas de caracteres que empiezan con un apóstrofo. Será útil tener un predicado que determine si algo es un símbolo o no lo es. El siguiente predicado hace el trabajo.

Código 3.3: *Predicado para determinar si es un símbolo*

```
1 ; (simbolo? s) ↦ boolean?
2 ; s : cualquiera?
3 (define simbolo?
4   (λ (s)
5     (or (number? s) (symbol? s) (string? s) (boolean? s))))
```

■ **Ejemplo 3.3** Se comprueban si los elementos de un conjunto son símbolos y se muestra que ni las listas, conjuntos o vectores, se considerarán como símbolos.

```
> (paraTodo (λ (s) (simbolo? s)) (conj 1 'x "hola mundo" #t))
#t
> (simbolo? '(2 3 4)) ; Una lista no es un símbolo.
#f
> (simbolo? #(2 3 4)) ; Un vector no es un símbolo.
#f
>
```

■

3.2 Alfabetos

Un **alfabeto** es una colección finita de símbolos. Se pueden utilizar los símbolos imprimibles por un teclado convencional como los teclados QWERTY[2] para elegir los símbolos contenidos en los alfabetos y para designar nombres para alfabetos en particular.

En teoría de lenguajes formales, generalmente se utiliza el símbolo /Σ/ para representar un alfabeto. En este documento se prefiere el identificador `S` simplemente porque se logra con menos esfuerzo.

■ **Ejemplo 3.4** Se define un alfabeto con los símbolos `0` y `1`, nombrado `S1`:

```
> (define S1 (conj 0 1))
> S1
'(0 1)
>
```

■

[2]QWERTY designa los teclados con la distribución de teclas creada por Christopher Sholes en 1968 y que posteriormente se distribuyó en las máquinas de escribir de la marca Remington. `http://www.ideafinder.com/history/inventions/qwerty.htm`

En escencia, un alfabeto tiene la misma forma que un conjunto, tal y como se ha utilizado hasta ahora [para la notación y concepto de conjuntos, vaya a la página 31]. La diferencia radica en la interpretación de los objetos que se han creado. Un conjunto no es un alfabeto si alguno de sus elementos no es un símbolo de los símbolos permitidos.

Código 3.4: *Crear un alfabeto*

```
; (alf S ...) ↦ (or (listade simbolo?) #f)
; S ... : cualquiera
(define alf
  (λ S
    (if (paraTodo (λ (s) (simbolo? s)) S) S \#f)))
```

■ **Ejemplo 3.5** Crear los siguientes alfabetos:

1. $\Sigma_1 = \{0,1\}$.
2. $\Sigma_2 = \{a,b,c\}$.

```
> (define S1 (alf 0 1))
> S1
'(0 1)
> (define S2 (alf 'a 'b 'c))
> S2
'(a b c)
>
```

■

El alfabeto más pequeño que existe es el alfabeto que no tiene símbolos. Este alfabeto se conoce como **alfabeto vacío**, escrito como un conjunto vacío `'()`. En la teoría de lenguajes formales se ha utilizado el símbolo `'`ε para representar la falta de símbolos, es decir, para enfatizar en que no hay símbolo alguno, se utiliza `'`ε. En la práctica, el alfabeto `'(`ϵ`)` es el mismo que el alfabeto vacío, aunque lo correcto es utilizar `'()` para denotar el alfabeto vacío.

Aunque el espacio en blanco no es visible, es diferente que `'`ε y tiene propósitos especiales; cuando se utilice, debe ser considerado como parte del alfabeto y puede ser denotado como /␣/.

Código 3.5: *El alfabeto vacío*

```
;; El alfabeto vacío es el conjunto
;; que no contiene símbolo alguno.
(define alfaVacio '())
```

Quizá por ahora, el concepto de *alfabeto vacío* puede parecer inútil, sin embargo es importante como punto de referencia. Cuando se desea hacer procedimientos recursivos, la verificación del alfabeto vacío mediante `alfaVacio?` es una manera efectiva para detener el proceso.

Código 3.6: *Determinar si un alfabeto es vacío*

```
; (alfaVacio? S) ↦ boolean?
; S : lista?
(define alfaVacio? empty?)
```

■ **Ejemplo 3.6** Ejemplos para determinar la condición de vacuidad de un alfabeto.

```
> (alfaVacio? alfaVacio)
#t ; la constante que se ha definido antes
> (alfaVacio? '())
#t
> (alfaVacio? empty?)
#t
> (alfaVacio? (alf))
#t
> (alfaVacio? '(0 1 2))
#f
>
```

■

3.2.1 Verificar un alfabeto

Como se ha definido antes (página 64), un alfabeto es un conjunto finito de símbolos, un alfabeto contiene una cantidad finita de símbolos que son diferentes entre ellos, incluso cabe la posibilidad de que no haya símbolo alguno.

Así por el momento se considerarán cuatro restricciones importantes [pero fáciles de cumplir] para que un conjunto sea considerado un alfabeto:

1. **El conjunto vacío `'()` es un alfabeto**. El conjunto vacío no tiene símbolos, pero aún así es un alfabeto, se llama `alfaVacio`.
2. **El conjunto de símbolos es definido explícitamente.** Para verificar este punto, la definición del alfabeto debe ser una lista con cada uno de los símbolos permitidos. De modo que el hecho de que sea una lista, es suficiente para entender que sus elementos se han descrito de manera extensional, ya que las listas pueden ser vacías (ver punto anterior) o bien tener un primer elemento y el resto de ellos.
3. **El conjunto de símbolos es finito.** Los alfabetos tienen en general pocos símbolos, pero pueden tener muchos, siempre que sea un número finito. En `DrRacket` se puede verificar la finitud verificando que el número es estrictamente menor que `+inf.0` que es un símbolo que representa al infinito positivo ($+\infty$, en matemáticas).
4. **Todos los miembros del alfabeto son símbolos.** Aunque luego puede incluirse el caso en que los alfabetos incluyan símbolos que significan procedimientos, por ahora se considerará solamente los símbolos atómicos, los numericos, las cadenas de caracteres, los símbolos booleanos y en general aquellos que se puedan verificar mediante el procedimiento `simbolo?`

Código 3.7: *Verificación de un alfabeto*

```
; (alf? S) ↦ boolean?
; S : (listade cualquier)
(define alf?
  (λ (S)
    (o (alfaVacio? S) ; Es el alfabeto vacío o es
       (and (list? S) ; ... una lista
            (< (card S) +inf.0) ; ... finita
  (paraTodo (λ (s) (simbolo? s)) S) ; ... de símbolos.
            ))))
```

En un sentido más eficiente, la verificación de `alf?` se logra únicamente verificando que el argumento sea una lista, ya que en `DrRacket` una lista puede ser vacía, o de cardinalidad finita; en cuyo caso incluye símbolos que pueden ser atómicos o no. Sin embargo la definición anterior se hace con el propósito de hacer explícitas las restricciones citadas.

■ **Ejemplo 3.7** Las siguientes interacciones tienen como propósito verificar cada uno de los casos expuestos en la lista de restricciones para los alfabetos:

- El alfabeto vacío

```
> (alf? alfaVacio)
#t
> (alf? '())
#t
>
```

- El alfabeto como una lista finita de símbolos

```
> (alf? '(0 1)) ; Es una lista explícita.
#t
> (alf? '(a b c))
#t
>
```

- Conjuntos de cardinalidad finita

```
> (alf? '(0 1 3 4 5)) ; Cardinalidad finita.
#t
>
```

- Conjuntos de símbolos

```
 > (alf? (list #t #f)) ; Lista de constantes.
#t
 > (alf? (list "hola" "mundo"))
#t
>
```

■

3.2.2 Pertenencia de un símbolo a un alfabeto

Decimos que un símbolo `s` *es de* un alfabeto `S`, si es el símbolo nulo, o bien pertenece al alfabeto. La palabra *pertenece* debe evocar sin confusión alguna al procedimiento `en?` de conjuntos [página 35], aquí se define una versión modificada, verificando explícitamente el símbolo nulo.

Código 3.8: *Pertenencia de un símbolo a un alfabeto*

```
; (enS? s S) ↦ booleano?
; s : simbolo?
; S : alf?
(define enS?
  (λ (s S)
    (o (equal? s nulo)
       (en? s S))))
```

■ **Ejemplo 3.8** El comportamiento del procedimiento `enS?` se utiliza para verificar la pertenencia de un símbolo a un alfabeto:

```
> (enS? 'x  (conj a w i g x))
#t
> (define S '(0 1)) ; S tiene los símbolos '(0 1)
> (enS? 2 S)
#f
>
```

■

3.2.3 Cardinalidad de un alfabeto

La cardinalidad de un alfabeto es la cantidad de símbolos que contiene, es similar a la cardinalidad en los conjuntos (página 36) por lo que no hay necesidad de crear una nueva definición para calcular la cardinalidad de un alfabeto.

La cardinalidad de un alfabeto es un número entero no negativo, que es cero en el único caso cuando el alfabeto es precisamente `alfaVacio`.

Saber el número de símbolos de un alfabeto sirve para responder preguntas como «¿cuántos símbolos tiene el alfabeto S?» o «cuántos símbolos tienen en común los alfabetos S_1 y S_2».

También se utiliza como un criterio de orden, ya que se puede determinar que un alfabeto S_1 ocurre antes que otro alfabeto S_2, si la cardinalidad de S_1 es menor a la de S_2.

```
; (<alf? S1 S2) ↦ booleano?
; S1 : alf?
; S2 : alf?
(define <alf?
  (λ (S1 S2)
    (< (card S1) (card S2))))
```

■ **Ejemplo 3.9** Determinar si el alfabeto $\Sigma_1 \leftarrow\!\shortmid \{q,k,s,u,e,l,a,p\}$ es menor (en términos de su cardinalidad) que $\Sigma_2 \leftarrow\!\shortmid \{4,6,9\}$.

```
> (define S1  (alf 'q 'k 's 'u 'e 'l 'a 'p))
> (card S1)
8
> (define S2  (alf 4 6 9))
> (card S2)
3
> (<alf? S1 S2)
#f
>
```

■

3.2.4 Subalfabeto

Si S_1 y S_2 son alfabetos, decimos que S_1 es un subalfabeto de S_2 cuando todos los símbolos de S_1 pertenecen al alfabeto S_1.

La definición efectiva de subconjunto con el nombre de `subc?` es un predicado que se estudió en la página 37, se puede emplear este predicado para crear otro similar, pero con un nombre más adecuado, que se ajuste al tema de estudio actual:

Código 3.9: *Subalfabeto*

```
; (subalf? S1 S2) ↦ booleano?
; S1 : alf?
; S2 : alf?
(define subalf?
  (λ (S1 S2)
    (paraTodo (λ (s) (enS? s S2)) S1)))
```

■ **Ejemplo 3.10** Determinar si un alfabeto es subalfabeto de otro en los siguientes casos:

1. $\{a,b\} \subseteq \{c,d,b,a\}$
2. $\{c,d,b,a\} \subseteq \{a,b\}$
3. $\{\} \subseteq \{a,b\}$
4. $\{c,d,b,a\} \subseteq \{c,d,b,a\}$

```
> (subalf? '(a b) '(c d b a))
#t
> (subalf? '(c d b a) '(a b))
#f
> (subalf? alfaVacio '(a b))
#t
> (subalf? '(c d b a) '(c d b a))
#t
>
```

■

3.2.5 Igualdad en alfabetos

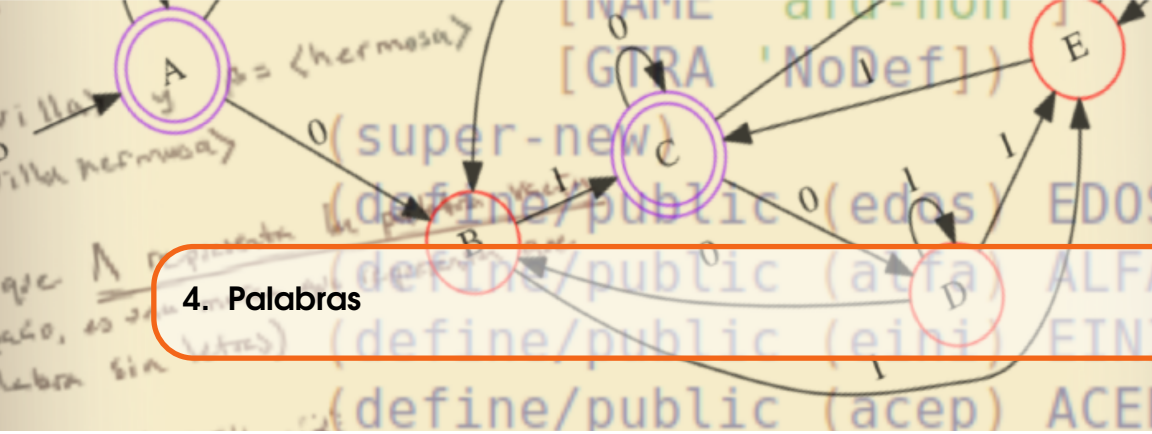
Dos alfabetos S_1 y S_2 son iguales si contienen exactamente los mismos símbolos.

Código 3.10: *Verificar igualdad en alfabetos*

```
; (S=? S1 S2) ↦ booleano?
; S1 : alf?
; S2 : alf?
(define S=?
  (λ (S1 S2)
    (y (subalf? S1 S2) (subalf? S2 S1))))
```

```
> (S=? (alf 2 8 3 4) (alf 3 8 4 2))
#t
> (S=? '() alfaVacio) ;; requiere la def de alfaVacio
#t
> (S=? (alf 1 2 3) (alf 0 2 3 1))
#f
```

Una vez que se tienen los conceptos básicos, podemos construir nuevos conceptos más complejos, que incluyan conceptos más simples, con el fin de crear herramientas más potentes manipular objetos que utilicen símbolos.



4. Palabras

Para que la utilidad de los símbolos sea notable es necesario que se interprete el significado que tengan. Los símbolos primitivos se autosignifican, pero al juntar dos o más símbolos, entonces es posible asignar e interpretar nuevos significados. En este capítulo se defininen maneras para combinar símbolos; y maneras de generar y hacer operaciones con palabras.

4.1 Creación de palabras

Una **palabra** es una secuencia finita de símbolos. Una secuencia conceptualmente es lo mismo que una tupla, cadena, lista o *string*. En este libro se prefiere el término **palabra** por estar más cercano al término «lenguaje». Se define el procedimiento `pal` crea palabras con una lista no determinada de símbolos. Si se utiliza el procedimiento `pal` sin símbolos, entonces se genera una palabra vacía.

Código 4.1: *Creación de palabras*

```
;(pal S ...) ↦ pal?
; S : simbolo
(define pal
  (λ S
    (if (paraTodo (λ (s) (simbolo? s)) S) S #f)))
```

■ **Ejemplo 4.1** Los siguientes ejemplos son palabras

```
> (pal 1 0 1 1 0 0)
'(1 0 1 1 0 0)
> (pal)
'()
>
```

■

Tal como se presentan, las palabras se parecen mucho a los conjuntos o a los alfabetos [página 64], porque se escriben entre paréntesis y con un apóstrofo que les precede.

■ **Ejemplo 4.2** No hay diferencia en la definición de alfabetos y palabras, salvo por la intención con la que se hace.

```
> (define S (alf 0 1)) ; Define un alfabeto.
'(0 1)
> (define s (pal 0 1)) ; Define una palabra.
'(0 1)
>
```

■

Se verifica que un objeto sea una palabra al determinar que el objeto sea una lista y cada elemento de la lista sea un símbolo.

Código 4.2: *Determina si un objeto es una palabra*

```
1 ;(pal? w) ↦ booleano?
2 ; w : cualquier
3 (define pal?
4   (λ (w)
5     (if (y (list? w)
6            (paraTodo (λ (s) (simbolo? s)) w)))))
```

La definición matemática de una palabra se podría expresar como:

$$pal(w) \leftrightarrow lista(w) \wedge \forall s \in w : simbolo(w).$$

Donde *lista* es una función con cualquier dominio y rango en los booleanos, y determina `#t` si el parámetro *w* es una lista; por su parte *simbolo* es una función con dominio en las palabras y rango en los booleanos, y determina `#t` si el objeto *w* es un símbolo permitido.

Se define enseguida el predicado `penS?` (leído como «pe-en-ese»), es un predicado que determina si el argumento de la función es una palabra hecha con los símbolos de determinado alfabeto `S`, se puede crear con el nombre `penS?`, devuelve `#t` cuando se le presenta un objeto que sea una palabra y que todos los símbolos con los que se conforma son del mismo alfabeto; debe responder con `#f` en cualquier otro caso.

El predicado `penS?` recibe dos argumentos: una palabra y un alfabeto, de hecho son dos listas. La interpretación debe ser cuidadosa para no cometer errores que conlleven a conclusiones erróneas.

Código 4.3: *Detecta si una palabra procede de un alfabeto*

```
;(penS? w S) ↦ booleano?
; w : pal?
; S : alf?
(define penS?
  (λ (w [S 'null])
    (if (equal? S 'null)
        (list? w)
        (y (list? w) ; w es una lista
           (paraTodo (λ (s) (enS? s S)) w)))))
```

■ **Ejemplo 4.3** El primer argumento en `penS?` es palabra y el segundo un conjunto.

```
> (penS? (pal 1 1 0)  (conj 0 1))
#t
> (penS?  (pal 1 2 0 0 2)  (conj 0 1))
#f
> (penS?  (pal 1 2 0 0 2)  (conj 0 1 2))
#t
>
```

■

4.2 Longitud de una palabra

La **longitud** de una palabra es la cantidad de símbolos que contiene.

☞ En la notación convencional, para denotar la longitud de una palabra w, se utiliza el mismo símbolo que la cardinalidad de un conjunto. $|w| = n$ significa que la longitud de la palabra w es de n símbolos, donde n es un número entero no negativo.

La longitud de una palabra `w` se expresará como `(long w)` $\mapsto$ `k`, donde `k` es un número entero no negativo. El procedimiento `long` no modifica el resultado en presencia de `'`ε.

Código 4.4: *Longitud de una palabra*

```
; (long w [res]) ↦ numero
; w : pal?
; res : numero? = 0 -- parametro opcional.
(define long
  (λ (w [res 0])
    (cond ((vacio? w) res)
          ((equal? (car w) nulo) (long (cdr w) res))
          (else (long (cdr w) (+ res 1))))))
```

■ **Ejemplo 4.4** Los símbolos pueden tener una o más letras.

```
> (long '(c a m i s a))
6
> (long '(camisa de cuadros))
3
>
```

■

4.2.1 La palabra vacía

La **palabra vacía** es una palabra que no tiene símbolos. Una palabra sin letras es un concepto abstracto, sin embargo es útil al hacer operaciones con palabras que requieren analizar las letras una a una. Un modo de saber cuándo se ha terminado de analizar las letras, es sabiendo cuándo la palabra es vacía.

Código 4.5: *La palabra vacía*

```
;; La palabra vacía es una palabra sin símbolos.
(define pVacia '())
```

Como $'\varepsilon$ no modifica en nada a la palabra, tampoco aporta para la longitud, de modo que $'(\varepsilon \ldots \varepsilon)$ es la única palabra que tiene longitud 0.

A su vez, será útil definir un predicado que determine `#t` si el objeto cuestionado es igual que la palabra vacía; y que determine `#f` en cualquier otro caso.

Código 4.6: *Verificación de palabra vacía*

```
; (pVacia? w) ↦ booleano
; w : cualquiera
(define pVacia?
  (λ (w)
    (o (equal? w pVacia) (= (long w) 0))))
```

■ **Ejemplo 4.5** Las siguientes interacciones muestran cómo se utiliza el predicado cuando se le proporciona la palabra vacía y cualquier otra palabra.

```
> (pVacia? '())
#t
> (pVacia? pVacia)
#t
> (pVacia? '(0 1 0 0 1))
#f
> (pVacia? '(ε ε))
#t
>
```

■

La palabra vacía es palabra de cualquier alfabeto. Esto es cierto porque la palabra vacía no tiene letras, de modo que el conjunto de letras con las que se forma la palabra vacía es el conjunto vacío y el conjunto vacío es subconjunto de cualquier conjunto.

```
> (penS? pVacia '(2 3 4))
#t
> (penS? pVacia '(a b c d))
#t
```

4.2.2 Palabras unitarias

Si `w` es una palabra con `(long w)` $\mapsto$ `1`, decimos que `w` es una **palabra unitaria**.

Es posible definir un procedimiento que genere palabras unitarias, proporcionando un símbolo `s` de cualquier alfabeto. El procedimiento debe devolver una palabra de longitud `1` que contenga a `s` como único símbolo, exceptuando el caso en que se de `'`ε como argumento.

Código 4.7: *Generador de palabras unitarias*

```
; (punit s) ↦ pal?
; s : simbolo?
(define punit
  (λ (s)
    (if (y (simbolo? s) (neg (equal? s nulo)))
      (tupla s)
      'ERR-noSimbolo-o-nulo)))
```

■ **Ejemplo 4.6** Con el alfabeto S↤`'(w x y z)`, genere las `4` palabras unitarias en S.

```
> (define S (conj w x y z))
> (map (λ (s) (punit s)) S)
'((w) (x) (y) (z))
>
```

■

4.3 El alfabeto generador de una palabra

El **alfabeto generador** de una palabra es el conjunto de símbolos necesarios para construir la palabra. Para obtener el alfabeto generador de una palabra simplemente hay que coleccionar las letras de la palabra que no se repitan.

La definición efectiva requiere una palabra `w` y un repositorio `res` con los símbolos de `w` no repetidos. Al analizar una a una las letras de la palabra, si el símbolo leído se encuentra ya en el conjunto temporal `res`, significa que fue considerado anteriormente y se debe recursar el procedimiento `Sgen`, sin

modificar el alfabeto que se está construyendo; por otro lado, cuando no pertenece, entonces sí de debe agregar el primer elemento de `w` al alfabeto generador `res`.

Código 4.8: *El alfabeto generador de una palabra*

```
; (Sgen w [res '()]) ↦ alfabeto
; w : palabra
; res : alf? = '()
(define Sgen
  (λ (w [res '()])
    (cond ((pVacia? w) res)
          ((en? (car w) res) (Sgen (cdr w) res))
          (else (Sgen (cdr w) (agregar (car w) res #:chk #f))))))
```

■ **Ejemplo 4.7** Generando alfabetos. ¿Cuál es el alfabeto generador de la palabra vacía? ¿Cuál es el alfabeto de la palabra `'(c a m i o n e t a)`?

```
> (Sgen '())
'()
> (Sgen '(c a m i o n e t a))
'(t e n o i m a c)
>
```

■

El alfabeto generador de una palabra `w`, es el alfabeto de menor cardinalidad que puede generar la palabra `w`. De lo anterior se desprenden tres razonamientos:

1. Cualquier subalfabeto propio del alfabeto generador de una palabra `w`, es incapaz de proporcionar todos los símbolos que requiere la palabra `w`.
2. `w` es palabra de cualquier superalfabeto del alfabeto generador de `w`.
3. Si `v` y `w` son palabras y `V` el alfabeto generador de `v` y `W` el alfabeto generador de `w`, `(<= (card (union V W)) (+ (long v) (long w)))`↦ `#t`..

4.4 Igualdad entre palabras

Dos palabras `v` y `w` son iguales, si tienen exactamente los mismos símbolos en las mismas posiciones. A diferencia de los conjuntos, en las palabras sí importa el orden y se permiten las repeticiones de los símbolos.

■ **Ejemplo 4.8**

- `'(1 3 4 2)` es diferente que `'(4 3 1 2)` aunque tienen los mismos símbolos, no están en el mismo orden.
- `'(a b b b a a)` es igual que `'(a b b b a a)`.
- `'(1 1 1 1)` es una palabra válida, aunque se repitan los símbolos; cualquier reacomodo de los símbolos de esta palabra genera palabras que son iguales.

■

Un procedimiento recursivo para determinar que dos palabras son iguales es verificar la igualdad de los símbolos en cada posición de las palabras:

Para verificar la igualdad de las palabras `v` y `w`:

1. Si `(pVacia? v)` $\mapsto$ `#t` y `(pVacia? w)` $\mapsto$ `#t`, es claro que son iguales, ya que ambas palabras son la palabra vacía.
2. Si una de las palabras es vacía pero la otra no lo es, entonces inmediatamente es falso que las palabras sean iguales.
3. Si `v` $\mapsto$ `(`v_1`|v')`, `w` $\mapsto$ `(`w_1`|w')` y `(iguales? `v_1 w_1`)` $\mapsto$ `#t`, entonces significa que ambas palabras son iguales en al menos en el primer símbolo. Para que las palabras sean iguales en todos los símbos, se debe verificar la igualdad del resto de la palabra `v`, es decir de la palabra `v'`, con el resto de la palabra `w`, esto es con la palabra `w'`, sin duda esto sugiere un procedimiento recursivo.
4. Si no se cumple ninguna de las tres primeras opciones seguramente es `#f` que sean iguales, ya que se ha encontrado un símbolo en que son diferentes ambas palabras.

Nóte que una de las condiciones exige que una proposición sea `#t` y la otra sea `#f`, esto requiere una disyunción exclusiva, con la que se puede determinar que una palabra es vacía pero la otra no lo es.

```
(define p=?
  (λ (v w)
    (cond ((and (pVacia? v) (pVacia? w)) #t)
          ((ox (pVacia? v) (pVacia? w)) #f)
          ((equal? (car v) (car w))
                   (p=? (cdr v) (cdr w)))
          (else #f))))
```

Aunque el procedimiento anterior funciona adecuadamente, `DrRacket` cuenta ya con la primitiva `equal?` que hace el trabajo de comparar dos expresiones, si ambas expresiones son listas, se comparan de la manera en que ya se ha descrito, de modo que definiremos un sinónimo de `equal?` que esté acorde con la idea de verificar la igualdad entre palabras.

Código 4.9: *Verificación de palabras iguales*

```
1 ; (p=? v w) ↦ booleano
2 ; v : palabra
3 ; w : palabra
4 (define p=?
5   (λ (v w)
6     (equal? v w)))
```

■ **Ejemplo 4.9** Determine la igualdad de palabras en los siguientes casos:

1. $\langle\rangle = \langle\rangle$
2. $\langle 23\rangle = \langle 2\rangle$
3. $\langle 234\rangle = \langle 234\rangle$

```
> (p=? '() '())
#t
> (p=? '(2 3) '(2))
#f
> (p=? '(2 3 4) '(2 3 4))
#t
>
```

■

Si dos palabras no son iguales, entonces son **palabras diferentes**. Esta sentencia puede ser traducida en DrRacket como un predicado p!=? que sea #t cuando (p=? v w) ↦ #t; y #f en caso contrario.

Código 4.10: *Palabras diferentes*

```
; (p!=? v w) ↦ booleano
; v : palabra
; w : palabra
(define p!=?
  (λ (v w)
    (neg (p=? v w))))
```

4.5 Concatenación de palabras

Si v↤'(v_1 ... v_n) es una palabra de longitud n y w↤'(w_1 ... w_m) es otra palabra que es de longitud m, la **concatencación** de la palabra v con la palabra w que se puede escribir como vw y es una nueva palabra creada con los símbolos de la palabra v, seguidos de los símbolos de la palabra w:

$$'(v_1 \ \ldots \ v_n \ w_1 \ \ldots \ w_m)$$

Como (long v) ↦ n y (long w) ↦ m, al concatenar ambas palabras se logra una palabra de longitud (+ n m), que empieza con todos los símbolos de la palabra v y termina con todos los símbolos de la palabra w, en ese orden, así:

(equal? (long (*p v w)) (+ (long v)) (long w)) ↦ #t

Para modelar la función de concatenación se puede utilizar la primitiva append* [página 50], con lo que se tendrá una manera que permita aplicar la misma función para concatenar incluso más de 2 palabras.

Código 4.11: *Concatenar palabras*

```
; (*p ps ...) ↦ pal?
; ps : palabra ...
(define *p
  (λ ps
    (append* ps)))
```

■ **Ejemplo 4.10** En las siguientes interacciones se muestran tres diferentes situaciones al utilizar `*p`: primero una situación normal, se concatenan dos palabras no vacías. Segundo se concatena una palabra no vacía con un símbolo. Tercero se concatena un símbolo con una palabra.

```
> (*p '(a b c d) '(1 0 1 1))
'(a b c d 1 0 1 1)
> (*p '(a b c d) '(1 0 1 1) '(x y z))
'(a b c d 1 0 1 1 x y z)
> (*p '(a b c d) 'x)  ;; <------ 1
'(a b c d . x)
> (*p 'x '(1 0 1 1)) ;; <------- 2
... /rep.rkt:1088:24: append: contract violation
  expected: list?
  given: 'x
>
```

El mensaje ...contract violation..- etc es resultado de haber utilizado mal una función, en este caso la función `append` [`*p` es sinónimo de `append`]. El contrato de `append` exige que todos sus operandos sean listas, excepto por el último que puede o no ser una lista. Si el último operando no es una lista, entonces se tiene un par, como en la segunda interacción. ■

Un objeto de la forma `'(a1 ... an . b)`, para algún número n mayor que `0` se llama par. El par más pequeño que se escribe de esta manera es de la forma `'(a . b)`. Un par es diferente que una lista en el sentido de que siempre debe tener un `car` y un `cdr`. Así la lista vacía no es un par, pero sí es una lista. En este libro se ha preferido modelar los pares mediante listas, con el fin de utilizar en lo posible un mismo tipo de dato.

4.5.1 El elemento neutro de la concatenación

La concatenación es una operación binaria que tiene como elemento neutro a la palabra vacía `pVacia`. De hecho es un elemento neutro tanto por la izquierda como por la derecha. Si `w` es cualquier palabra de cualquier alfabeto, al concatenar `pVacia` por la derecha se obtiene:

$$(*p\ w\ pVacia) \mapsto w$$

de igual modo si se concatena por la izquierda:

$$(*p\ pVacia\ w) \mapsto w$$

La palabra vacía `pVacia` es el único elemento neutro de la concatenación.

4.5.2 Cerradura en la concatención

Sea S^* el conjunto de todas las palabras de longitud finita generadas con símbolos del alfabeto S. Aunque los elementos de S^* son de longitud finita, el conjunto es de cardinalidad infinita.

La concatenación permite generar palabras de longitud arbitraria que pertenecen a S^*. Suponga que $v \mapsto$ '$(v_1 \ \ldots \ v_n)$ y $w \mapsto$ '$(w_1 \ \ldots \ w_m)$ son dos palabras en S. Al hacer (*p v w), el resultado es una palabra '$(v_1 \ \ldots \ v_n \ w_1 \ \ldots \ w_m)$, que es de nuevo una palabra en S porque cada uno de los símbolos de la nueva palabra pertenecen a S. Incluso si ambas palabras fueran pVacia, la palabra vacía es una palabra en S.

4.5.3 Asociatividad en la concatenación

Sean x, y y z, cualesquiera tres palabras. Se observa que:

$$\texttt{(p=?\ \ (*p x (*p y z))\ (*p (*p x y) z))} \ \mapsto \texttt{\#t}$$

Si $x \leftarrow\!\!\shortmid$'$(x_1 \ \ldots \ x_n)$; $y \leftarrow\!\!\shortmid$'$(y_1 \ \ldots \ y_m)$ y $z \leftarrow\!\!\shortmid$'$(z_1 \ \ldots \ z_k)$, el resultado de (*p x (*p y z)) es la concatenación

$$\texttt{(*p '(}x_1 \ \ldots \ x_n\texttt{) '(}y_1 \ \ldots \ y_m \ z_1 \ \ldots \ z_k\texttt{))},$$

y de esta operación resulta:

$$\texttt{'(}x_1 \ \ldots \ x_n \ y_1 \ \ldots \ y_m \ z_1 \ \ldots \ z_k\texttt{)}.$$

Por otro lado el resultado de (*p (*p x y) z) es la concatenación

$$\texttt{(*p '(}x_1 \ \ldots \ x_n \ y_1 \ \ldots \ z_m\texttt{) '(}z_1 \ \ldots \ z_n\texttt{))},$$

y de esta operación resulta nuevamente:

$$\texttt{'(}x_1 \ \ldots \ x_n \ y_1 \ \ldots \ y_m \ z_1 \ \ldots \ z_k\texttt{)}.$$

Así [aunque no es el objetivo del libro] $\langle S^*;\ \texttt{*p}\rangle$ es un semigrupo [MH81, Gin68] que tiene a pVacia como elemento neutro bajo la operación asociativa de concatenación sobre las palabras finitas de S^*.

4.5.4 Escribir en una palabra

Si $w \mapsto$ '$(w_1 \ \ldots \ w_n)$ es una palabra de longitud n, al agregar símbolos en w se genera una nueva palabra. Es posible agregar símbolos por la izquierda o por la derecha. El único modo que la palabra permanezca sin cambio cuando se le escribe un símbolo, es cuando se escriba el símbolo nulo 'ε, que produce el mismo efecto que concatenar w con pVacia.

Si $S \leftarrow\!\!\shortmid$'$(s_1 \ \ldots \ s_m)$ es un alfabeto y s un símbolo del alfabeto S, se puede escribir s en la palabra w en dos pasos:

1. Se crea una palabra unitaria con el símbolo s.
2. Se concatena la palabra unitaria con w.

Si $w \mapsto$ '$(w_1 \ \ldots \ w_n)$ es la palabra, al escribir por la izquierda el símbolo s, se genera la nueva palabra:

$$\texttt{'(s w}_1 \ \ldots \ \texttt{w}_n\texttt{)},$$

si se escribe el símbolo por la derecha, se genera la palabra:

$$\texttt{'(w}_1 \ \ldots \ \texttt{w}_n \ \texttt{s)}.$$

El nuevo procedimiento `escribe` escribe el símbolo `s` en la palabra `w`, devolviendo una nueva palabra:

1. Si el símbolo que se desea escribir es el símbolo nulo `'`ε, entonces la palabra devuelta es `w`, sin importar si se desea escribir por izquierda o por derecha.
2. Si no se trata del símbolo nulo y se desea escribir `s` en la palabra `w` por derecha, entonces la palabra devuelva es `(*p w (punit s))`.
3. Si no se trata del símbolo nulo y se desea escribir `s` en `w` por la izquierda, la palabra devuelta será `(*p (punit s) w)`.

Código 4.12: *Escribe un símbolo en una palabra*

```
; (escribe s w [#:der #t]) ↦ pal?
; s : símbolo?
; w : palabra
; der : booleano = #t
(define escribe
  (λ (s w #:der [der #t])
    (cond ((nulo? s) w)
          (der (concat w (punit s)))
          (else (concat (punit s) w)))))
```

■ **Ejemplo 4.11** Las siguientes interacciones muestran cómo funciona el procedimiento `escribe` en situaciones estratégicamente seleccionadas:

```
> (escribe 's '(x r t))
'(x r t s)
> (escribe 's '(x r t) #:der #f)
'(s x r t)
> (escribe nulo '(x r t) #:der #f)
'(x r t)
> (escribe nulo '(x r t))
'(x r t)
>
```

■

4.5.5 Palabra inversa

Si `w`$\leftarrow\!\!\!\shortmid$`'(w`$_1$ `...` `w`$_n$`)` es una palabra, la **palabra inversa** de `w` es otra palabra que contiene los mismos símbolos que `w`, pero en orden inverso:

$$\texttt{(pInv w)} \mapsto \texttt{'(w}_n \ \ldots \ \texttt{w}_1\texttt{)}$$

Es posible definir un procedimiento recursivo para modelar la función de la `pInv` observando los siguientes casos:

$$(\texttt{pInv w}) \mapsto \begin{cases} \texttt{w}, \text{ cuando } \texttt{w} \mapsto \texttt{pVacia} \\ \\ (\texttt{escribe } \texttt{w}_1 \ \ (\texttt{pInv w'})), \text{ cuando } \texttt{w} \mapsto (\texttt{w}_1 | \texttt{w'}), \end{cases}$$

Pero `DrRacket` ofrece la primitiva `reverse`, que permite obtener la lista inversa de una lista.

La primitiva `reverse` toma una lista y devuelve otra lista con los mismos elementos pero en orden inverso.

```
(reverse lst) ↦  lista?
; lst : lista?
```

Código 4.13: *Palabra inversa*

```
1 ; (pInv w) ↦ lista?
2 ; w : lista?
3 (define pInv reverse)
```

La palabra inversa de una palabra tiene las siguientes propiedades que se ilustrarán con ejemplos:

1. `(pInv pVacia)` $\mapsto$ `pVacia`, esto es que la palabra inversa de la palabra vacía es la palabra vacía.

```
> (pInv pVacia)
'()
```

2. `(pInv (pInv w))` $\mapsto$ `w` lo que significa que la palabra inversa de la palabra inversa de una palabra `w` es la misma palabra `w`.

```
> (pInv (pInv '(a b c d)))
'(a b c d)
```

4.6 Potencia de una palabra

La **potencia de una palabra** es una operación sobre las palabras que utiliza la concatenación de palabras `*p`. La potencia de una palabra requiere dos parámetros, una palabra `w` que actúa como *base*; y un número entero no negativo denotado `n` que sirve como *exponente*. Así la palabra `w` elevada a la potencia `n` es la

concatenación de `w` consigo misma `n` veces, de acuerdo al siguiente procedimiento recursivo:

$$(\texttt{**p-rec w n}) \mapsto \begin{cases} \texttt{pVacia}, \text{ cuando } \texttt{n} \mapsto \texttt{0} \\ \\ \texttt{(*p (**p-rec w (- n 1)) w)}, \\ \text{cuando } \texttt{(> n 0)} \mapsto \texttt{\#t} \end{cases}$$

El procedimiento se ha nombrado `**p-rec` para enfatizar el proceso recursivo generado:

```
(define **p-rec
  (λ (w n)
    (if (= n 0)
        pVacia
        (*p (**p-rec w (- n 1)) w))))
```

Este procedimiento genera procesos recursivos [ASS96] que son costosos en términos de memoria y tiempo de ejecución. Es posible hacer un procedimiento que genera procesos iterativos, utilizando argumentos opcionales y diseñando una función que en su parámetro opcional se construya progresivamente en cada iteración la palabra resultante; aplicando manera recursiva la concatenación del resultado actual, con la palabra proporcionada como argumento. Una mejor propuesta es la que se muestra en el código del procedimiento `**p` para obtener la Potencia de una palabra.

Código 4.14: *Potencia de una palabra*

```
1 ; (**p w n [res]) ↦ pal?
2 ; w : palabra
3 ; n : numero-entero-no-neg
4 ; res : palabra = '()
5 (define **p
6   (λ (w n [res '()])
7     (if (= n 0)
8         res
9         (**p w (- n 1) (*p res w)))))
```

■ **Ejemplo 4.12** Obtenga la potencia 4 de las palabras `'(a b b)` y `'()`.

```
> (**p (pal 'a 'b 'b) 4)
'(a b b a b b a b b a b b)
> (**p (pal) 4)
'()
>
```

■

4.7 Prefijo y sufijo

Cualquier palabra w de algún alfabeto S, puede descomponerse en el **prefijo** y el **sufijo** de la palabra.

4.7.1 Prefijo de una palabra

Un **prefijo** de la palabra w↤'(w_1 ... w_n) es una palabra a↤'(a_1 ... a_k) con (< k n) ↦ #t, de tal forma que exista otra palabra c↤'(c_1 ... c_m) tales que:

$$\texttt{(p=? w (*p a c))} \mapsto \texttt{\#t}.$$

Esto significa que los primeros k símbolos de la palabra w (la longitud de a) son los que forman la palabra a y el resto forma la palabra c. Así la palabra w puede descomponerse en dos partes, la primera es el prefijo a y la segunda es c.

■ **Ejemplo 4.13** Si la palabra w↤'(l i b r e r o). La palabra a↤'(l i b r e) es un prefijo de w, porque al seleccionar la palabra c↤'(e r o) se puede hacer la concatenación (*p a c):

$$\begin{aligned}\texttt{(*p a c)} &\mapsto \texttt{(*p '(l i b r) '(e r o))}\\ &\mapsto \texttt{'(l i b r e r o)},\end{aligned}$$

verificando la igualdad entre w y la concatenación de a y c:

$$\texttt{(p=? w (*p a c))} \mapsto \texttt{\#t}.$$

■

Dos hechos fundamentales acerca del prefijo:

1. La palabra vacía es prefijo de cualquier palabra.
2. Una palabra es prefijo de ella misma.

Si (> (long a) (long w)) ↦ #t, a no puede ser prefijo de w porque tiene más letras y la concatenación no reduce la longitud de una palabra.

Para determinar si una palabra a es prefijo de otra palabra w, se toma la longitud del supuesto prefijo para verificar la igualdad entre las palabras a y los primeros (long a) símbolos de w.

Código 4.15: *Prefijo de una palabra*

```
; (prefijo? a w) ↦ booleano
; a : palabra
; w : palabra
(define prefijo?
  (λ (a w)
    (y (<= (long a) (long w))
       (p=? a (take w (long a))))))
```

■ **Ejemplo 4.14** Los siguientes ejemplos ilustran el funcionamiento de `prefijo?`

```
> (prefijo? '() '(a l c a n c i a))
#t
> (prefijo? '(a l c a n c i a) '(a l c a n c i a))
#t
> (prefijo? '(c a n) '(a l c a n c i a))
#f
> (prefijo? '(a l c a n) '(a l c a n c i a))
#t
>
```

■

La primitiva `take` devuelve una nueva lista tomando los primeros `pos` elementos de la lista `lst`.

```
(take lst pos) ↦ lista?
; lst : cualquiera?
; pos : numero-entero-no-negativo?
```

Una palabra `w`$\leftarrow\!\!\shortmid$`'(`w_1 ... w_n`)` tiene `(+ n 1)` prefijos, contando la palabra vacía de longitud `0` hasta `w` de longitud `(long w)`.

La **familia de prefijos** de una palabra `w` es la colección de todos los prefijos de `w` y se obtiene al tomar desde el extremo izquierdo de `w`, palabras de longitud `0`, `1`, ..., hasta `n`. En el procedimiento `prefijos` se utiliza la primitiva `build-list` para generar una lista con los números que representarán el tamaño de palabra del sufijo.

Código 4.16: *Familia de prefijos de una palabra*

```
1 ; (prefijos w) ↦ (listade pal?)
2 ; w : palabra
3 (define prefijos
4   (λ (w)
5     (map (λ (n) (take w n))
6          (build-list (+ 1 (long w)) values))))
```

La primitiva `build-list` genera una lista de `n` números enteros no negativos, donde el número consecutivo se calcula mediante el procedimiento `proc`.

```
(build-list n proc) ↦ lista?
; n : numero-entero-no-negativo?
; proc : (numero-entero-no-negativo? . -> . cualquiera)
```

■ **Ejemplo 4.15** Enliste la familia de prefijos de la palabra `'(g u e r r a)`.

```
> (prefijos '(g u e r r a))
'(() (g) (g u) (g u e) (g u e r) (g u e r r) (g u e r r a))
>
```

■

4.7.2 Sufijo

Si `w`, `a` y `c` son palabras, `c` es un **sufijo** de `w` si se verifica que `(p=? w (*p a c))` $\mapsto$ `#t`.

Podemos observar que si `c`$\leftarrow$`'(`c_1 ... c_m`)` es un sufijo de `w`, `c` es la última parte de la palabra `w`, de hecho son los últimos m símbolos de `w`. Si `c`$\leftarrow$`pVacia` entonces `w` es un sufijo de `w`, porque podemos hacer que `a`$\leftarrow$`w` y lograr que `(p=? w (*p a pVacia))` $\mapsto$ `#t`.

Para determinar qué palabras son elegibles para formar el sufijo [o prefijo] de una palabra `w`, se debe buscar en la misma `w`. Si son prefijos, buscar desde la izquierda; si son sufijos, desde la derecha de la palabra.

Código 4.17: *Sufijo de una palabra*

```
; (sufijo? c w) ↦ booleano
; c : palabra
; w : palabra
(define sufijo?
  (λ (c w)
    (y (<= (long c) (long w))
       (p=? c (take-right w (long c))))))
```

La primitiva `take-right` es similar a `take`, pero los símbolos son tomados desde la derecha.

```
(take-right lst pos) produce lista?
; lst : cualquiera
; pos : numero-entero-no-negativo?
```

■ **Ejemplo 4.16** Determine si una palabra es sufijo de otra en los siguientes casos:

1. ¿$\langle\rangle$ es sufijo de $\langle alcanzar\rangle$?
2. ¿$\langle alcanzar\rangle$ es sufijo de ella misma?
3. ¿$\langle canz\rangle$ es sufijo de $\langle alcanzar\rangle$?
4. ¿$\langle canzar\rangle$ es sufijo de $\langle alcanzar\rangle$?

```
> (sufijo?  '() '(a l c a n z a r))
#t
> (sufijo? '(a l c a n z a r) '(a l c a n z a r))
#t
> (sufijo? '(c a n z) '(a l c a n z a r))
#f
> (sufijo? '(c a n z a r) '(a l c a n z a r))
#t
>
```

■

De manera similar la familia de sufijos de una palabra `w` es la colección de todos los sufijos de `w`.

Código 4.18: *Familia de sufijos de una palabra*

```
; (sufijos w) ↦ (listade pal?)
; w : palabra
(define sufijos
  (λ (w)
    (map (λ (n) (take-right w n))
         (build-list (+ 1 (long w)) values))))
```

En general la familia de prefijos de una palabra `w`, no es igual a la familia de sufijos de la misma palabra `w`.

4.8 Subpalabras

Si además `b`↤`'(`b_1 ... b_j`)` es una palabra. La palabra `b` es una **subpalabra** de `w` si hay palabras `a` y `c` tales que:

`(p=? w (*p a b c))` ↦ `#t`.

Si `z` es subpalabra de `w` y `w`↤`(*p x z y)`, entonces se observa que `x` es un prefijo de `w` y `y` es un sufijo de `w`. De igual manera `(*p x z)` es prefijo de `w` y `(*p z y)` es sufijo de `w`.

■ **Ejemplo 4.17** Si `w`↤`'(z a p a t o)`; `z`↤`'(p a t)` es subpalabra de `w` porque si `x`↤`'(z a)` y `y`↤`'(o)` son otras dos palabras se verifica:

```
(*p '(z a) '(p a t) '(o))↦ '(z a p a t o)
  (p=? w '(z a p a t o))↦ #t
```

■

Código 4.19: *Determina si una palabra es subpalabra de otra*

```
; (subpalabra? z w) ↦ booleano?
; z : palabra?
; w : palabra?
(define subpalabra?
  (λ (z w)
    (existeUn (λ (pref) (sufijo? z pref))
              (prefijos w))))
```

■ **Ejemplo 4.18** Determine si `'(a l g a)` es subpalabra de la palabra `'(c a b a l g a t a)`.

```
> (subpalabra? '(a l g a) '(c a b a l g a t a))
#t
>
```

■

El razonamiento anterior se extiende para encontrar todas las subpalabras de una palabra. Así la familia de subpalabras de una palabra es la unión extendida de los sufijos de cada prefijo de `w`.

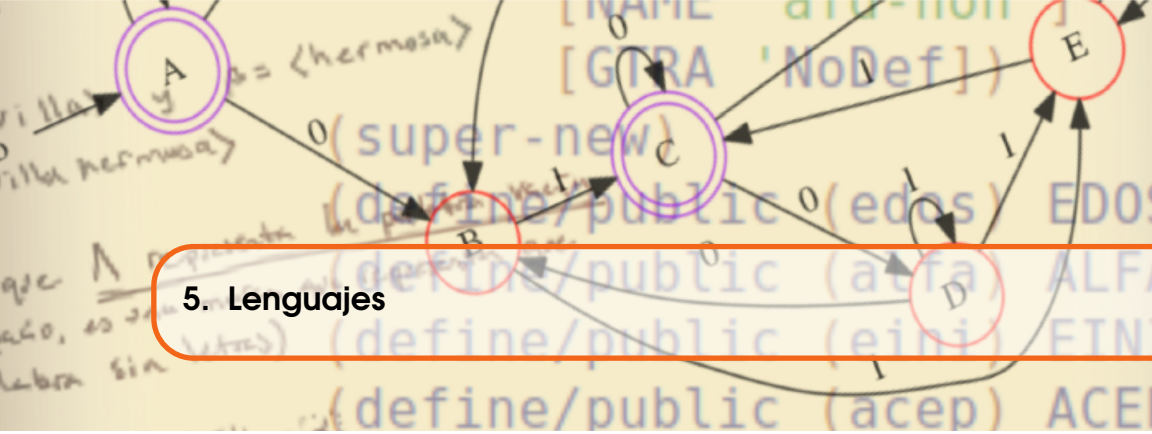
Código 4.20: *Familia de subpalabras de una palabra*

```
; (subpalabras w) ↦ (listade lista?)
; w : lista?
(define subpalabras
  (λ (w)
    (apply union* (map (λ (p) (sufijos p)) (prefijos w)))))
```

■ **Ejemplo 4.19** Enliste todas las subpalabras de `'(t i p o)`

```
> (subpalabras '(t i p o))
'((p) (i p) (t i p) (t) (t i) (i) (t i p o) (i p o)
(p o) (o) ())
>
```

■



5. Lenguajes

Con los símbolos de un alfabeto y la operación `escribe` (página 81), el número de palabras que se pueden formar crece sin límite.

Para ver este crecimiento supónga que el alfabeto es unitario, digamos `S`↤`'(a)`, un alfabeto bastante pequeño de una letra. La operación `escribe` permite construir palabras ya sea escribiendo símbolos por la izquierda o por la derecha. Empezando con `pVacia`, se crea una nueva palabra con la letra `'a`, luego para cada nueva palabra repetir el proceso escribiendo por la derecha cada símbolo en `S`:

- `(escribe 'a pVacia)` ↦ `'(a)`
- `(escribe 'a '(a))` ↦ `'(a a)`
- `(escribe 'a '(a a))` ↦ `'(a a a)`
- `(escribe 'a '(a a a))` ↦ `'(a a a a)`
- `(escribe 'a '(a a a a))` ↦ `'(a a a a a)`
- `(escribe 'a '(a a a a a))` ↦ `'(a a a a a a)`
- ...

Es fácil observar que con esta manera de crear nuevas palabras que contengan símbolos del mismo alfabeto, se pueden generar un número infinito de palabras. Así la cantidad de palabras que se pueden crear al escribir letras con un alfabeto de cardinalidad finita, ¡es infinito!

El alfabeto del idioma español tiene 27 letras minúsculas, suponiendo iguales las palabras con minúsculas o con mayúsculas. Las letras minúsculas son: `'(a b c d e f g h i j k l m n ñ o p q r s t u v w x y z)`, también es posible dar un método para generar todas las palabras, aunque este método nunca termine pues es imposible tenerlas todas.

Primero se generan todas las palabras unitarias, son 27 plabras unitarias. Luego a cada palabra se le escribe cada una de las letras del alfabeto ahora hay 729 palabras de dos letras (la gran mayoría sin significado en el idioma español). Luego a cada una de las 729 se le agrega una nueva letra de las 27, esto genera 19683 palabras de tres letras, la cantidad de palabras aumenta en orden exponencial y rápidamente se observa que el número de palabras que se pueden formar es infinito.

Sin embargo, las personas que conocen de las reglas que rigen el idioma español han ingresado casi unos 100,000 vocablos al idioma [Rea16], pero este número no es el número total de palabras con las que se pueden comunicar las personas en idioma español, faltan al menos los regionalismos y los modismos; hay personas que se atreven a decir que son unas 800,000 palabras. Aún así, 800,000 es un número mucho menor que infinito.

¿Qué tienen de especial las palabras que pertenecen al idioma español? ¿y las que pertenecen a otro idioma en particular? Lo único que tienen de especial estas palabras son que ellas tienen un significado.

Seleccionando las palabras que se utilizan es posible transmitir un mensaje, porque cada palabra tiene un significado en sí y juntas aportan un mensaje.

5.1 Creación de lenguajes

Un **lenguaje** es un conjunto donde todos sus elementos son palabras. Este conjunto puede ser finito (como el caso del español, aunque no se sepa bien cuántas son) o puede ser infinito. Un lenguaje puede ser un listado de palabras, o pueden seleccionarse las palabras que obedezcan a una regla de escritura.

■ **Ejemplo 5.1** Los siguientes son ejemplos de lenguajes:

- `'(())`. Es el lenguaje que solo tiene a la palabra vacía.
- `'((b e g i n) (e n d) (i f) (e l s e))`. Este lenguaje solo contiene cuatro palabras.
- Las palabras en el alfabeto de las vocales (del español) de longitud `5` y la palabra `(pal 'a)` es prefijo y sufijo de las palabras.
- El lenguaje de las palabras para hacer placas de automóviles, que constan de 3 números y 3 letras minúsculas, por ejemplo `'(9 3 5 v g w)`.
- $\{w \in \Sigma^* : |w| > 1\}$. Es el lenguaje de las palabras creadas con letras en un alfabeto Σ (cualquiera que sea), que tienen longitud mayor que 1.

■

Código 5.1: *Determina un lenguaje*

```
; (lenguaje? L) ↦ booleano
; L : lenguaje
(define lenguaje?
  (λ (L)
    (paraTodo(λ (w) (penS? w)) L)))
```

■ **Ejemplo 5.2** Determinar cuál de los siguientes conjuntos no es un lenguaje:

1. '(1 2 (2 4 1) (1 1))
2. '((1 2) (2 4 1) (1 1))
3. '(())

```
> (lenguaje?  '(1 2 (2 4 1) (1 1)))
#f
> (lenguaje?  '((1 2) (2 4 1) (1 1)))
#t
> (lenguaje?  '())
#t
>
```

■

5.1.1 Alfabeto generador de un lenguaje

El alfabeto generador de un lenguaje es el conjunto de cardinalidad mínima que contiene los símbolos necesarios para crear todas las palabras del lenguaje.

■ **Ejemplo 5.3** El alfabeto generador de el lenguaje '((h o l a) (a m i g o s) (g o m a) (m a g o s)) es el alfabeto '(a g h i l m o s). ■

El alfabeto generador de un lenguaje se obtiene con el alfabeto generador de cada palabra del lenguaje (ver página 76) y unir los conjuntos resultantes.

Un procedimiento efectivo resulta de la definición al aplicar la unión extendida a la colección de alfabetos generadores de las palabras del lenguaje.

```
; (SgenL L) ↦ lenguaje?
; L : (listade pal?)
(define SgenL
  (λ (L)
    (apply union* (map (λ (w) (Sgen w)) L))))
```

■ **Ejemplo 5.4** A partir del lenguaje L ↦ '((q p q) (p p p) (q p p)), obtener su lenguaje generador.

```
> (define L  '((q p q) (p p p) (q p p)))
> (SgenL L)
'(p q)
>
```

■

Si L es el lenguaje de todas las palabras finitas en A, donde A↤(SgenL L) es el alfabeto generador, entonces un conjunto B donde (subc? B A) ↦ #t para algún subalfabeto B, entonces todas las palabras generadas con B son palabras del lenguaje L.

(paraTodo (λ (w) (penS? w B)) L) ↦ #t

Lo anterior debe ser claro desde que todos los símbolos de A, también estan en B, puesto que A es un subconjunto de B.

5.1.2 Lenguaje vacio

El **lenguaje vacio** no tiene palabras.

Código 5.2: *El lenguaje vacio*

```
; Lenguaje vacio
(define lvacio '())
```

Para decidir si un lenguaje es precisamente lvacio se definirá como un sinónimo de empty?.

Código 5.3: *Verificación del lenguaje vacio*

```
; (lvacio? L) ↦ booleano?
; L : (listade pal?)
(define lvacio?
  (λ (L)
    (equal? L lvacio)))
```

■ **Ejemplo 5.5**

```
> (lvacio? '())
#t
>
```

■

5.1.3 Lenguaje unitario

El **lenguaje unitario** de un alfabeto finito S, es el lenguaje compuesto por todas las palabras unitarias del alfabeto S. Para crear el lenguaje unitario hay que generar palabras unitarias con cada símbolo del alfabeto dado S.

Código 5.4: *El lenguaje unitario de un alfabeto*

```
; (lunit S) ↦ lenguaje
; S : alfabeto
(define lunit
  (λ (S)
    (if (alf? S) (map (λ (s) (punit s)) S)
                 'ERR-no-simb)))
```

El lenguaje unitario es importante porque sirve como generador de nuevas palabras.

■ **Ejemplo 5.6** Sea S1 el alfabeto '(a b c d f g) y S2 el alfabeto '(0 1 2 3), escriba los lenguajes unitarios de estos dos alfabetos.

```
> (define S1 '(a b c d f g))
> (lunit S1)
'((a) (b) (c) (d) (f) (g))
> (define S2 '(0 1 2 3))
> (lunit S2)
'((0) (1) (2) (3))
>
```

■

El lenguaje unitario de un alfabeto vacío, es un conjunto vacío.

```
> (lunit '())
'()
> (lunit (conj nulo))
'()
> (lunit (conj nulo nulo))
'()
>
```

5.2 Cardinalidad de un lenguaje

La cardinalidad de un lenguaje es la cantidad de palabras que contiene. Debido a que un lenguaje puede ser definido de manera extensional como en los lenguajes unitarios de cardinalidad finita; o bien puede ser de cardinalidad infinita, como en el ejemplo al inicio del capítulo (ver página 89) aún cuando el alfabeto del lenguaje sea finito.

El lenguaje unitario de un alfabeto contiene únicamente palabras unitarias. Si S es un alfabeto donde (card S)↦ n, genera un lenguaje de cardinalidad n. La cardinalidad del lenguaje vacío es 0.

Para el caso finito, la cardinalidad de un lenguaje puede calcularse con el mismo procedimiento que la cardinalidad de un conjunto. Sin embargo para beneficio de las operaciones con lenguajes es necesario definir un nuevo procedimiento con un renombramiento.

Código 5.5: *Cardinalidad de un lenguaje*

```
1 ; (lcard L) ↦ numero-entero-no-negativo?
2 ; L : lenguaje
3 (define lcard card)
```

■ **Ejemplo 5.7** Calcule la cardinalidad del lenguaje S1↤'((a) (b) (c) (d) (f) (g)) y tambien del lenguaje S2↤'((0) (1) (2) (3))

```
> (define S1 (lunit '(a b c d f g)))
> S1
'((a) (b) (c) (d) (f) (g))
> (lcard S1)
6
> (define S2 (lunit '(0 1 2 3)))
> S2
'((0) (1) (2) (3))
> (lcard S2)
4
>
```

■

5.3 Palabra en un lenguaje

Para verificar la pertenencia de una palabra a un lenguaje definido explícitamente se verifica que la palabra en cuestión sea igual a alguna de las palabras del lenguaje; pero cuando el lenguaje dicta una regla para escribir las palabras, entonces se verifica que la palabra cumpla la regla del lenguaje. En este texto se utiliza el primer enfoque.

Una palabra `w` pertenece al lenguaje `L` si existe una palabra `p` en `L` que sea igual a `w`, considerando como criterio de igualdal el predicado `p=?` (página 77).

Código 5.6: *Pertenencia de una palabra a un lenguaje*

```
; (enl? w l) ↦ booleano
; w : pal?
; L : lenguaje?
(define enl?
  (λ (w L)
    (existeUn (λ (p) (p=? p w)) L)))
```

■ **Ejemplo 5.8** Las siguientes interacciones muestran el uso del predicado `enl?`

```
> (enl? '(p a d r e) '((h i j o) (h i j a) (m a d r e) (p a d r e) (t i o)
    (t i a)))
#t
> (enl? '(a b u e l o) '((h i j o) (h i j a)
(m a d r e) (p a d r e) (t i o) (t i a)))
#f
>
```

■

5.4 Sublenguajes

Sean `L` y `M` dos lenguajes. Decimos que `L` es un **sublenguaje** de `M` si todas las palabras en `L` también pertenecen al lenguaje `M`. Un cuantificador universal servirá

para definir el predicado `subl?`, siendo que se debe verificar la pertenencia de todas las palabras de `L` en `M`.

Código 5.7: *Sublenguaje de un lenguaje*

```
; (subl? L M)  booleano?
; L : (listade lista?)
; M : (listade lista?)
(define subl?
  (λ (L M)
    (paraTodo (λ (w) (enl? w M)) L)))
```

■ **Ejemplo 5.9** Defina los siguientes lenguajes:

- L↤'(() (1) (0) (1 0) (0 1) (1 1 1) (0 1 1)),
- M↤'((1 0) (1 1 0)),
- P↤'((1 0) (1 1 1)).

Determine si:

1. ¿L es sublenguaje de L?
2. ¿M es sublenguaje de L?
3. ¿P es sublenguaje de L?

```
> (define L '(() (1) (0) (1 0) (0 1) (1 1 1) (0 1 1)))
> (define M '((1 0) (1 1 0)))
> (define P '((1 0) (1 1 1)))
> (subl? L L)
#t
> (subl? M L)
#f
> (subl? P L)
#t
>
```

■

El lenguaje vacio `lnulo` es sublenguaje de cualquier lenguaje. Porque para cualquier lenguaje `L`, `lnulo` es sublenguaje de `L` si todas las palabras de `lnulo` son palabras de `M`, supongamos que `(subl? lnulo M)` ↦ `#f`, esto significaría que al menos una palabra `w` en `lnulo` no es de `L`, claramente no existe tal palabra, pues en el lenguaje vacio no hay palabras, de modo que `lnulo` debe ser un sublenguaje de `L`.

■ **Ejemplo 5.10**

```
> (subl? lnulo '((2) (1) (0))
#t
>
```

■

5.5 Igualdad entre lenguajes

Si L y M son lenguajes, L es igual a M, o bien que ambos son **lenguajes iguales** si L y M contienen exactamente las mismas palabras. Cuando L y M sean lenguajes iguales, podemos expresarlo como:

$$(\texttt{l=?}\ \texttt{L}\ \texttt{M}) \mapsto \#\texttt{t}$$

Para que L y M sean **lenguajes iguales** es necesario que todas las palabras de L sean palabras de M y que no exista palabra alguna en M que no sea palabra de L. La primera parte de la expresión establece que L debe ser un sublenguaje de M. La segunda parte se puede escribir como:

$$(\texttt{neg}\ (\texttt{existeUn}\ (\lambda\ (\texttt{w})\ (\texttt{neg}\ (\texttt{enl?}\ \texttt{w}\ \texttt{L})))\ \texttt{M})),$$

pero al simplificar esta expresión quitando las negaciones se puede reescribir como

$$(\texttt{paraTodo}\ (\lambda\ (\texttt{w})\ (\texttt{enl?}\ \texttt{w}\ \texttt{L}))\ \texttt{M}),$$

es decir, que todas las palabras de M deben pertenecer a L, es decir que M debe ser un sublenguaje de L. Así, L debe ser un sublenguaje de M; y desde luego, M debe ser un sublenguaje de L:

$$(\texttt{y}\ (\texttt{subl?}\ \texttt{L}\ \texttt{M})\ \ (\texttt{subl?}\ \texttt{M}\ \texttt{L})) \mapsto \#\texttt{t}$$

Código 5.8: *Igualdad entre lenguajes*

```
; (l=? L M) ↦ booleano?
; L : (listade pal?)
; M : (listade pal?)
(define l=?
  (λ (L M)
    (y (subl? L M) (subl? M L))))
```

Observe que en lenguajes así como en conjuntos, la siguiente igualdad permanece:

$$(\texttt{l=?}\ \texttt{L}\ \texttt{L}) \mapsto \#\texttt{t},$$

ya que todas las palabras en L (en el primer operando) están en L (el segundo operando) y en sentido inverso también.

■ **Ejemplo 5.11**

```
> (l=? '((4) (3) (2) (1) (0)) (lunit '(0 1 2 3 4)))
#t
> (l=? lvacio lnulo)
#f
>
```

■

5.6 Operaciones con lenguajes

En esta sección se extenderán los conceptos de concatenación, potencia e inverso creados para palabras, de modo que sea posible hacer operaciones con lenguajes.

5.6.1 Concatenación de lenguajes

Concatenar lenguajes es una operación que nos permite extender todas las palabras de un lenguaje con todas las palabras de otro lenguaje. La **concatenación del lenguaje** `L` con el lenguaje `M`, es el conjunto de todas las palabras `ab`, donde `a` es una palabra en `L` y `b` es una palabra en `M`.

Entonces la concatenación de los lenguajes `L` con `M` se conforma de la concatenación de cada palabra en `L` con cada palabra en `M`. Nombraremos `*l` a la operación de concatenar dos lenguajes. Es fácil ver que la cardinalidad de la concatenación de los dos lenguajes es el producto de la cardinalidad de ambos lenguajes, ya que se hace una concatenación por cada palabra en `L` con cada palabra en `M`.

Código 5.9: *Concatenación de lenguajes*

```
; (*l L M) ↦ lenguaje
; L : lenguaje
; M : lenguaje
(define *l
  (λ (L M)
    (append* (map (λ (a) (map (λ (b) (*p a b)) M))
                  L))))
```

Uno de los argumentos de la primitiva `map` [ver página 46] es un procedimiento, que en este caso es otra expresión `map`. Si `(card L)` ↦ `l` es la cardinalidad de la lista externa y `(card M)` ↦ `m` es la cardinalidad de la lista interna, el resultado de un `map` anidado es una lista que contiene `l` listas y cada una de estas con `m` elementos.

```
> (map (λ (l) (map (λ (m) (list l m))
                   '(1 2 3))) '(x y))
'(((x 1) (x 2) (x 3)) ((y 1) (y 2) (y 3)))
```

■ **Ejemplo 5.12** Si `L` ↦ `'((l i b r) (f l o r))` y `M` ↦ `'((e r o) (e r i a))` son dos lenguajes, ¿cuál es la concatenación de `L` con `M`?

```
> (define L '((l i b r) (f l o r)))
> (define M '((e r o) (e r i a)))
> (*l L M)
'((l i b r e r o) (l i b r e r i a) (f l o r e r o) (f l o r e r i a))
>
```

■

Si S_L↤(SgenL L) y S_M↤(SgenL M) son los alfabetos generadores de los lenguajes L y M respectivamente, las palabras en (*l L M) inducen el alfabeto generador:

(union S_L M_L)

5.6.2 Lenguaje identidad

Existe un lenguaje que al participar en la concatenación de lenguajes, no altera el producto ni al concatenar por la izquierda ni por la derecha. Este lenguaje funciona como elemento identidad bajo la operación *l, que se denomina lid, definido como:

Código 5.10: *Lenguaje identidad*

```
;; El lenguaje identidad
(define lid '(()))
```

A diferencia del lenguaje vacío lvacio que no tiene palabras, el lid contiene únicamente una sola palabra, la pVacia, es decir su cardinalidad no es cero.

lvacio ↦ '(); *Este es el lenguaje vacío.*

lid ↦ '(()); *Este es el lenguaje identidad.*

Claro que es necesario un predicado que permita determinar si un lenguaje es precisamente el lenguaje identidad.

Código 5.11: *Determina el lenguaje identidad*

```
; (lid? L) ↦ booleano?
; L : (listade lista?)
(define lid?
  (λ (L)
    (l=? L lid)))
```

■ **Ejemplo 5.13** Calcular la concatenación de lid con los lenguajes L y M definidos anteriormente.

```
> (*l lid L)
'((l i b r) (f l o r))
> (*l M lid)
'((e r o) (e r i a))
>
```

■

5.6.3 Concatenación extendida de lenguajes

La idea de concatenar lenguajes se puede extender para crear el concepto de la **concatenación extendida de lenguajes**, que permite concatenar 0, 1 o más lenguajes. Denotaremos la operación de concatenación extendida por *L y requiere una lista indefinida con cero o más lenguajes. El lenguaje generado por

la concatenación extendida de lenguajes tendrá una cardinalidad igual al producto de las cardinalidades de los lenguajes que participan en la concatenación extendid y se verifica la siguiente igualdad

$$(= (\texttt{card } (*\texttt{L } L_1 \quad \ldots \quad L_n)) \ (* \ (\texttt{card } L_1) \ldots (\texttt{card } L_n))) \mapsto \texttt{\#t}$$

Si `LLs` es la lista de lenguajes, el procedimiento de concatenación extendida es:

1. Si `LLs` $\mapsto$ `'()`, la concatenación extendida es `lnulo`.
2. Si `LLs` $\mapsto$ (L_1|`LLs'`), se debe construir la concatenación extendida considerando ahora `LLs'` con la concatenación de los lenguajes `res` y L_1, donde `res` es un lenguaje con valor provisional que identifica el lenguaje que se está construyendo y que inicialmente `res` $\mapsto$ `lid`, cuando la lista `LLs` llegue a ser vacía, entonces el valor actual de `res` contendrá la concatenación extendida buscada.

Código 5.12: *Concatenación extendida de lenguajes*

```
; (*L LLs ...+) ↦ lenguaje?
; LLs : (listade lenguaje?)
(define *L
  (λ LLs
    (letrec ([lccat-aux  (λ (LLs [res lid])
                           (if (lvacio? LLs)
                               res
                               (lccat-aux (cdr LLs)
                            (*l res (car LLs))))))])
      (if (lvacio? LLs)
          lvacio
          (lccat-aux LLs)))))
```

`letrec` permite crear definiciones locales como `let`, pero a diferencia de este, utilizar `letrec` permite definir procedimientos recursivos.

```
(letrec ([id val-expr] ...) cuerpo ...+)
; id : cualquier identificador
; val-expr : expresión que proporciona el valor a id
; cuerpo : expresiones donde id es aplicable
```

■ **Ejemplo 5.14** Calcule la concatenación extendida de los siguientes lenguajes:

- `'((u n) (u n a))`;
- `'((l i b r) (f l o r))`;
- `'((e r o) (e r i a))`.

```
> (*L '((u n) (u n a)) '((l i b r) (f l o r)) '((e r o) (e r i a)))
'((u n l i b r e r o) (u n l i b r e r i a) (u n f l o r e r o)
(u n f l o r e r i a) (u n a l i b r e r o) (u n a l i b r e r i a)
(u n a f l o r e r o) (u n a f l o r e r i a))
>
```

■

5.7 Potencia de un lenguaje

De manera similar que las palabras, en los lenguajes también es posible definir un concepto **potencia**, la idea general es la misma, aplicar una función al mismo argumento varias veces, en este caso el argumento es un lenguaje.

La potencia de un lenguaje es una operación que afecta a las palabras contenidas en el lenguaje. La potencia de un lenguaje es la concatenación de un lenguaje en sí mismo una cantidad predeterminada de veces; de modo que tiene a la concatenación de lenguajes como su operación fundamental. La `n`-ésima potencia de un lenguaje `L` se puede denotar como `(**l L n)` y se define recursivamente como:

$$(\texttt{**l L n [res lid]}) \mapsto \begin{cases} \texttt{res}, \text{ cuando } \texttt{n} \mapsto \texttt{0} \\ \\ \texttt{(**l L (- n 1) (*l res L))}, \\ \text{cuando } \texttt{(> n 0)} \mapsto \texttt{\#t} \end{cases}$$

Código 5.13: *Potencia de un lenguaje*

```
; (**l L n) ↦ lenguaje
; L : lenguaje
; n : numero-entero-no-negativo?
(define **l
  (λ (L n [res lid])
    (if (= n 0)
        res
        (**l L (- n 1) (*l res L)))))
```

■ **Ejemplo 5.15** Calcular la tercera potencia del lenguaje `B` $\mapsto$ `'((0) (1))`.

```
> (define B '((0) (1)))
> (**l B 3)
'((0 0 0) (0 0 1) (0 1 0) (0 1 1) (1 0 0) (1 0 1) (1 1 0) (1 1 1))
>
```

■

Las siguientes expresiones son verdaderas para cualquier lenguaje `M`:

1. `(**l M 0)`$\mapsto$ `lid`, esto es `(**l M 0)`$\mapsto$ `'(())`.
2. `(**l M 1)`$\mapsto$ `M`.

5.8 Cerradura de Kleene

La **cerradura de Kleene** es un lenguaje generado por una operación unaria, que llamaremos `Kleene*` y que actúa sobre un alfabeto generador `S`. Esta operación y el lenguaje que esta operación genera, llevan en su nombre el apellido

del célebre matemático estadounidense Stephen C. Kleene, quien entre otras cosas estableció las bases de la teoría de las funciones recursivas [Kle81], que junto con el cálculo-λ [Chu41, Rog87] fundamentan el lenguaje de programación `DrRacket` y el de otros lenguajes de programación [Lan65a, Lan65b].

El término *operación unaria* se refiere a *aridad* de la operación. La aridad es la cantidad de operandos que requiere un operador. Así que una operación unaria requiere solamente un operando, una operación binaria requiere dos y así en adelante. En `DrRacket` es posible saber la aridad de un procedimiento mediante `procedure-arity` por ejemplo `(procedure-arity **1)` $\mapsto$ `2`

La cerradura de Kleene se describe como el lenguaje que contiene a todas las palabras de longitud finita que se pueden crear con los símbolos de un alfabeto `S`, incluyendo a la palabra vacía `'()`. La cerradura de Kleene es un lenguaje de cardinalidad infinita, aunque las palabras que contiene son de longitud finita.

En la teoría de lenguajes formales, se utiliza $*$ como exponente en un alfabeto, como en Σ^*, para denotar la cerradura de Kleene del alfabeto generador Σ:

$$\Sigma^* = \{\langle \sigma_1 \ldots \sigma_k \rangle | (\forall i \in \{1,\ldots,k\} : \sigma_i \in \Sigma) \wedge (k \ll \infty)\}.$$

Una parte clave de esta descripción es la longitud finita de las palabras, significa que la cerradura de Kleene contiene a todas las palabras formadas con símbolos del alfabeto `S`, desde la palabra vacía que es de longitud `0`, hasta las palabras de longitud tan arbitrariamente grande como se quiera, pero no contendrá palabras de longitud infinita.

Desde la perspectiva de conjuntos, la cerradura de Kleene es la unión de los lenguajes obtenidos como potencias del lenguaje unitario de `S`, desde la potencia `0` en adelante.

Si `M` $\mapsto$ `(lunit S)` es el lenguaje unitario del alfabeto `S`, la cerradura de Kleene se puede entender como la unión extendida sobre todas las potencias del lenguaje `M`.

```
(Kleene* S)↦ (union* (**1 M 0) (**1 M 1) ...)
           ↦ (union* '(()) M (**1 M 2) ...)
```

El problema con esta definición es que computacionalmente no es posible crear un procedimiento efectivo que calcule todas las palabras de este lenguaje, simplemente porque se requiere calcular las potencias de un lenguaje desde la potencia `0` hasta `+inf.0`.

`+inf.0` es una constante primitiva, que en `DrRacket` equivale a $+\infty$. Tanto la constante `+inf.0`, así como las constantes `-inf.0`, `+inf.f` y `-inf.f` no son números, sino representaciones de infinito (e infinito negativo), aunque es posible comparar un número con algún infinito, por ejemplo `(< -inf.0 100)` $\mapsto$ `#t`. Las constantes sobre infinito, `+nan.0` (*not a number*) y `-nan.0` estan definidas bajo el estándar IEEE-754.

5.8.1 Cerradura finita de Kleene

Aún cuando es imposible definir efectivamente un procedimiento para obtener todas las palabras en `(Kleene* S)` de algún alfabeto finito `S`, sí es posible definir un lenguaje de cardinalidad finita que se acerque a `(Kleene* S)` tanto como se necesite; esta aproximación a la cerradura de Kleene se puede denotar como la **cerradura finita de Kleene**, simbolizada por `nKleene*`. El lenguaje que genera la operación `nKleene*` es un sublenguaje propio de la cerradura de Kleene, es una aproximación efectiva de la unión de las primeras `n` potencias del lenguaje unitario del alfabeto `S` y que incluye por supuesto a la potencia `0`.

Código 5.14: *Cerradura finita de Kleene*

```
; (nKleene* S [n]) ↦ lenguaje?
; S : alfabeto?
; n : numero-entero-no-negativo? = 10
(define nKleene*
  (λ (S [n 10])
    (let ((L    (lunit S)) ; El lenguaje unitario de S.
          (pots (build-list (+ n 1) values)))
      (append* (map (λ (i) (**l L i)) pots)))))
```

En el código de la definición de `nKleene*` hay cuatro elementos de programación en los que es conveniente profundizar:

1. El uso de `let` para la creación de variables locales. Se usa para evitar cálculos repetidos, particularmente de `(lunit S)`, que se debe hacer para el cálculo de cada potencia.
2. El uso de `append*`, en lugar de `union*`. Se observa que el lenguaje `(**l L i)` es disjunto a `(**l L j)` cuando `(= i j)` ↦ `#f`, de modo que con el fin de agilizar el cómputo se prefiere anexar los lenguajes.
3. El uso de `build-list`, que permite crear una lista, en este caso de `(+ n 1)` valores y cada valor es generado por `values`, que es propiamente una lista desde `0` hasta `(- n 1)`. El uso de `lpots` es para crear todas las potencias para elevar el lenguaje a cada una de esas potencias.
4. El valor de `n` dado por omisión. El valor asignado a `n` por omisión es arbitrario.

■ **Ejemplo 5.16** Calcular la cerradura de Kleene hasta la potencia 4 del alfabeto `S` ↦ `'(a bb)`.

```
> (define S '(a bb))
> (nKleene* S 4)
'(() (a) (bb) (a a) (a bb) (bb a) (bb bb) (a a a) (a a bb) (a bb a)
(a bb bb) (bb a a) (bb a bb) (bb bb a) (bb bb bb) (a a a a) (a a a bb)
(a a bb a) (a a bb bb) (a bb a a) (a bb a bb) (a bb bb a) (a bb bb bb)
(bb a a a) (bb a a bb) (bb a bb a) (bb a bb bb) (bb bb a a) (bb bb a bb)
(bb bb bb a) (bb bb bb bb))
>
```

■

5.8.2 Cardinalidad de la cerradura finita de Kleene

Si `L↤(lunit S)` es el lenguaje unitario del alfabeto `S`, la cardinalidad de la cerradura finita de Kleene para el alfabeto `S` está dada por

```
(apply + (map (λ (i) (card (**l L i))) (build-list (+ n 1) values))),
```

que es la suma de las cardinalidades de las primeras `n` potencias de `L`:

```
(+ (card (**l L 0)) (card (**l L 1)) ... (card (**l L n))).
```

Calcular de este modo es ineficiente porque para la `n`-ésima potencia de un lenguaje, se requiere el cálculo de la potencia anterior y a su vez se requiere la anterior a esta y así sucesivamente hasta la potencia `0`, que es redundante porque `(**l M 0)` se habrá calculado `n` veces, `(**l N 1)` se habrá calculado `(- n 1)` veces y así cada término de la suma.

Un modo mas eficiente es observar que `(**l M 0)` tiene un solo elemento; `(**l M 1)` ofrece `k` nuevas palabras y cada nueva potencia multiplica el número de palabras en factor de `k`. Como `(card S)` $\mapsto$ `k` entonces:

```
(card (nKleene* S n)) ↦ (apply + (map (λ (i) (expt k i)) (0 1 ... n)),
```

esto es:

```
(card (nKleene* S n)) ↦ (+ (expt k 0) (expt k 1) ... (expt k n)),
```

que finalmente es la suma de las primeras `n` potencias de `k`. Recordemos que `k` es la cantidad de símbolos del alfabeto `S`, de modo que `(card (nKleene* S n))` produce `(/ (- 1 (expt k (+ n 1))) (- 1 k))`.

La función `(expt z w)` eleva la base `z` a la potencia `w`.

```
(expt z w) ↦ numero?
; z : number?
; w : number?
```

Al considerar un único alfabeto `S` de cardinalidad finita `k`, se observa que a medida que la máxima potencia `n` a ser calculada aumenta, la cantidad de palabras aumenta exponencialmente y se acerca cada vez más a la cantidad de palabras que contiene el lenguaje generado por `Kleene*` que es infinito.

■ **Ejemplo 5.17** Calcular la cardinalidad de la cerradura finita de Kleene hasta la potencia 5 del alfabeto `'(a b c d e f)`.

```
> (+ (expt 6 0) (expt 6 1) (expt 6 2) (expt 6 3) (expt 6 4) (expt 6 5))
9331
> (/ (- 1 (expt 6 (+ 5 1))) (- 1 k))
9331
>
```

■

Una cosa más es crear un predicado `Kleene*?` que determine si una palabra pertenece a la cerradura de Kleene para un alfabeto determinado, sin tener que verificar una a una cada palabra del lenguaje.

Este predicado funciona con dos argumentos. El primero debe ser una palabra o bien un lenguaje. El segundo debe ser el alfabeto generador de la cerradura de Kleene. Operativamente una palabra es modelada como una lista y un lenguaje es modelado como una lista de listas.

El predicado `Kleene*?` devuelve `#t` si el argumento es una palabra compuesta con símbolos del alfabeto; también devuelve `#t` si cuando el objeto es un lenguaje, todas las palabras están compuestas con símbolos del alfabeto. Por otro lado, el predicado debe devolver `#f` en cualquier caso distinto.

El primer argumento se llamará `x`, porque puede ser una palabra o un lenguaje. Se consideran dos casos, uno cuando se trata de una palabra y el otro es cuando se trata de un lenguaje:

1. Para reconocer que `x` es una palabra creada únicamente con símbolos de un alfabeto `S` se puede verificar con `penS?` [ver página 73].
2. Para reconocer que el argumento `x` es un lenguaje generado con palabras de unalfabeto `S` se deben cumplir dos condiciones:
   a) Que `x` sea una lista.
   b) Y que para todos los elementos de la lista `x`, debe ser cierto que son palabras del alfabeto `S`.

Código 5.15: *Pertenencia de una palabra o un lenguaje a Kleene**

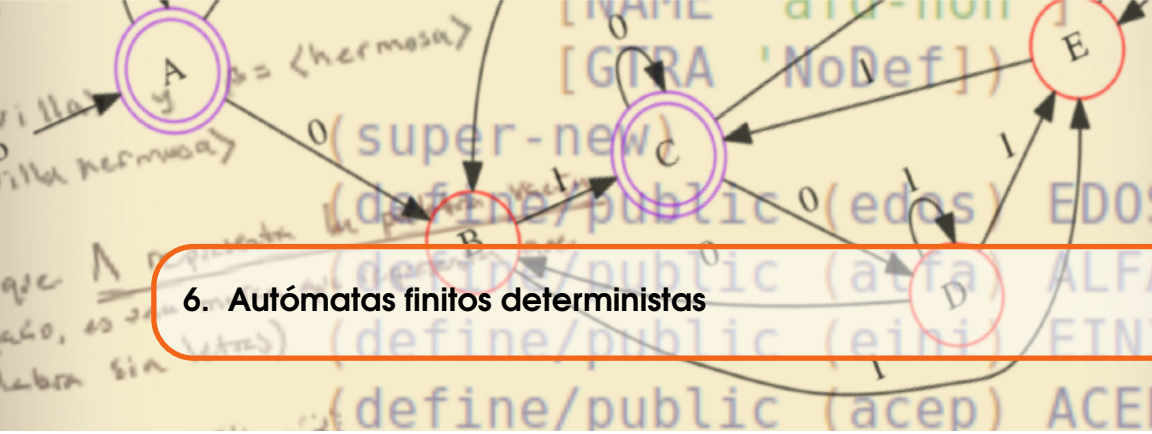
```
; (Kleene*? x S) ↦ booleano?
; x : (or palabra? lenguaje?)
; S : alfabeto?
(define Kleene*?
  (λ (x S)
    (cond ((penS? x S) #t)
          ((y (lenguaje? x)
              (paraTodo (λ (w) (penS? w S)) x)) #t)
          (else #f))))
```

■ **Ejemplo 5.18** Observe las siguientes interacciones

```
>(Kleene*? '(1 0 0) '(0 1))
#t ; Porque es una palabra en el alfabeto '(0 1).
>(Kleene*? '(1 0 2) '(0 1))
#f ; Porque no es una palabra en el alfabeto '(0 1).
>(Kleene*? '((1 0 0) () (1) (1 0 0)) '(0 1))
#t; Todas las palabras son del alfabeto '(0 1).
>(Kleene*? '((1 0 0) () (1) (1 2 0)) '(0 1))
#f;La palabra '(1 2 0) no es del alfabeto '(0 1).
>(Kleene*? 1 '(0 1))
#f ; Porque 1 no es una palabra ni un lenguaje.
```

■

II Autómatas



6. Autómatas finitos deterministas

6.1 Notas históricas

Los **autómatas finitos deterministas** (AFD%) son herramientas computacionales que permiten modelar problemas de decisión sobre secuencias de eventos; se trata de decidir si una secuencia de eventos recibida pertenece a la clase de secuencias permitidas. También permiten modelar el comportamiento de aquellos fenómenos que puedan ser observables en diferentes estados en diferentes momentos y donde cada cambio de estado ha sido provocado por la ocurrencia de algún evento. Se llaman deterministas porque se conoce cuál debe ser la reacción del AFD% ante cualquier evento reconocible que se presente.

Los AFD% o al menos la idea que subyace en ellos, se han usado y aplicado desde mucho antes de la revolución industrial, quizá desde el uso de las máquinas que de manera automática reaccionan ante algún tipo de eventos, como un instrumento musical automático, el telar de Jacquard, o la máquina de cálculo de Babbage [Koe01].

Uno de los primeros trabajos en los que se formalizó la idea de autómatas finitos, fué el trabajo de McCulloch y Pitts [MP43], quienes establecieron algunos de los principios teóricos de las redes neuronales, haciendo mención de una arquitectura de redes, un alfabeto binario y un principio de reacción de las neuronas

ante sus entradas, aunque el término «autómata finito» se hizo más común en la década de 1950 con los trabajos de George H. Mealy [Mea55] y Edward F. Moore [Moo56], ambos trabajos sobre estudios de máquinas secuenciales que generan salidas, esto es que a medida que el autómata analiza la secuencia de eventos, éste va reaccionando y realizando ciertas actividades predefinidas.

A finales de la década de 1950, Arthur W. Burks y Hao Wang [BW57a, BW57b] dieron a conocer sus estudios sobre la clase de autómatas determinísticos que pronto sentarían las bases para futuros estudios sobre las aplicaciones de los autómatas finitos. En 1958, Claude Shannon [Sha58] dió un panorama general de la importante obra de John Von Neumman en el campo de los autómatas, dando a conocer al menos dos areas de oportunidad dentro de este campo, la auto reproducción de autómatas y la construcción de máquinas confiables a partir de componentes no confiables.

Michael O. Rabin y Dana S. Scott [RS59] dieron a conocer los fundamentos teóricos del concepto actual de «autómata finito» como clasificadores, ellos vislumbraron estas máquinas teóricas como cajas negras que podían responder «sí» o «no», ante una cantidad indefinida (pero finita) de preguntas; en cada pregunta, esta máquina es capaz de responder con alguna de sus dos posibles respuestas y la respuesta a la última pregunta es considerada como la respuesta definitiva de la máquina. Aún más, se dió a concer el concepto de máquinas no-determinísticas. El trabajo de Rabin y Scott fué tan importante que en 1976 se les otorgó el prestigiado y máximo galardón que la comunidad de computación otorga, el premio Turing [Ass16].

Debido a su gran importancia y utilidad, en los siguientes años se produjo mucha información sobre la teoría de lenguajes formales, entre los que se destaca el trabajo de Abraham Ginzburg [Gin68], quien analizó la teoría de autómatas existente, especificando la idea de «semiautómata» como parte fundamental de los autómatas finitos.

Años más tarde en 1979, se publicó el libro *Introduction to Automata Theory, Languages, and Computation* [HU79], que ha servido como referente en los recientes cursos de autómatas finitos y lenguajes formales.

Los autómatas finitos, en particular las máquinas de Mealy y de Moore, han servido como modelo para representar el comportamiento de máquinas que reaccionan ante los eventos a los que han sido sometidas; han sido el fundamento de otros campos de estudio como la teoría de sistemas multiagentes [NS03], robótica [YYWL11], control [Zad13], por mencionar unos ejemplos.

Otras aplicaciones en las que se han utilizado los conceptos de la teoría de autómatas finitos han sido el procesamiento de lenguaje natural [Moh96] y los lenguajes formales [Har78, HU79], que han servido para crear lenguajes de programación.

6.1.1 La idea central

Los autómatas finitos deterministas, son esencialmente modelos teóricos de sistemas que son observables en un estado de entre un conjunto finito de estados y que reaccionan ante la ocurrencia de un evento dentro de una cantidad finita de eventos específicos. Cuando ocurre alguno de esos eventos, el autómata reacciona cambiando de estado [si es necesario].

El AFD% es finito porque tiene un número finito de estados y un número finito de símbolos que representan a los eventos que puede reconocer y ante los que el autómata finito reacciona. Cada posible combinación de estado–evento se conoce y es determinístico porque hay una única reacción precisa que dirige el AFD% a alguno de sus estados. La actividades realizadas por los AFD% son rutinarias, por lo que se han comparado con las actividades que solían realizar los computadores.

«Computador» era una actividad que al principio fué realizada por personas y que posteriormente, con los avances tecnológicos se crearon máquinas que hacían el trabajo rutinario de manera más eficiente. El término «computador» era el nombre del puesto de trabajo de la persona [Abb03, Cer91, Gri13] que se encargaba de realizar los procesos automáticos con la ayuda de una máquina computadora, como una sumadora.

Un AFD% empieza a trabajar cuando se le presenta el primer símbolo de una secuencia finita de símbolos (palabra), que representa cierta cadena de eventos que el AFD% es capaz de reconocer y la actividad de un AFD% termina después de haber analizado el último símbolo de la palabra que inició su actividad. El producto que esta máquina teórica produce está determinado por el último estado alcanzado depués de haber analizado cada evento (símbolo). En dependencia del tipo de estado final, se ha de clasificar la palabra leída.

Para los interesados en el área de las ciencias computacionales, la teoría de autómatas es muy importante porque permite entre otras cosas:

1. Saber cómo se realiza el cálculo de las funciones y en general cómo los programas de cómputo pueden resolver esos problemas.
2. Permiten establecer dos categorías importantes sobre los problemas, aquellos que puede resolver un autómata y aquellos que no pueden [MM67].

6.2 Definición de los AFD%

Un AFD% está definido por una tupla de 5 elementos:

$$\texttt{AFD\%} \mapsto \langle \texttt{Q}\ \texttt{S}\ \texttt{T}\ q_0\ \texttt{A} \rangle$$

1. Un conjunto no vacío Q↤(q$_1$|Q'), llamado los **estados** del autómata. .
2. Un conjunto S↤(s$_1$|S'), llamado el **alfabeto** de símbolos que representan el conjunto de eventos que puede reconocer el AFD%. Los términos «símbolo», «evento» o «letra» son sinónimos.

3. Una función `T:Q×S→Q`. La función es definida explícitamente por una colección de la forma `'((q s q`$_d$`) ...+)`, con `q` y `q`$_d$ son estados y `s` una letra del alfabeto `S`, esto es `(en? q Q)` $\mapsto$ `#t`; `(en? s S)` $\mapsto$ `#t` y `(en? q`$_d$ `Q)` $\mapsto$ `#t`. El estado `q`$_d$ es el estado destino. La información en `T`, permite crear el método `tr` llamado **función de transición** que asocia un par *estado-símbolo* con un estado destino `(Tr q s)`$\mapsto$ `q`$_d$ que permite determinar en qué nuevo estado se encontrará el `AFD%`, considerando su estado actual `q` y al reconocer algún evento `s`. Aunque tienen el mismo nombre, `tr` es un procedimiento, mientras que `T` es un conjunto de 3-tuplas. El conjunto de tuplas `T` permite representar el comportamiento del `AFD%`, esta representación conoce como **lista de transiciones**, o **reglas de transición** .
4. Un único estado `q`$_0$ distinguido de entre los estados. Es el **estado inicial**, seleccionado arbitrariamente entre los estados de `Q`, así `(en? q`$_0$ `Q)` $\mapsto$ `#t`. El autómata iniciará el análisis de cualquier palabra desde este estado. Aunque la selección es arbitraria, en la práctica real, se debe establecer cuidadosamente para que el modelo computacional sea adecuado.
5. Un subconjunto no vacío de estados `A`. Conjunto de estados **aceptores** que permitirá hacer la clasificación de las palabras, así `(subc? A Q)` $\mapsto$ `#t`.

☞ En la notación convencional, un autómata finito determinista es una quíntupla $\langle Q, \Sigma, \delta, q_0, A\rangle$, donde:

- $Q = \{q_1, \ldots, q_n\}$, es el conjunto de estados;
- $\Sigma = \{\sigma_1, \ldots, \sigma_{m-1}\}$, es el conjunto de símbolos de entrada, llamado el alfabeto del autómata. Es importante que $\Sigma \cap Q \neq \emptyset$.
- $\delta : Q \times \Sigma \to Q$, es una función llamada función de transición; para cambiar del estado actual a otro [posiblemente el mismo] cuando ocurre un evento, representado por un símbolo del alfabeto.
- $q_0 \in Q$ el estado inicial.
- $A \subseteq Q$, con $A \neq \emptyset$ es el subconjunto de estados aceptores.

Los `AFD%` tienen tres tipos de estados:

1. El estado inicial, único en el `AFD%`, identificado como `q`$_0$.
2. Los estados aceptores identificado con `A`, es un subconjunto de estados `Q`. El estado inicial puede ser también aceptor.
3. Los estados comunes, son estados que no son los aceptores ni es el estado inicial.

Lo que determina el comportamiento del autómata es el conjunto de transiciones identificada por `T`. Cada regla de **transición** es una relación entre un par *estado-símbolo* y un estado del autómata. Las reglas de transicióon se coleccionan en un conjunto para que durante la operación del autómata se seleccione la regla adecuada que se debe aplicar de acuerdo a cada situación específica que se presente. Se observa que

```
(= (card T)  (card (pCart E  S))) ↦ #t
```

Esto es porque para cada par en (pCart E S), hay exactamente una transición en T.

■ **Ejemplo 6.1** Suponga una caja fuerte que tiene un cerradura que abre haciendo girar una perilla. La caja fuerte se encuentra inicialmente en una posición estratégica, que es identificada con el símbolo posI; la combinación que abre la caja, a partir de la posición inicial es una vuelta a la izquierda 'i, dos vueltas a la derecha 'd y una vuelta más a la izquierda, es decir que la caja fuerte se abre cuando se ha realizado la secuencia '(i d d i). Cualquier error en la combinación devolverá la perilla de la cerradura a su posición inicial.

Sea cf↤(list Q S T q_0 A) un AFD% definido por:

```
Q↤'(posI pos2 pos3 pos4 abie); los estados.
S↤'(i d); el alfabeto.
T↤'((posI i pos2) (posI d posI) (pos2 i posI)
(pos2 d pos3) (pos3 i posI) (pos3 d pos4)
(pos4 i abie) (pos4 d posI) (abie i posI)
(abie d posI)) ; las transiciones
q0↤'posI; el estado inicial.
A↤'(abie); estados aceptores.
```

■

6.3 Representaciones de los AFD%

La información de los AFD% puede representarse en diferentes maneras, una de ellas ya ha sido descrita, es su propia definición, donde se enlistan cada uno de los campos del AFD%; pero aún hay otras representaciones que son comunmente utilizadas. La representación en tablas de transiciones, la representación en grafos de transiciones y una representación en modo texto.

6.3.1 Tablas de transiciones

La tabla de transiciones es una manera de representar toda la información de un AFD% en un formato tabular donde:

1. Cada símbolo del alfabeto encabeza una columna.
2. Cada estado encabeza una fila.
3. Cada estado aceptor se marca con un punto [•].
4. El estado inicial se marca con un símbolo de mayor-que [>].
5. Si existe una triada '(q s q_d) en la lista de transiciones T del autómata, entonces en la tabla de transiciones la entrada ⟨q,s⟩ tiene un valor 'q_d.

■ **Ejemplo 6.2** Suponga el autómata de la caja fuerte descrita en el ejemplo 6.1, la tabla de transiciones del autómata finito determinista es

	'i	'd
>'posI	'pos2	'posI
'pos2	'posI	'pos3
'pos3	'posI	'pos4
'pos4	'abie	'posI
•'abie	'posI	'posI

■

En la tabla de transiciones se debe verificar que exista un único estado inicial, aunque pueden haber uno o más estados aceptores. En el ejemplo 6.2 solamente hay un sólo estado aceptor que es `'abie`, el estado inicial es `'posI`. El estado inicial también puede ser un estado aceptor, de modo que deben acumularse las marcas que corresponden, así si `q` es un estado aceptor que también es el estado inicial, se debe representar como >•`'q`, o bien como •>`'q`.

En la tabla de un autómata finito determinista es notoria la característica de que existe un único valor asociado a cada estado–símbolo, ya que se trata de una función (página 56). De modo que el número de transiciones en la tabla de transiciones coincide con el producto del número de estados con el número de símbolos en el alfabeto:

```
(= (card T) (* (card E) (card S))) ↦ #t
```

El determinismo del autómata queda de manifiesto al notar que dentro de la tabla no quedan espacios en blanco y hay solamente un estado que está asociado a un renglón y columna. Un espacio en blanco indica que no se ha definido la relación entre el *estado-símbolo* y el estado destino; por otro lado, si hubiera más de un estado en alguna entrada de la tabla, indica que el autómata al estar en un estado `q` y analizar un símbolo `s`, se tendrían más de un estado destino, lo que no es aceptable en los autómatas deterministas.

6.3.2 Grafos de transiciones

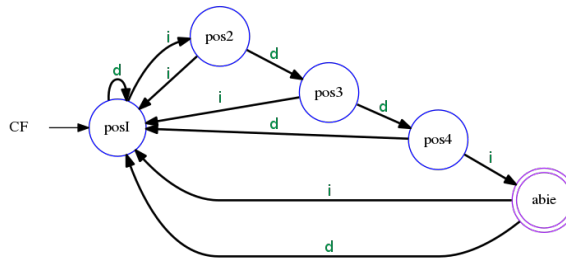
Otra representación de los `AFD%` se llama **grafo de transiciones**, esta representación es gráfica y permite un panorama general del comportamiento del autómata. Sin embargo, cuando el autómata es muy grande, utilizar grafos de transiciones puede resultar aún más confuso.

Los grafos de transiciones tienen los siguientes elementos:

1. Los **estados** se representan por medio de los nodos del grafo y se dibujan generalmente con círculos etiquetados con el identificador del estado. Las etiquetas suelen estar dentro de los círculos. Cuando hay demasiados estados o símbolos en el alfabeto, el grafo de transiciones suele ser confuso y se prefiere entonces utilizar la tabla de transiciones.
2. Cada par de estados se conectan mediante una arista dirigida (flecha) etiquetada con alguno de los símbolos del **alfabeto**.

3. Las **transiciones** se representan uniendo nodos mediante flechas. Si en las transiciones hay una regla '(q s q_d), se dibuja una flecha con etiqueta 's que inicia en el nodo etiquetado con 'q y con la punta de flecha en el nodo etiquetado con 'q_d.
4. El **estado inicial** se representa con un nodo que es señalado mediante una flecha y tiene un letrero que dice «Inicio», o bien con el identifiador del autómata.
5. Los estados aceptores se representan con nodos que se identifican mediante un doble círculo, etiquetado con el identificador del estado. El nodo inicial también puede ser un nodo aceptor.

■ **Ejemplo 6.3** La caja fuerte (CF) del ejemplo 6.1 se puede representar por el grafo:



■

Con la representación en forma de grafo de transiciones es posible revisar las transiciones de un solo vistazo; el grafo de transiciones es una herramienta que sirve para determinar visualmente el funcionamiento del autómata determinista. Si el autómata actualmente se encuentra en un estado marcado con q, el cálculo del siguiente estado del autómata a causa de la lectura del símbolo s, se logra al ubicar el estado q y seguir la arista etiquetada con s, de modo que el siguiente estado es precisamente la etiqueta del nodo al final de la arista.

La característica determinista del autómata, garantiza que para cada estado, siempre habrá una y sólo una flecha por cada símbolo del alfabeto.

6.3.3 Transiciones en formato texto

Una representación más para mostrar la información de un AFD% es en formato de texto plano. Este formato es útil para leer/escribir autómatas desde/hacia archivos de texto. El criterio propuesto se parece a una definición de grafos ponderados en forma de listas de adyacencias [BM76]. Supondremos un AFD% definido con la tupla ⟨E S T q_0 A⟩:

1. El formato consta de (card E) renglones, un renglón por cada estado.
2. Cada renglón inicia con un símbolo de tipo de estado, que puede ser uno de los siguientes '(>> >* *> ** --) donde:

- `'>>` Indica que se trata del estado inicial.
- `'>*` Indica el estado inicial que también pertenece a los estados aceptores.
- `'*>` Indica el estado inicial que también pertenece a los estados aceptores.
- `'**` Indica un estado aceptor, pero que no es el estado inicial.
- `'--` Indica un estado, que no es ni el estado inicial, ni un estado aceptor.
  - El renglón del estado inicial, puede comenzar con `'>>`, `'>*` o bien con `'*>`.
  - Cada renglón de un estado aceptor no inicial, empieza con `'**`.
  - Un renglón que no inicia con `'>*`, ni `'*>`, ni `'>>`, ni `'>>`, inicia con `'--`.

3. Después de la marca de inicio de renglón se escribe el símbolo de un estado; al que le siguen una lista de elementos que ocurren en pares *símbolo–estado*, para cada uno de los símbolos del alfabeto y sus correspondientes estados siguientes. Esto es, suponga que

   `(subc? '(`q_a q_b q_c`)  Q)` $\mapsto$ `#t`,
   `S` $\mapsto$ `'(`s_i s_j`)`,
   `(subc? '((`q_a s_i q_b`) (`q_a s_j q_c`))  T)` $\mapsto$ `#t`.

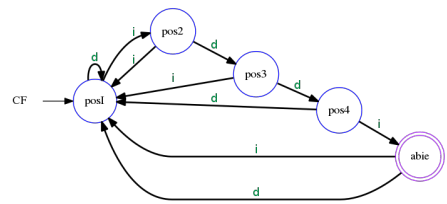
   Suponiendo además que `'`q_a es un estado regular (no es ni aceptor ni el estado inicial), el renglón que codifica la información referente al estado `'`q_a es:

   `--` q_a s_i q_b s_j q_c

   - En cada renglón se encuentran todos los símbolos del alfabeto del autómata determinista, porque hay transiciones para cada símbolo de entrada.
   - Es posible que no todos los estados tengan representación en un sólo renglón.

■ **Ejemplo 6.4** El autómata finito determinista definido en el ejemplo de la caja fuerte (página 111) se representa en formato de texto como:

```
>> posI i pos2 d posI
-- pos2 i posI d pos3
-- pos3 i posI d pos4
-- pos4 i abie d posI
** abie i posI d posI
```



■

6.4 Los AFD% como una clase

Un autómata finito determinístico (y todas las demás clases de autómatas, como se verá en capítulos posteriores) puede verse desde una perpectiva orientada a objetos como una clase que tiene ciertos atributos y ciertos métodos. Esta visión es conveniente para tratar a los AFD% como entidades autónomas, que es la visión fundamental en la teoría de sistemas multiagentes [NS03, YKSK05, LY08].

Cada AFD% es una instancia de una clase y reacciona ante un conjunto particular de eventos y haciendo un conjunto particular de acciones. En las siguientes secciones se detalla una clase llamada AFD%.

En las siguientes subsecciones se describirán los atributos de la clase AFD%, así como los métodos informativos.

En la teoría de objetos, los métodos informativos (*getters*) son aquellos que se utilizan para obtener la información actual de los atributos del objeto [Amb04].

Para crear una clase se requiere el nombre de una super clase y una o más expresiones o declaraciones [FFF06]:

```
(class expr-superclase decl-o-expr ...)
; expr-superclase : Nombre de superclase.
; decl-o-expr : declaraciones y/o expr.
```

Para efectos prácticos, se propone la siguiente plantilla para crear una clase anónima:

```
(class expr-superclase
  decl-init
  decl-camp ...
  (super-new)
  expr-met ... )
; decl-init : para inicializar campos
; decl-camp ... : para definir campos
; (super-new)
; expr-met ... : para definir métodos
```

Posteriormente, es aconsejable poner un nombre a la clase para ser referenciada en el momento deseado:

```
(define id class-expr)
; id : Un identificador.
; class-expr : Expresión que declara una clase.
```

6.4.1 Campos de los AFD%

La clase de autómatas finitos deterministas es definida con cinco **campos** [*fields*] principales. En la teoría de diseño y modelado orientado a objetos los campos se conocen con el nombre de «atributos». Cada campo coincide con cada elemento en la tupla $\langle \mathtt{E}\ \mathtt{S}\ \mathtt{T}\ \mathtt{q}_0\ \mathtt{A} \rangle$ que define un AFD% [ver página 109]:

1. Un conjunto finito y no vacío de estados `E`. Para la definición de los estados, cualquier lista de símbolos, preferentemente símbolos que guien la intuición del comportamiento del autómata.
2. Un alfabeto `S` que es un conjunto finito y no vacío de símbolos.
3. Un conjunto finito y no vacío de reglas de transición `T`, dada como una lista de tripletas `'((q s` q_d`) ...)`, cada tripleta declara un estado actual `q`, un símbolo del alfabeto de entrada `s` y un estado siguiente q_d.
4. Un estado inicial q_0, que es un elemento del conjunto de estados.
5. Un subconjunto no vacío de estados `A`, que se conocen como estados aceptores.

La clase inicializa el valor de sus campos con la quíntupla de la definición del autómata que se llamará `tuplaDef`, así:

```
tuplaDef↤(list Q S T q0 A)
```

Definición de campos en la clase ***AFD%***

```
(define AFD%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)] ; El estado inicial.
           [A  (list-ref tuplaDef 4)]) ; Los estados aceptores.
    (super-new)
     ; .... Métodos públicos y privados.
     ))
```

■ **Ejemplo 6.5** La caja fuerte del ejemplo 6.1 (ver página 111) se puede definir como sigue:

```
> (define cf (new AFD% [tuplaDef '((posI pos2 pos3 pos4 abie) (i d)
((posI i pos2) (posI d posI) (pos2 i posI) (pos2 d pos3) (pos3 i posI)
 (pos3 d pos4) (pos4 i abie) (pos4 d posI) (abie i posI) (abie d posI))
 posI '(abie))]))
> cf
(object:AFD% ...)
>
```

■

La expresión `(init tuplaDef)` en la definición de la clase `DrRacket`, quiere decir que para inicializar el valor de cada uno de los campos, se requiere una lista llamada `tuplaDef`.

La expresión `(super-new)` inicializa la superclase con los argumentos que se han especificado [FP16]. Esta expresión puede ubicarse en cualquier parte de la definición de la clase. En este documento se prefiere ubicarlo entre las declaraciones de campos y las expresiones de métodos.

6.4.2 Métodos informativos de los AFD%

Los primeros métodos implementados en la clase AFD%, son los métodos informativos, ya que será necesario saber con qué valores se ha definido el autómata determinista.

- edos. Devuelve la lista de los estados del AFD%.
- alfa. Devuelve la lista de símbolos del alfabeto de AFD%.
- tran. Devuelve una lista de transiciones del AFD%.
- eini. Devuelve el estado inicial definido para el AFD%.
- acep. Devuelve una lista con los estados aceptores definidos para el AFD%.

Metodos informativos en la clase AFD%

```
(define AFD%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa) S)
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
     ; .... Otros métodos públicos y privados.
     ))
```

La definición de métodos públicos se se hace con

```
(define/public (id atrib ...) expr ... )
; id : Un identificador.
; atrib ... : Atributos.
; expr ... : Expresiones.
```

Para el caso de los métodos privados, solamente hay que sustituir la palabra public por la palabra private. La diferencia es que los métodos privados no son accesibles sino por los métodos dentro de la clase.

La información de los campos en el AFD% se obtiene al enviar un mensaje al objeto de la clase AFD%. Un mensaje contiene el objeto destinatario, el nombre del método y la lista de atributos.

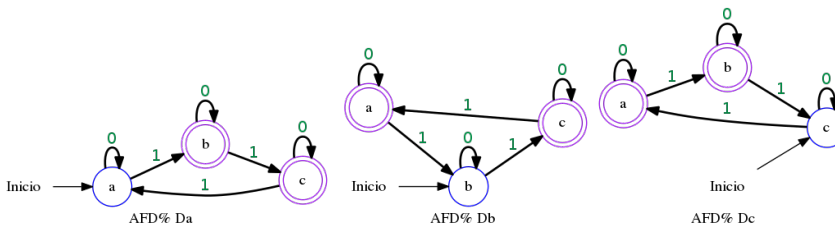
Los mensajes en DrRacket tienen la siguiente estructura:

```
(send obj-id met-id arg ...)
; obj-id : El id delel objeto receptor.
; met-id : El id del método en el objeto.
; arg ... : Los argumentos del método.
```

Una vez que se tiene la definición de la clase AFD% con sus campos y los métodos informativos, es posible crear diferentes autómatas de la misma clase, esto es crear diferentes instancias de la misma clase.

■ **Ejemplo 6.6** Se definen tres diferentes autómatas finitos deterministas que comparten el mismo conjunto de estados, de símbolos y transiciones; pero difieren en el estado inicial y el conjunto de estados aceptores.

```
> (define est '(a b c))
> (define alf '(0 1))
> (define tra '((a 0 a) (a 1 b) (b 0 b) (b 1 c)
   (c 0 c) (c 1 c)))
> (define Da (new AFD% [tuplaDef
     (list est alf tra 'a '(b c))]))
> (define Db (new AFD% [tuplaDef
     (list est alf tra 'b '(a c))]))
> (define Dc (new AFD% [tuplaDef
     (list est alf tra 'c '(a b))]))
> (send Da eini)
'a
> (send Db eini)
'b
> (send Dc eini)
'c
>
```



Cuando se crean diferentes instancias de una misma clase, como en este ejemplo, a pesar de que comparten la información, cada objeto tiene una copia de la información con la que se han definido. ■

6.5 Los cambios de estado en el AFD%

Las transiciones en un AFD%, se refieren al cambio de estado a causa de un evento, que es representado por la ocurrencia de un símbolo del alfabeto, o bien a causa de una secuencia de eventos, representada por una palabra de símbolos en el alfabeto.

En esta sección se tratan dos tipos de transiciones, la primera de ellas es la causada por la lectura de un sólo símbolo del alfabeto y la segunda es la transición ocasionada por una palabra.

6.5.1 Las transiciones del AFD%

El AFD% como un ente computacional con capacidades propias, debe cambiar de estado cuando se presenta un símbolo del alfabeto. La elección del siguiente estado está completamente determinada por la función de transiciones T, inscrita en la definición del autómata determinístico.

El conjunto de transiciones definido como el conjunto de tuplas T es solo información que requiere un método que pueda utilizarla adecuadamente. Para utilizar adecuadamente el conjunto de transiciones se necesita un método con la misión de determinar el estado que el autómata debe tener a causa de haber analizado un símbolo del alfabeto.

Si D1 es un AFD% con transiciones en T y (en? (q s q_d) T) $\mapsto$ #t y además el autómata se encuentra en el estado 'q, al presentarse el símbolo 's, el siguiente estado debe ser 'q_d.

Definiremos ahora la **función de transición** con el nombre Tr. La función de transición Tr es diferente que el conjunto T que determina la función de transición, porque T es un conjunto, mientras que Tr es un procedimiento que ocupa el conjunto T para determinar el siguiente estado del estado actual del AFD%.

El método Tr sirve para modelar la función de transición. Analiza el conjunto T de tuplas con el estado actual q y algún símbolo s, selecciona la regla adecuada de T, que es aquella que tiene como elemento del dominio al par (tupla q s) y devuelve el estado destino q_d que corresponda.

Así (Tr q s) $\mapsto$ E?, que se puede leer como «Al hacer la transición desde el estado q con un símbolo s, se alcanza un estado en E».

Lo que el método Tr debe hacer es evaluar la función de transición T, con los siguientes argumentos:

1. La tupla (tupla q s),
2. La función de transición T. Como T es un atributo, no es necesario especificarlo en los argumentos del método porque es accesible en la clase.

En la primera parte del libro se ha hecho una definición que hace el trabajo de la evaluación de una función, se trata de la definición evalf (página 58), que toma un elemento del dominio de la función y devuelve el único elemento del rango.

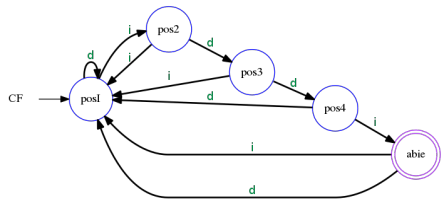
Aunque este método Tr debe ser privado, es buena idea establecerlo como público, mientras se observa su comportamiento. El método Tr debe ser insertado en la definición de la clase AFD%.

```
; (Tr q s) ↦ E?
; q : E? - un estado
; s : S? - un símbolo del alfabeto
(define/public (Tr q s)
   (evalf (tupla q s) T))
```

La definición de la clase AFD% es como se muestra:

```
(define AFD%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa) S)
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
    (define/public (Tr q s) (evalf (tupla q s) T))
    ))
```

■ **Ejemplo 6.7** Supongamos nuevamente el autómata determinista que modela la operación de la caja fuerte (página 111), que para comodidad se muestra nuevamente el grafo de transiciones (se debe cargar nuevamente el autómata como en el ejercicio 6.5).



Suponga que `cf` es el `AFD%` que define el grafo de transiciones presente y se encuentra en el estado inicial `'posI` y se recibe el símbolo `'i`, que indica que la perilla se ha movido una posición a la izquierda. El siguiente estado del autómata se obtiene al hacer la transición `(Tr 'posI 'i)`:

```
> (define cf (new AFD% [tuplaDef '((posI pos2 pos3 pos4 abie) (i d)
((posI i pos2) (posI d posI) (pos2 i posI) (pos2 d pos3) (pos3 i posI)
 (pos3 d pos4) (pos4 i abie) (pos4 d posI) (abie i posI) (abie d posI))
 posI '(abie))]))
> (send cf Tr 'posI 'i)
'pos2
>
```

■

6.5.2 Transición extendida del `AFD%`

Consideremos el autómata determinista $\mathtt{D} \mapsto \langle \mathtt{Q\ S\ T\ q_0\ A} \rangle$. Definiremos un nuevo método que extiende la idea de la función de transición, que se llamará

función de transición extendida y denotaremos por Tr*. Este método será útil para determinar el último estado que alcanza el autómata después de analizar una secuencia finita de símbolos del alfabeto. En otras palabras, se obtiene el estado final después de analizar una palabra.

La función (pCart (Kleene* S) E) proporciona el dominio la función de transición extendida y codominio el conjunto de estados E, así:

$$\texttt{(Tr* q w)} \mapsto \texttt{Estado?}$$

donde:

q: Es un estado en E.

w: Es una palabra en (Kleene* S).

La función de transición extendida queda definida por:

$$\texttt{(Tr* q w)} \mapsto \begin{cases} \texttt{q}, \text{ cuando } \texttt{w} \mapsto \texttt{pVacia}. \\ \texttt{(Tr* (Tr q (car w)) (cdr w))}, \\ \text{cuando } \texttt{w} \mapsto \texttt{(}w_0|\texttt{w')} \end{cases}$$

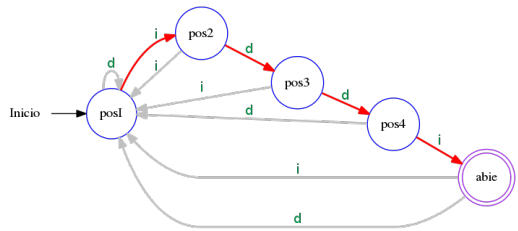
*Clase AFD% con el método Tr**

```
(define AFD%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa) S)
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)

    (define/public (Tr q s) (evalf (tupla q s) T))
    (define/public (Tr* q w) ;<-- Transición extendida
      (if (pVacia? w)
          q
          (Tr* (Tr q (car w)) (cdr w))))))
```

■ **Ejemplo 6.8** En el ejemplo 6.1 (página 111), si el autómata esta en 'posI y se gira la perilla a la izquierda, dos veces a la derecha y nuevamente a la izquierda, ¿en qué estado se encontrará la caja fuerte?

```
> (send cf Tr* 'posI '(i d d i))
'abie
>
```

La siguiente imagen muestra el recorrido desde el estado inicial `'posI`, hasta el estado final que se alcanza después de analizar la palabra `'(i d d i)`.



■

6.6 El lenguaje del AFD%

Hay otros métodos que hacen de los `AFD%` realmente útiles; son los métodos que permiten analizar secuencias de eventos. Las secuencias de eventos son codificadas mediante palabras de símbolos; se requiere analizarlas para saber qué palabras le permiten alcanzar alguno de sus estados aceptores, lo que significaría algún beneficio. Se asumirá `D`↤⟨`E S T` q_0 `A`⟩ un `AFD%` y `S*`↤`(Kleene* S)` [ver página 100].

6.6.1 Palabras aceptadas

Al interactuar con el autómata `D` utilizando palabras en `S*`, se crea una bi-partición del espacio `S*`, generando dos lenguajes: las palabras aceptadas y las palabras no aceptadas.

Se dice que un autómata determinista `D` **acepta** una palabra `w`, si a partir del estado inicial q_0 y después de analizar cada uno de los símbolos en `w`, el estado final pertenece al conjunto de estados aceptores definido para el `AFD%`. En otras palabras el estado resultante de la transición extendida a partir del estado inicial q_0 con la palabra `w`, pertenece al conjunto de estados aceptores `A`. La definición textual anterior permite la definición formal en `DrRacket`:

```
; (acepta? w) ↦ booleano?
; w : lista?
(define/public (acepta? w) (en? (Tr* q0 w) A))
```

Si después de analizar una palabra `w` a partir del estado incial q_0, el autómata alcanza un estado que no pertenece al conjunto de estados aceptores, entonces el autómata determinista `D`, **no acepta** la palabra `w`.

Clase AFD% con el método acepta?

```
(define AFD%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa) S)
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)

    (define/public (Tr q s) (evalf (tupla q s) T))
    (define/public (Tr* q w)
      (if (pVacia? w)
          q
          (Tr* (Tr q (car w)) (cdr w))))
    (define/public (acepta? w) (en? (Tr* q0 w) A)); <---
    ))
```

■ **Ejemplo 6.9** En el ejemplo de la caja fuerte, la palabra `'(i d d i)` no es la única que puede abrir la caja, otras secuencias como `'(d i d d i)` y `'(i i i d d i)` también son aceptadas, entre muchas otras.

```
> (send cf acepta?  '(d i d d i))
#t
> (send cf acepta?  '(i i i d d i))
#t
> (send cf acepta?  '(i d d i d d d i d d i))
#t
> (send cf acepta?  '(i d d i i i d i i d d i))
#t
>
```

■

6.6.2 El lenguaje finito del AFD%

El **lenguaje del** AFD% es conjunto de palabras aceptadas. Se denotará Leng al lenguaje del autómata determinista D. Se observa que para toda palabra w de S* se cumple que:

(<-> (acepta? w) (en? w Leng)).

☞ En la notación convencional se puede escribir:

$$\forall w \in \Sigma^* : acepta(w) \Leftrightarrow w \in Leng(D),$$

considerando que $D = \langle Q, \Sigma, \delta, q_0, A \rangle$ es un $afd\%$.

Así predicado `(enLeng? w)`, será `#t`si la palabra `w` en `S*` es aceptada por el autómata determinista `D` y que sea `#f` en otro caso:

```
; (enLeng? w) ↦ boolean?
; w : lista?
(define/public (enLeng? w) (acepta? w))
```

Con frecuencia el lenguaje del autómata es infinito. Esto ocurre en autómatas que incluyen transiciones cíclicas.

Aún cuando el lenguaje del autómata sea de cardinalidad infinta, es posible calcular un acercamiento al lenguaje, calculando todas las palabras aceptadas que tengan una longitud menor o igual a cierto umbral `n`. Este método se puede llamar el **lenguaje finito** del autómata, identificado por `nLeng` y recibe un sólo argumento, el número entero no negativo `n` que determina la longitud máxima de palabra que debe ser analizada.

Es claro entonces que `(<= (card nLeng) (card Leng))` ↦ `#t`.

Así el lenguaje `nLeng` se define como todas las palabras aceptadas de la cerradura finita de Kleene de hasta longitud `n`. Esto define formalmente la siguiente definición.

```
; (nleng n) ↦ (listade lista?)
; n : número-entero-no-negativo? = 10
(define/public (nLeng [n 10])
  (filter (λ (w) (acepta? w)) (nKleene* S n)))
```

■ **Ejemplo 6.10** Calcule en lenguaje finito hasta tamaño `4` del autómata determinista `Da`, definido en el ejemplo 6.6 (página 118)

```
> (send Da nLeng 4)
'((1) (0 1) (1 0) (1 1) (0 0 1) (0 1 0) (0 1 1) (1 0 0) (1 0 1)
(1 1 0) (0 0 0 1) (0 0 1 0) (0 0 1 1) (0 1 0 0) (0 1 0 1)
(0 1 1 0) (1 0 0 0) (1 0 0 1) (1 0 1 0) (1 1 0 0) (1 1 1 1))
>
```

■

■ **Ejemplo 6.11** ¿Cuántas palabras de hasta 5 letras acepta el autómata `Da` (página 118)?

```
> (define DaL5 (send Da nLeng 5))
> (card DaL5)
42
>
```

■

Se observa que si q es algún estado en Q; v ↦ '(v1 ... vk) es alguna palabra en S* y w ↦ '(w1 ... wm) es cualquier palabra en Leng, entonces es cierto que:

```
(paraTodo (λ (v)
   (paraTodo (λ (w)
      (-> (equal? (Tr* q v) q0)
      (enLeng? (concat v w))))
      Leng))
   S*)
```

☞ En notación convencional, la expresión anterior se puede traducir como:

$$\forall v \in \Sigma^* : [\forall w \in L_D : \delta^*(q,v) = q_0 \Rightarrow v \cdot w \in L_D]$$

donde:

- D: Es un autómata finito determinista $D = \langle Q, \Sigma, \delta, q_0, A \rangle$.
- Σ^*: Es la cerradura de Kleene para el alfabeto Σ.
- L_D: Es el lenguaje del autómata D, el conjunto de palabras aceptadas. $L_D \subseteq \Sigma^*$.
- $\delta^*(q,v)$: Es la función de transición extendida, evaluada desde un estado $q \in Q$ y una palabra $v \in \Sigma^*$.
- $v \cdot w$: Es la concatenación de v con w.

Suponga que a pesar de las condiciones de las palabras v y w (figura 6.1), la concatenación de ellas (concat v w) genera una palabra que no pertenece al lenguaje del autómata. Lo que significa en primer lugar, que el estado en que se encuentra D después de haber analizado v es q_0. Y en segundo lugar, que el estado final (Tr* q_0 w) no pertenece al conjunto de estados aceptores, esto porque la palabra vw no es aceptada; pero esto es una contradicción, porque se había establecido que w pertenece a Leng.

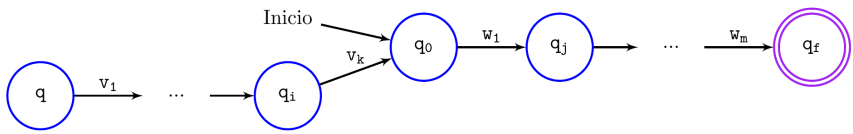
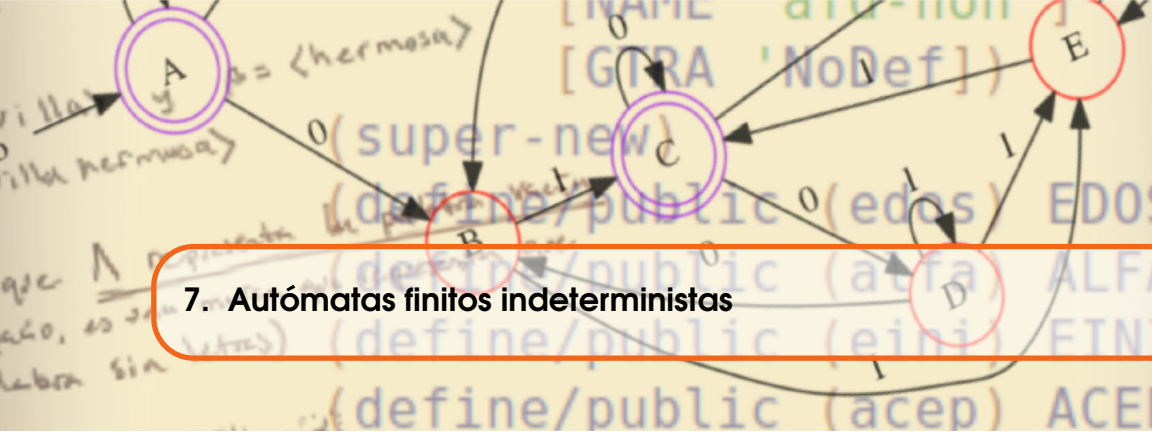


Figura 6.1: *Situación que muestra que la concatenación de cualquier palabra* $v \mapsto$ '(v_1 ... v_k) *que hace que el autómata termine en el estado inicial* q_0*, a partir del cual se analiza la palabra* $w \mapsto$ '(w_1 ... w_m) *que pertenece al lenguaje* Leng.

Código 6.1: *Modelo orientado a objetos de un* `AFD%`

```
(define AFD%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa) S)
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
    (define/public (Tr q s) (evalf (tupla q s) T))
    (define/public (Tr* q w)
      (if (pVacia? w)
          q
          (Tr* (Tr q (car w)) (cdr w))))
    (define/public (acepta? w) (en? (Tr* q0 w) A))
    (define/public (enLeng? w) (acepta? w))
    (define/public (nLeng [n 10])
      (if (< n 10)
        (filter (λ (w) (acepta? w)) (nKleene* S n))
        (append (filter (λ (w) (acepta? w)) (nKleene* S 10)) (list '...))))
    ))
```



7. Autómatas finitos indeterministas

En ocasiones puede resultar más difícil el diseño de un autómata finito determinista, especialmente cuando se debe asignar un estado para cada situación posible de estado–símbolo. O bien porque en realidad no se conocen o no se han determinado todos los eventos que hacen reaccionar al autómata; sin embargo, la condición determinística exige que debe haber un estado siguiente para todos y cada uno de los posibles pares estado–símbolo.

En otras ocasiones se puede observar que en realidad para cierto estado y cierto símbolo, el autómata finito debería estar en más de un estado.

■ **Ejemplo 7.1** Los siguientes son ejemplos que pueden ser modelados mediante autómatas finitos indeterministas:

- Una grúa al estar en *reposo*, luego de ser accionada por cierto mecanismo, podría estar *levantando* la carga y *girando* en su eje al mismo tiempo.
- Un radio personal de comunicación de dos vías simultáneas. Cuando el radio se encuentra en *espera* y es accionado un botón, el radio cambia al estado *escuchando*, pero también al estado *transmitiendo*.

■

Un autómata finito indeterminista (`AFN%`) es una máquina teórica que analiza secuencias finitas de eventos, que son representados por símbolos.

Se han definido una cantidad finita de estados, en los que puede ser observado el AFN%. Una vez que la palabra se ha terminado de analizar, el AFN% es capaz de decidir si esa secuencia de entrada es aceptada o no lo es.

Una diferencia importante con respecto a los AFD% es que un AFN% puede, a partir de un estado y un evento reconocible, alcanzar uno o más estados al mismo tiempo, incluso podría no estar definido algún estado para tal situación.

En este capítulo se analizará una manera de definir los AFN% que es orientada a objetos, creando una clase para definir este tipo de autómatas. También se definirá la manera en que cambia de estados.

7.1 Definición formal de un AFN%

La clase de autómatas finitos no deterministas AFD% se define por medio de una tupla:

$$\texttt{AFN\%} \mapsto \langle \texttt{Q S T } q_0 \texttt{ A} \rangle,$$

donde todos los elementos son como en el caso determinista, excepto por el conjunto de reglas de transición:

- ⋆ Q↤(q1|Q'): un conjunto finito y no vacío de símbolos que representan los estados del autómata.
- ⋆ S↤(q1|S'): Un conjunto finito de símbolos que representan los eventos que puede reconocer el autómata, este conjunto es llamado el **alfabeto** del AFN%.
- ⋆ T↤(t1|T'): Una **relación** T:Q×S→Q, es una lista de tuplas '((q s q_d) ...).
- ⋆ q_0: Un único estado distinguido de entre los estados. Es el **estado inicial**, seleccionado arbitrariamente entre los estados de Q de acuerdo a las necesidades del modelo.
- ⋆ A↤(a1|A'): Es el conjunto no vacío de estados **aceptores** que permitirá hacer la clasificación de las palabras, así (subc? A Q) ↦ #t

■ **Ejemplo 7.2** La siguiente definición representa un radio de comunicaciones de dos vías, donde se busca alcanzar estados de comunicación efectiva.

```
⋆ Q↤'(r1 r2 r3 r4);
⋆ S↤'(enc bot lib);
⋆ q0↤'r1;
⋆ T↤'((r1 enc r2) (r2 enc r1) (r2 bot r3)
  (r2 bot r4) (r3 bot r3) (r3 lib r2)
  (r4 bot r4) (r4 lib r2));
⋆ A↤'(r3 r4).
```

En este ejemplo, el alfabeto está compuesto por símbolos que representan tres eventos: 'enc, que indica que se ha oprimido el interruptor de encendido/apagado;

'bot, indica que se ha oprimido el botón para transmitir; y 'lib indica que se ha liberado el botón para transmitir.

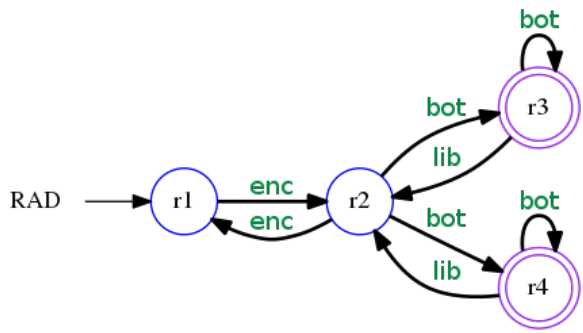
El estado 'r1, indica que el autómata se encuentra apagado. El estado 'r2 muestra al autómata encendido y en espera. El estado 'r3 significa que el radio se encuentra transmitiendo y el estado 'r4 indica que el radio de encuentra recibiendo. ∎

7.1.1 El grafo de transiciones de un AFN%

Las reglas de construcción de un grafo de transiciones para un AFN%, son las mismas que las que se han mencionado para el caso de un AFD% (ver página 112). Sin embargo se observan dos características que pueden determinar la clase de autómata que se crea. Si ocurre alguna de las siguientes situaciones, entonces el grafo representa un AFN%:

1. Debido a que el grafo de transiciones representa una relación entre estados, a partir de cada estado pueden salir una o más transiciones (flecha) etiquetada con un mismo símbolo s.
2. Es posible que a partir de un estado, no exista transición para algún símbolo s en particular.

∎ **Ejemplo 7.3** El grafo de transiciones que se muestra, corresponde al radio de comunicaciones de dos vías simultáneas definido en la página 128.



Nótese lo siguiente:

1. A partir del estado 'r2, hay transiciones hacia 'r3 y hacia 'r4 con el mísmo símbolo 'bot.
2. A partir del estado inicial 'r1, no existen transiciones ni con el símbolo 'bot, ni con 'lib.

∎

7.1.2 La tabla de transiciones de un AFN%

La tabla de transiciones para un AFN% presenta muy pocas diferencias respecto a la tabla de transiciones para un AFD%, de hecho la única diferencia es la manera

en que se muestran los estados siguientes en las transiciones.

Como `T` es una relación, es posible que una transición ocasione que el autómata se muestre en 0, 1 o más estados, por lo que los estados siguientes se representan con un conjunto [en forma de lista] de estados.

■ **Ejemplo 7.4** Suponga el autómata del ejemplo 7.2 de la página 128. La tabla de transiciones para este autómata es la siguiente:

	`'enc`	`'bot`	`'lib`
>`'r1`	`'(r2)`	`'()`	`'()`
`'r2`	`'(r1)`	`'(r3 r4)`	`'()`
•`'r3`	`'()`	`'(r3)`	`'(r2)`
•`'r4`	`'()`	`'(r4)`	`'(r2)`

Note que los estados siguientes han sido agrupados en una lista, para poder especificar que a partir de un estado y un símbolo, el autómata hace una transición a todos los estados que se indican en el conjunto de estados, que puede incluso ser vacío. ■

El estado inicial se identifica con un símbolo />/ y los estados aceptores con un punto /•/, si un estado es al mismo tiempo el estado inicial y pertenece al conjunto de los estados aceptores, entonces se marca con ambos símbolos. Los símbolos son arbitrarios, pero deben mantenerse a fin de no provocar confusiones.

El orden en que se han dispuesto los estados en la tabla de transiciones, en la columnda del extremo izquierdo, es irrelevante, esto porque los estados perenecen a un conjunto y en los conjuntos el orden de sus elementos es indistinto.

De manera similar ocurre con las columnas que representan las transiciones para cada uno de los símbolos de entrada. Sin embargo, se debe seleccionar un orden y respetar el mismo orden en cada una de las entradas de la tabla.

7.1.3 `AFN%` como listas de transiciones

El formato de un archivo de texto es muy similar al que se utiliza para leer autómatas `AFD%` (ver página 113), la diferencia es que en los `AFN%`, es posible que a partir de un estado `q` y un símbolo `s`, se alcance algún subconjunto de estados incluyendo posiblemente el conjunto `vacio`; a diferencia del caso `AFD%`, donde en cada transición se alcanza exactamente un estado.

Suponga $\texttt{N} \leftarrow\!\shortmid \langle \texttt{Q S T } q_0 \texttt{ A} \rangle$ es un `AFN%`, donde:

- `(subc? '(`s_a s_b`) S)` $\mapsto$ `#t`, y
- `(subc? '(`q_{i_1} … q_{i_n}`) Q)` $\mapsto$ `#t` y no hay restricción sobre la pertenencia de q_0 al conjunto `'(`q_{i_1} … q_{i_n}`)`.

También suponga que a partir del estado `q` y al analizar un símbolo s_a, se alcanzan los estados `'(`q_{i_1} … q_{i_n}`)`, que de hecho representa un subconjunto de estados. Para representar esta situación en la forma de una lista de transiciones, se utiliza la siguiente codificación:

q ... s_a q_{i_1} ... q_{i_n} s_b ...

Los primeros dos símbolos /##/ representan alguno de los símbolos que identifican el tipo de estado, que puede ser alguno de />>/ si es el estado inicial; /**/ si es un estado aceptor; /--/ si es un estado que no es ni un estado aceptor, ni el estado inicial; o bien /*>/ o />*/ si es el estado inicial que además es aceptor.

El siguiente símbolo debe ser un estado e_i que representa al estado actual q; los siguientes símbolos están escritos con el siguiente patrón: un símbolo del alfabeto s_a, seguido de sus estados alcanzables q_{i_1} ... q_{i_n}.

■ **Ejemplo 7.5** El autómata AFN% cuyo grafo de transiciones se muestra en la figura del ejemplo 7.3, se representa en formato de listas de transiciones como:

```
>> r1 enc r2 bot lib
-- r2 enc r1 bot r3 r4 lib
** r3 enc bot r3 lib r2
** r4 enc bot r4 lib r2
```

Los estados son '(r1 r2 r3 r4), donde 'r1 es el estado inicial y el subconjunto de estados aceptores es '(r3 r4). El autómata no determinista cuando se encuentra en el estado 'r2 (segundo renglón) y analiza el símbolo 'bot, alcanza los estados 'r3 y 'r4, pero si recibe el símbolo 'lib, el subconjunto de estados alcanzables es vacío. ■

El formato en «solo texto» es especialmente útil a la hora de obtener la configuración de un AFN% desde un archivo de texto. Pueden hacerse algunas observaciones en el procedimiento para reconocer cada elemento del autómata desde un archivo de texto:

1. Cada renglón tiene como segundo elemento un estado diferente.
2. En cada lista aparecen todos los símbolos del alfabeto, una vez que se tienen los estados y ya sabiendo los símbolos para hacer diferencia entre tipos de estado, el alfabeto queda determinado por la diferencia entre los símbolos que aparecen en el primer renglón [como cualquier otro] pero que no aparecen en la unión de los estados con los tipos de estados.
3. Las transiciones se obtienen considerando solamente aquellas que sí tienen estado o estados siguientes, por ejemplo el siguiente renglón:

   ```
   -- r2 enc r1 bot r3 r4 lib
   ```

   se convierte en '(-- r2 enc r1 bot r3 r4 lib), a partir de la cual se obtienen las transiciones '(r2 enc r1), '(r2 bot r3) y '(r2 bot r4); ya que con el símbolo 'lib no se alcanza ningún estado.
4. El estado inicial es aquel elemento del conjunto de estados, cuyo renglón inicia con algún símbolo en '(>> >* *>), puesto que estos símbolos tienen una marca />/.
5. Los símbolos del conjunto de estados aceptores se pueden conocer al filtrar todos aquellos estados, cuyo renglón inicia con algún símbolo en '(** >* *>), porque tienen un símbolo /*/.

7.2 Los AFN% como clase

De manera similar a los AFD%, en términos de diseño orientado a objetos, se crea una clase AFN%, con atributos y métodos específicos, de modo que se puedan crear tantos objetos como se desee y que puedan interactuar entre ellos como entidades independientes.

La ventaja de utilizar un enfoque orientado a objetos, es la posibilidad de definir entidades abstractas independientes, donde cada objeto contenga sus propias definiciones de atributos y métodos. En el enfoque imperativo, que es más convencional, se comparten los accesos a las definiciones de los métodos.

Los atributos básicos de la clase AFN% se pueden conocer por los elementos de la tupla de definición: ⟨Q S T q_0 A⟩, donde todos los elementos son como en los AFD%, excepto el conjunto de reglas de transición T que ahora es una relación con dominio en (pCart Q S) y codominio en Q, a diferencia del caso AFN% que se exige que T defina una función.

Definición de campos en la clase AFN%

```
(define AFN%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    ; ... Métodos públicos y privados
    ))
```

Q: Es un conjunto finito y no vacío de símbolos que representan los estados del AFN%, en este libro usualmente los denotamos con una letra q inicial y luego un número a manera de índice, como en '(q0 q1 q2 q3), pero esto no es una regla.

S: Es un conjunto finito y no vacío de símbolos de entrada, es el alfabeto con el que se define el AFN%. Usualmente se usan las letras o los números, como '(0 1), '(a b c) o símbolos mnemónicos como el caso del ejemplo '(enc bot lib). El uso de mnemónicos ayuda a relacionar el símbolo con un significado [KKR05].

T: Es la relación de transición de estados, es una asociación pares estado-símbolo de entrada, con un subconjunto de estados.

q_0: Es uno de los estados, esto es que (en? q_0 Q) ↦ #t.

A: Es un subconjunto de estados. Son los estados aceptores. También (subC? A Q) ↦ #t.

■ **Ejemplo 7.6** La siguiente interacción muestra la definición de un AFN% que tiene las características requeridas en el ejemplo 7.2 (ver página 128): Se crearán

defininiciones para los estados, para el alfabeto, la función de transiciones, el estado inicial y los estados aceptores.

```
> (define EST '(r1 r2 r3 r4))
> (define ALF '(enc bot lib))
> (define TRA '((r1 enc r2) (r2 enc r1) (r2 bot r3)
(r2 bot r4) (r3 lib r2) (r4 lib r2) (r3 bot r3)
(r4 bot r4)))
> (define EIN 'r1)
> (define ACE '(r3 r4))
> (define RAD (new AFN%
       [tuplaDef (list EST ALF TRA EIN ACE)]))
> RAD
(object:AFN% ...)
```

Enlistar elementos mediante `list`, además de crear una lista, los elementos no pierden sus propiedades, es decir la lista no es una lista de símbolos sin significado, sino una lista de estados, un alfabeto, etcétera. ■

7.2.1 Los métodos informativos

Los métodos llamados «informativos» se llaman así porque recuperan la información establecida en cada uno de los atributos. La visivilidad de estos métodos es pública, pues su finalidad es conocer los atributos del autómata. Los métodos definidos anteriormente para el caso AFD% sirven perfectamente (definidos a partir de la página 117).

Metodos informativos en la clase AFN%

```
(define AFN%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa) S)
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
     ; .... Otros métodos púublicos y privados
    ))
```

■ **Ejemplo 7.7** Una vez que se han agregado las definiciones de los métodos informativos a la definición del AFN% del ejemplo 7.6, se puden hacer las siguientes interacciones para utilizar los métodos informativos:

```
> (send RAD edos)
'(r1 r2 r3 r4)
> (send RAD alfa)
'(enc bot lib)
> (send RAD tran)
'((r1 enc r2) (r2 enc r1) (r2 bot r3) (r2 bot r4)
  (r3 bot r3) (r3 lib r2) (r4 bot r4) (r4 lib r2))
> (send RAD eini)
'r1
> (send RAD acep)
'(r3 r4)
>
```

■

7.3 Los cambios de estado en el AFN%

La manera en que el AFN% cambia de un estado a otro, fundamentalmente es la misma en que lo hacen los AFN%, el cambio de un estado a otro ocurre debido al análisis de un evento reconocible por el autómata. La diferencia ahora es que el autómata puede cambiar a más de un estado, o bien a ninguno.

7.3.1 Las transiciones del AFN%

El método de transición para los AFN% permite que el autómata pueda cambiar de estado actual y ser observado en cero, uno o más estados al mismo tiempo. Considerese como ilustración el autómata etiquetado como N1 de la figura 7.1:

	0	1
g1	'(g4)	'()
g2	'(g4)	'(g3 g5)
g3	'(g4)	'(g6)
g4	'(g5 g7)	'(g6)
g5	'(g5)	'(g5)
*g6	'()	'()
*g7	'()	'()
>g8	'(g2)	'(g1 g3)

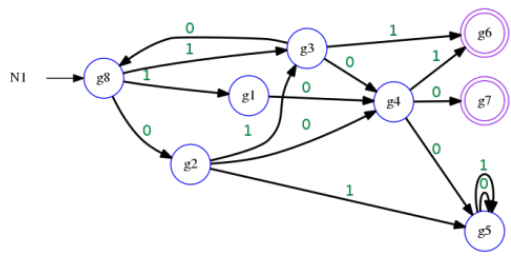


Figura 7.1: *Autómata finito indeterminista N1, definido con los estados '(g1 g2 g3 g4 g5 g6 g7 g8); con alfabeto '(0 1); su estado inicial es 'g8 y aceptores '(g6 g7); las transiciones se pueden deducir a partir de la tabla de transiciones a la izquierda.*

Para obtener los estados siguientes en una transición, el autómata indeterminista requiere su estado actual, que se puede identificar con q y un símbolo del alfabeto, que es representado por s. El conjunto de transiciones T está conformado por tuplas de la forma '(q s q_d), donde los primeros dos términos representan

el estado actual y el símbolo del alfabeto y ambos forman una tupla que pertenece a (pCart Q S). Se requiere enconces calcular la imagen de la tupla '(q s) en la relación T (ver la página 52).

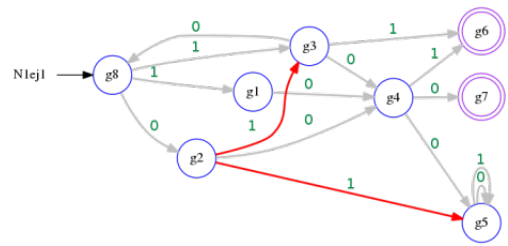
☞ En símbolos convencionales, se puede definir la transición de estados de un AFN% como:

$$\delta(q, \omega) = Q'$$

Suponiendo que $\delta : Q \times \Sigma \to \mathbb{P}(Q)$ es una función con rango en el conjunto potencia del conjunto de estados; $q \in Q$ y $\omega \in \Sigma^*$, mientras que $Q' \subseteq Q$.

■ **Ejemplo 7.8** Si el autómata N1 (figura 7.1) se encuentra actualmente en el estado 'g2 y analiza el símbolo 1, el autómata reacciona y cambia de estado; lo característico en los autómatas indeterministas es que ahora puede cambiar a más de un estado. Una vez hecha la transición, el autómata N1 ahora se encontrará simultáneamente tanto en el estado 'g3 como en el estado 'g5; esto es debido a que en el conjunto de transiciones T se encuentran las tuplas '(g2 1 g3) y '(g2 1 g5), por lo que la función de transición debe asociar el par de entrada '(g2 1) con el subconjunto '(g3 g5). En la figura siguiente se han resaltado las transiciones en cuestión.

	0	1
g1	'(g4)	'()
g2	'(g4)	'(g3 g5)
g3	'(g4)	'(g6)
g4	'(g5 g7)	'(g6)
g5	'(g5)	'(g5)
*g6	'()	'()
*g7	'()	'()
>g8	'(g2)	'(g1 g3)



La tabla de transiciones muestra el subconjunto asociado al par de elementos de entrada. En los renglones se encuentran los estados y en las columnas se encuentran los símbolos del alfabeto. En la tabla de la figura se ha resaltado en renglón que corresponde al estado 'g2 y la columna que corresponde al símbolo 1; en la casilla donde se intersectan renglón y columna, se encuentra el subconjunto asociado. ■

Cuando en las transiciones no hay alguna a partir de un estado q con un símbolo s, como en el caso del par '(g1 1), el subconjunto de estados asociado es simplemente el conjunto vacio.

La función de transición en la clase AFN%

```
1 (define AFN%
2   (class object%
3     (init tuplaDef)
4     (field [Q  (list-ref tuplaDef 0)]
5            [S  (list-ref tuplaDef 1)]
```

```
6           [T  (list-ref tuplaDef 2)]
7           [q0 (list-ref tuplaDef 3)]
8           [A  (list-ref tuplaDef 4)])
9     (super-new)
10    ;----------
11    (define/public (edos) Q)
12    (define/public (alfa) S)
13    (define/public (tran) T)
14    (define/public (eini) q0)
15    (define/public (acep) A)
16    (define/public (Tr q s) (Im (tupla q s) T))
17     ; .... Otros métodos púublicos y privados
18     ))
```

7.3.2 La transición extendida

La transición extendida es la consecusión de cambios de estados del autómata, a causa de la lectura de una secuencia de símbolos. Suponiendo que `N` es un `AFN%`, cuando `N` se encuentra en un estado inicial `q`, debe analizar una palabra `w` en `S*↤(Kleene S)`, donde o bien `w` ↦ `pVacia`, o bien `w` ↦ `(w1|w')`. El objetivo ahora es determinar qué estados alcanzará el autómata indeterminista después de analizar dicha palabra.

El método de **transición extendida** tendrá el identificador `Tr*`, el cual requiere como argumentos de entrada un estado `q` donde `(en? q E)` ↦ `#t` y una palabra `w` en `S*`. El método `Tr*` debe evaluar la entrada y producir un conjunto de estados `'(`q_d `...)`, todos ellos estados en `Q`.

Cuando `w` ↦ `pVacia`, `(Tr* q w)`↦ `(conj q)`, el conjunto unitario que contiene a `q`.

Por otro lado, si `w` ↦ `(w1|w')`, es necesario hacer una transición a causa de `w1` y recursar el procedimiento `Tr*` a partir de cada nuevo estado y con `w'`, el resto de la palabra, lo que genera una familia de subconjuntos, uno por cada transición hecha a partir de uno de los estados en `(Tr q w1)`. Finalmente se debe obtener la unión de todos los subconjuntos para lograr el conjunto de estados siguientes:

```
(apply union* (map (λ (e) (Tr* e (cdr w)))
                   (Tr q (car w)))).
```

$$
\texttt{(Tr* q w)} \mapsto \begin{cases} \texttt{(conj q)}, \text{ cuando } \texttt{w} \mapsto \texttt{pVacia}. \\ \\ \texttt{(apply union*} \\ \quad \texttt{(map (λ (e) (Tr* e (cdr w))} \\ \quad\quad \texttt{(Tr q (car w)))))}, \\ \text{cuando } \texttt{w} \mapsto \texttt{(w1|w')} \end{cases}
$$

☞ De manera equivalente, en un lenguaje más apropiado en matemáticas, se puede expresar la transición extendida como:

$$\delta^*(q,\omega) = \begin{cases} \{q\}, \text{cuando } \omega = \langle\rangle; \\ \bigcup_{p\in\delta(\langle q,\omega_1\rangle)} \delta^*(p,\omega'), \text{cuando } \omega = \langle\omega_1|\omega'\rangle, \end{cases}$$

con $q \in Q$ y $\omega \in \Sigma^*$.

```
; (Tr* q w) ↦ (listade Q?)
; q : Q? - es un estado
; w : S*? - es una palabra
(define/public (Tr* q w)
      (if (pVacia? w)
          (tupla q)
          (apply union* (map (λ (e) (Tr* e (cdr w))) (Tr q (car w))))))
```

Así en la clase AFN% se agrega este nuevo método:

```
1 (define AFN%
2   (class object%
3     (init tuplaDef)
4     (field [Q  (list-ref tuplaDef 0)]
5            [S  (list-ref tuplaDef 1)]
6            [T  (list-ref tuplaDef 2)]
7            [q0 (list-ref tuplaDef 3)]
8            [A  (list-ref tuplaDef 4)])
9     (super-new)
10    ;----------
11    (define/public (edos) Q)
12    (define/public (alfa) S)
13    (define/public (tran) T)
14    (define/public (eini) q0)
15    (define/public (acep) A)
16    (define/public (Tr q s) (Im (tupla q s) T))
17    (define/public (Tr* q w) ; <- transición extendida
18      (if (pVacia? w)
19          (tupla q)
20          (apply union* (map (λ (e) (Tr* e (cdr w))) (Tr q (car w))))))
21    ;....
22    ))
```

■ **Ejemplo 7.9** Dado el autómata indeterminista N1, definido en la página 134, calcular (Tr* 'g8 '(0 1 1)).

Con el autómata de la figura N1 se calcula (Tr* 'q8 '(0 1 1)):

1. Como la palabra no es vacía, se aplicará la union* a los conjuntos que resulten de la transición (Tr 'g8 0), esto produce el conjunto '(g2) y el resto de la palabra es '(1 1).
2. Ahora es necesario hacer nuevamente la transición extendida Tr* a partir de cada estado del conjunto obtenido y con el resto de la palabra:

a) Hacer `(Tr* 'g2 '(1 1))`, porque el conjunto obtenido fue `'(g2)`, de lo que se obtiene `'(g3 g5)` y el resto de la palabra ahora es `'(1)`.

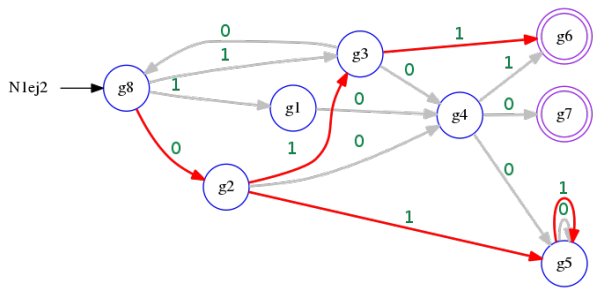
3. Para cada estado `e` en `'(g3 g5)` hacer `(Tr* e '(1))`:
   a) ↦`(Tr 'g3 '(1))'(g6)`
   b) ↦`(Tr 'g5 '(1))'(g5)`

   generando ahora la familia de conjuntos `'((g5) (g6))` que al aplicar la `union*` se obtiene el conjunto `'(g5 g6)`; mientras que el resto de la palabra es finalmente `'()`.
4. En una cuarta recursión se considera el caso base, en el que se evalúa la palabra vacía y finalmente se obtienen los conjuntos `'(g5 g6)`.

Así `(Tr* 'g8 '(0 1))` ↦ `'(g5 g6)`. Si se ha definido ya el autómata `N1`, la transición extendida `(Tr* 'g2 '(0 1 1))` se puede verificar mediante una rápida interacción en `DrRacket`:

```
> (send N1 Tr* 'g8 '(0 1 1))
'(g5 g6)
>
```

En la figura siguiente se han resaltado las transiciones hechas a partir del estado `'g8`, cuando se analiza la palabra `'(0 1 1)` en `N1`:



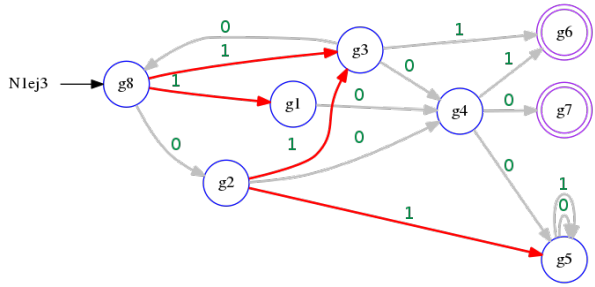
■

7.3.3 Transición extendida por estados

Otro modo de extender el método de transición entre estados del `AFN%`, es calculando las transiciones a partir de un conjunto de estados y un único símbolo del alfabeto de entrada. Este método puede llamarse `Tr+` y recibe como entrada un subconjunto de estados `G` y un símbolo `s` del alfabeto `S`.

El autómata debe evaluar la transición definida en `Tr` (ver página 135) para cada par de argumentos formado por cada uno de los estados en `G` y el símbolo `s`.

```
(Tr+ G s) ↦ (apply union* (map (λ (g) (Tr g s)) G))
```



■ **Ejemplo 7.10** Considerando el mismo autómata N1 definido en la página 134, calcular (Tr+ G 1), cuando G ↦ '(g2 g8).

```
> (define G '(g2 g8))
> (send N1 Tr+ G 1)
'(g1 g3 g5)
>
```

■

7.4 El lenguaje de los AFN%

Al analizar las palabras en S*↤(Kleene* S) a partir del estado inicial, un AFN% induce una bipartición del conjunto S*. Una de las biparticiones contiene las palabras que son aceptadas y la otra contiene al resto de las palabras. El conjunto de las palabras aceptadas forman el **lenguaje del** AFN%. En esta sección se analizará la condición que permite que una palabra sea aceptada o no.

7.4.1 Palabras aceptadas

El propósito principal de un AFN% es decidir si una palabra pertenece a su lenguaje. Las palabras que no pertenecen al lenguaje son interesantes en un sentido complementario. Todas las palabras pertenecen ya sea a una o a la otra clase.

Para determinar si una palabra w es aceptada, el autómata realiza la transición extendida a partir del estado inicial q_0 que ha sido definido para el autómata indeterminista, con lo que se puede definir una función booleana o predicado, con el propósito de determinar la clase correspondiente de la palabra analizada. Suponiendo que N↤⟨Q S T q_0 A⟩ es un AFN%. La función boolena es llamada acepta?, que genera un valor #t cuando al analizar w desde el estado inicial q_0, el conjunto final de estados alcanzados contiene al menos un estado q_d, tal que (en? q_d A) ↦ #t.

$$\texttt{(acepta? w)} \mapsto \begin{cases} \texttt{\#t} : \texttt{(existeUn (}\lambda\texttt{(}q_f\texttt{)(en? }q_f\texttt{ A))} \\ \qquad\qquad\qquad\qquad \texttt{(Tr* }q_0\texttt{ w))} \mapsto \texttt{\#t} \\ \texttt{\#f} : \text{en otro caso.} \end{cases}$$

☞ En la notación convencional, la condición para que una palabra $\omega \in \Sigma^*$ sea aceptada por el autómata indeterminista N, puede escribirse como:

$$\text{acepta}_N(\omega) \Leftrightarrow \exists[q_f \in \delta^*(q_0, \omega)] : q_f \in A$$

El siguiente método debe incluirse en la clase `AFN%`:

```
; (acepta? w) ↦ booleano?
; w : lista?
(define/public (acepta? w)
  (existeUn (λ (qf) (en? qf A)) (Tr* q0 w)))
```

Una vez que se tiene la manera de decidir si una palabra es aceptada o no, es posible también determinar si una palabra pertenece al lenguaje del autómata, ya que una palabra pertenece al lenguaje del autómata indeterminista, únicamente cuando la palabra es aceptada.

☞ En símbolos convencionales se puede escribir

$$\text{acepta}_N(\omega) \Leftrightarrow \omega \in \mathscr{L}_N,$$

donde $\mathscr{L}_N$ denota el lenguaje del autómata N. Además $\mathscr{L}_N \subseteq \Sigma^*$.

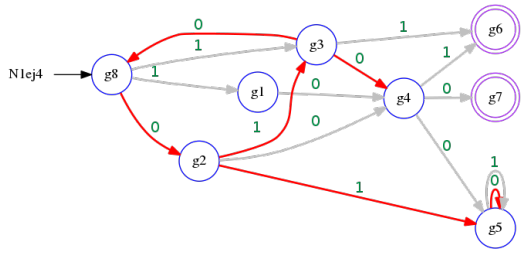
■ **Ejemplo 7.11** Dado el autómata indeterminista `N` mostrado en la página 134, determine si las siguientes palabras son aceptadas:

1. `'(0 1 0)`
2. `'(1 0 1)`
3. `'(0 1 0 1 0 1 0 1)`
4. `'(0 1 0 1 0 1 0 1 1 0)`

Para determinar si una palabra `w` es aceptada por el autómata `N1`, se hace una interacción enviando un mensaje al autómata `N1` solicitando se ejecute en método `acepta?` y el argumento `w`:

```
> (send N1 acepta? '(0 1 0))
#f
> (send N1 acepta? '(1 0 1))
#t
> (send N1 acepta? '(0 1 0 1 0 1 0 1))
#f
> (send N1 acepta? '(0 1 0 1 0 1 0 1 1 0))
#f
>
```

Se muestra la trayectoria de estados que el autómata N1 adquiere al analizar la palabra '(0 1 0) a partir de 'g8. La palabra no es aceptada porque (Tr* 'g8 '(0 1 0)) ↦ '(g8 g4 g5) y ninguno de esos estados es aceptor:



■

Ya que una palabra w en S* está en el lenguaje del autómata únicamente cuando la palabra w es aceptada, entonces se define:

```
; (enLeng? w) ↦ booleano?
; w : lista?
(define/public (enLeng? w) (acepta? w))
```

Aunque el lenguaje del autómata pudiera ser infinito, es posible calcular un subconjunto de éste, al considerar el universo de la cerradura finita de Kleene nKleene* con un límite n determinado (ver página 102). La versión finita del lenguaje del AFN% de tamaño n, contiene las palabras en (nKleene* S n) que han sido aceptadas.

☞ Convencionalmente el lenguaje finito del autómata N se puede expresar como $\mathscr{L}_N^{[n]}$ y es definido: $\mathscr{L}_N^{[n]} = \{\omega \in \Sigma^{[n]} | \text{acepta}_N(\omega)\}$, donde $\Sigma^{[n]}$ es precisamente la cerradura finita de Kleene de hasta un tamaño n.

```
; (nLeng n) ↦ (listade lista?)
; n : numero-entero-no-negativo? = 10
(define/public (nLeng [n 10])
  (filter (λ (w) (acepta? w)) (nKleene* S n)))
```

■ **Ejemplo 7.12** Calcular el lenguaje finito del autómata N1 para n ↦ 2, 3 y 4

```
> (send N1 nLeng 2)
'((1 1))
> (send N1 nLeng 3)
'((1 1) (0 0 0) (0 0 1) (0 1 1) (1 0 0) (1 0 1))
> (send N1 nLeng 4)
'((1 1) (0 0 0) (0 0 1) (0 1 1) (1 0 0) (1 0 1)
(0 1 0 0) (0 1 0 1) (1 0 1 1))
>
```

■

■ **Ejemplo 7.13** ¿Cuántas palabras hay en `(nLeng 20)`?

```
> (card (send N1 nLeng 20))
1559
>
```

■

Código 7.1: *Modelo orientado a objetos de un* `AFN%`

```
(define AFN%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa) S)
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
    (define/public (Tr q s) (Im (tupla q s) T))
    (define/public (Tr* q w)
      (if (pVacia? w)
          (tupla q)
          (apply union* (map (λ (es) (Tr* es (cdr w))) (Tr q (car w))))))
    (define/public (Tr+ G s)
      (apply union* (map (λ (g) (Tr g s)) G)))
    (define/public (acepta? w) (existeUn (λ (qf) (en? qf A)) (Tr* q0 w)))
    (define/public (enLeng? w) (acepta? w))
    (define/public (nLeng [n 10])
      (if (< n 10)
        (filter (λ (w) (acepta? w)) (nKleene* S n))
        (append (filter (λ (w) (acepta? w)) (nKleene* S 10)) (list '...))))
    ))
```

7.5 Equivalencia entre `AFN%` y `AFD%`

A la hora de diseñar un sistema utilizando autómatas finitos deterministas, uno de los principales problemas que surgen es considerar en todos los estados, todos los eventos que hacen reaccionar al autómata y cuál será el nuevo estado en cada situación.

Además, es posible que ante la presencia de un evento, el autómata pueda conducir a más de un estado, o posiblemente a ninguno de ellos. Considerar estas opciones puede resultar difícil para el diseñador de esta clase de máquinas.

La ventaja de modelar un sistema como un AFD%, es que el determinismo obligatorio brinda el control total del comportamiento del autómata, aunque el costo es considerar todas y cada una de las diferentes opciones. Por otro lado, la ventaja de modelar un AFN% es no tener que construir las transiciones para cada caso, sino solamente aquellos que verdaderamente ocurran; sin embargo pueden ocurrir casos no definidos en donde el autómata pierda su operabilidad, o bien no tener el control total del comportamiento del AFN%. Lo deseable es modelar un sistema como un AFN% pero tener la confiabilidad y control de un AFN%.

En esta sección se desarrollará un método para transformar un AFN% en un AFD% que tiene el mismo lenguaje, de ese modo se podrá modelar una máquina de estados finitos con las facilidades de diseño y construcción de un AFN% y con la seguridad y control de un AFD%.

7.5.1 Conversión de AFN% a AFD%

El procedimiento requiere un AFN% que es el que se desea convertir. El proceso se realiza en cinco pasos y produce como resultado un AFD% que acepta el mismo conjunto de palabras.

Para esto supondremos que el AFN% que se desea convertir es:

$$\mathtt{N} \mapsto \langle \mathtt{Q}\ \ \mathtt{S}\ \ \mathtt{T}\ \ \mathtt{q}_0\ \ \mathtt{A} \rangle,$$

el AFD% equivalente estará formalmente definido como

$$\mathtt{D} \mapsto \langle \mathtt{Q}\ \ \mathtt{S}\ \ \mathtt{T}\ \ \mathtt{q}_0\ \ \mathtt{A} \rangle.$$

Para hacer diferencia entre los elementos de cada autómata, se utilizará una notación que incluye el nombre del autómata y el atributo separados; por ejemplo para hacer referencia al conjunto de estados del autómata determinista escribiremos DQ; y el conjunto de estados del autómata indeterminista se escribe NQ.

En términos de notación convencional, se desea convertir el autómata indeterminista $N = \langle Q, \Sigma, \delta, q_0, A \rangle$ a un autómata determinista $D = \langle Q, \Sigma, \delta, q_0, A \rangle$; y para diferenciar los términos se utiliza el atributo y como subíndice el identificador del autómata, por ejemplo se utiliza Q_N para referirse al conjunto de estados de N, mientras que Q_D es el conjunto de estados del autómata D.

Los pasos de este procedimiento se detallan en las siguientes secciones, pero aquí se encuentra una lista resumida:

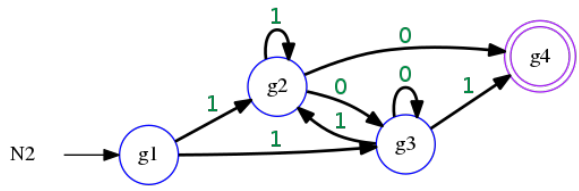
1. Definir DQ, el conjunto inicial de estados del autómata D.
2. Definir DS, el alfabeto del nuevo autómata.
3. Definir las transiciones DT con los estados conocidos y cada símbolo del alfabeto.
4. Actualizar los estados del conjunto DQ, ya que es posible que nuevos estados resulten del conjunto DT creado en el paso anterior, luego repetir los pasos 3 y 4 hasta que no se agreguen más estados.

5. Definir Dq_0, el estado inicial del nuevo AFD% y calcular el conjunto de estados aceptores DA, para finalmente crear el objeto AFD% buscado.

■ **Ejemplo 7.14** Como auxiliar en la explicación del procedimiento de conversión se trabajará con un ejemplo concreto. Suponga que N2 es el autómata indeterminista que se desea convertir, el cual está definido con los siguientes elementos:

```
N2Q↤'(g1 g2 g3 g4)
N2S↤'(0 1)
N2T↤'((g1 1 g2) (g1 1 g3) (g2 0 g3) (g2 0 g4) (g2 1 g2)
(g3 0 g3) (g3 1 g2) (g3 1 g4))
N2q0↤'g1
N2A↤'(g4)
```

```
> (define N2Q '(g1 g2 g3 g4))
> (define N2S '(0 1))
> (define N2T '((g1 1 g2) (g1 1 g3) (g2 0 g3) (g2 0 g4) (g2 1 g2)
(g3 0 g3) (g3 1 g2) (g3 1 g4)))
> (define N2q0 'g1)
> (define N2A '(g4))
> (define N2 (new AFN% [tuplaDef (list N2Q N2S N2T N2q0 N2A)]))
> N2
(object:AFN% ...)
>
```



■

Paso 1: El conjunto inicial de nuevos estados

Digamos que NQ↤'(q_1 ... q_m) es el conjunto de estados del autómata indeterminista. El conjunto de nuevos estados estará compuesto inicialmente de los conjuntos unitarios formados con los elementos de N:

$$\texttt{DQ}\mapsto\texttt{'}((q_1)\ \ldots\ (q_m))$$

La descripción anterior sugiere un procedimiento en el que se cree un conjunto unitario con cada estado de N. Recuerde que para crear un conjunto unitario que contenga a un elemento q, basta agregar (ver la página 40) el elemento q al conjunto vacío, identificado actualmente por vacio (ver página 33).

☞ En la notación convencional se puede obtener el conjunto inicial de estados para el autómata determinista equivalente D calculando: $Q_D = \{\{q\} \in \mathbb{P}(Q_N) | q \in Q_N\}$.

■ **Ejemplo 7.15** Considerando N2 (ver el ejemplo 7.14), el conjunto inicial de los estados para D, que será el nuevo AFD% es DQ↤'((q1) (q2) (q3) (q4)):

```
> (map (λ (q) (conj q)) (send N2 edos))
'((g1) (g2) (g3) (g4))
>
```

Con tabla de transiciones se observa el progreso de la transformación hacia el AFD% objetivo. Los pequeños círculos indican la información faltante.

	○	○	...	○
'(q1)	○	○	...	○
'(q2)	○	○	...	○
'(q3)	○	○	...	○
'(q4)	○	○	...	○

■

Paso 2: El alfabeto

El nuevo AFD% equivalente debe aceptar el mismo lenguaje que el AFN% base. Como el alfabeto de entrada debe ser el mismo, entonces el mismo alfabeto garantiza que las palabras aceptadas por D contendrán los mismos símbolos que las palabras aceptadas por N, así que:

DS↤NS

Si NS ↦ '(s_1 s_2 ... s_n) es el alfabeto del AFN% base, la tabla de transiciones del nuevo AFD% lucirá ahora como:

	s_1	s_2	...	s_n
'(q_1)	○	○	...	○
'(q_2)	○	○	...	○
⋮	⋮	⋮		⋮
'(q_m)	○	○	...	○

Alguno de los '((q_1) ... (q_n)) debe ser el estado inicial.

■ **Ejemplo 7.16** Ahora se agregan al ejemplo los símbolos de entrada y DS↤'(0 1):

	0	1
'(g1)	○	○
'(g2)	○	○
'(g3)	○	○
'(g4)	○	○

■

Paso 3: Las primeras transiciones del nuevo `AFD%`

Una vez que se tiene un primer conjunto de estados y el alfabeto, se deben calcular las transiciones para esos estados.

Para calcular las transiciones entre estados del nuevo autómata determinista a causa de un símbolo de entrada, se debe crear una tupla con cada nuevo estado `q`, cada símbolo del alfabeto `s` y cada transición extendida por estados `Tr+` (ver página 138), utilizando los argumentos anteriores `q` y `s`.

El resultado es una familia de conjuntos. Por cada símbolo en el alfabeto, se genera una familia de transiciones y esto se realiza por cada nuevo estado, por tal motivo se debe obtener un solo conjunto mediante `union*` (ver página 42).

En este momento `DQ` $\mapsto$ `'((`q_1`) ... (`q_m`))` es el conjunto de estados y `DS` $\mapsto$ `'(`s_1 `...` s_n`)` es el alfabeto. De modo que el conjunto de transiciones para estos estados se calcula:

```
(apply union*
       (map (λ(q)
              (map (λ(s)(list q s (Tr+ q s)))DS))
            DQ))
```

En la notación convencional el procedimiento para calcular las primeras transiciones se puede escribir como:

$$T_D = \{\langle q_D, s_D, \delta^+(q_D, s_D)\rangle | q_D \in Q_D \wedge s_D \in S_D\}$$

Aunque la traducción literal de esta manera de cálculo, implica la elaboración del conjunto potencia $Q_D \times S_D$ y posteriormente agregar por la derecha $\delta^+(q_D, s_D)$ cada tupla.

Ahora los estados de `D` son los conjuntos unitarios de estados de `N` y las transiciones son:

	s_1	s_2	...	s_n
`'(`q_1`)`	`(Tr+ '(`q_1`) `s_1`)`	`(Tr+ '(`q_1`) `s_2`)`	...	`(Tr+ '(`q_1`) `s_n`)`
`'(`q_2`)`	`(Tr+ '(`q_2`) `s_2`)`	`(Tr+ '(`q_2`) `s_2`)`	...	`(Tr+ '(`q_2`) `s_n`)`
⋮	⋮	⋮		⋮
`'(`q_m`)`	`(Tr+ '(`q_m`) `s_2`)`	`(Tr+ '(`q_m`) `s_2`)`	...	`(Tr+ '(`q_m`) `s_n`)`

Cada `(Tr+ '(`q_j`) `s_i`)` es un subconjunto de estados en `NQ`, representado como `'(`q_{m_1} q_{m_2} `...` q_{m_k}`)`, un conjunto de tamaño `k`, que contiene algunos de los `m` estados en `NQ`, incluso el `vacio`.

■ **Ejemplo 7.17** En el ejemplo en construcción, se deben que calcular las 8 transiciones [4 estados por 2 símbolos].

1. Para calcular la transición a partir de `'(g1)` con `0`:

```
> (send N2 Tr+ '(g1) 0)
'()
>
```

se crea la transición '((g1) 0 ()) para agregarla a DT.

2. Para calcular la transición a partir de '(g1) con 1:

```
> (send N2 Tr+ '(g1) 1)
'(g2 g3)
>
```

y la transición '((g1) 0 (g2 g3)) se agrega a DT.

Después de calcular las ocho primeras transiciones, la tabla de transiciones luce como sigue:

	0	1
'(g1)	'()	'(g2 g3)
'(g2)	'(g2 g4)	'(g2)
'(g3)	'(g3)	'(g2 g4)
'(g4)	'()	'()

Nótese ahora han resultado nuevos estados que no se encuentran actualmente en DQ, entre ellos el conjunto vacío y algunos estados no unitarios. Este asunto se cubre en el siguiente paso. Y aún faltan los estados aceptores, que corresponden al paso 6. ■

Paso 4: Nuevos estados y nuevas transiciones

Iterativamente se actualizan los estados DQ, con los nuevos estados que surgieron de calcular las primeras transiciones. La lista de estados actualizada sirve para generar nuevas transiciones y obtener nuevos estados; este proceso sigue hasta que no se puedan agregar más estados. Este cuarto paso se trata de detallar cómo se obtienen los nuevos estados y se conforman las nuevas transiciones.

Observe que ahora, después de calcular las primeras iteraciones, hay dos conjuntos de estados, el primero de ellos es DQ actual y el otro se conforma por la imagen de cada transición en DT actual; sin embargo al considerar a DT como una relación, el segundo de los conjuntos se obtiene al calcular el rango de la relación DT.

■ **Ejemplo 7.18** Considere T las transiciones que actualmente tiene DT, calcule el rango de la relación DT.

```
> (define DT '(((g3) 1 (g2 g4)) ((g3) 0 (g3))
((g1) 1 (g2 g3)) ((g1) 0 ()) ((g2) 0 (g3 g4))
((g2) 1 (g2)) ((g4) 0 ()) ((g4) 1 ())))
> (Ran DT)
'((g2 g4) (g3) (g2 g3) () (g3 g4) (g2))
>
```

■

Algunos estados (quizá todos o ninguno) en (Ran DT) aún no han sido considerados. Los estados no considerados son los que pertenecen al (Ran DT)

excepto los que ya se encuentran en DQ y esto es precisamente la diferencia de conjuntos DifC de (Ran DT) respecto a DQ (ver página 44):

$$\texttt{DifC} \leftarrow\!\!\mapsto \texttt{(difc (Ran DT) DQ)}.$$

Si (vacio? DifC) $\mapsto$ #t, entonces el proceso termina; de otro modo, DQ se modifica con agregando los estados de la diferencia de conjuntos recién calculada DifC:

$$\texttt{DQ} \leftarrow\!\!\mapsto \texttt{(union DifC DQ)}.$$

■ **Ejemplo 7.19** Con el ejemplo actual:

```
> (define DT '(((g3) 1 (g2 g4)) ((g3) 0 (g3))
((g1) 1 (g2 g3)) ((g1) 0 ()) ((g2) 0 (g3 g4))
((g2) 1 (g2)) ((g4) 0 ()) ((g4) 1 ())))
> (define DQ '((g1) (g2) (g3) (g4)))
> DQ
'((g1) (g2) (g3) (g4))
> (define DifC (difc (Ran T) Q))
> DifC
'((g3 g4) () (g2 g3) (g2 g4))
> (define DQ (union DC DQ))
> DQ
'((g2 g4) (g2 g3) () (g3 g4) (g1) (g2) (g3) (g4))
>
```

- La variable DifC contiene los nuevos estados que deben ser agregados a DQ, recuerde que los estados en DQ son conjuntos de estados en NQ.
- DQ debe ser actualizado para agregar los nuevos estados.

De este modo la tabla de transiciones tiene ahora más información:

	0	1
'(g1)	'()	'(g2 g3)
'(g2)	'(g3 g4)	'(g2)
'(g3)	'(g3)	'(g2 g4)
'(g4)	'()	'()
'(g2 g3)	○	○
'(g2 g4)	○	○
'(g3 g4)	○	○
'()	○	○

■

Cada pequeño círculo en la tabla, indica que hace falta información en la tabla. La información faltante se refiere a las transiciones a partir de los estados recién descubiertos. Las nuevas transiciones se calcularán utilizando la transición extendida por estados Tr+ en N (ver página 146).

■ **Ejemplo 7.20** Calcular las nuevas transiciones al nuevo AFD% que deben agregarse a causa de haber agregado nuevo estados.

- Para obtener la transición desde el estado '(g2 g3) con el símbolo 0:

```
> (send N2 Tr+ '(g2 g3) 0)
'(g3 g4)
>
```

  obteniendo la transición '((g2 g3) 0 (g3 g4)).
- Para obtener la transición desde el estado '(g2 g3) con el símbolo 1:

```
> (send N2 Tr+ '(g2 g3) 1)
'(g2 g4)
>
```

  obteniendo la transición '((g2 g3) 1 (g2 g4)).

Se obtiene una tabla de transiciones como la que se muestra:

	0	1
'(g1)	'()	'(g2 g3)
'(g2)	'(g3 g4)	'(g2)
'(g3)	'(g3)	'(g2 g4)
'(g4)	'()	'()
'(g2 g3)	'(g3 g4)	'(g2 g4)
'(g2 g4)	'(g3)	'(g2 g4)
'(g3 g4)	'(g3 g4)	'(g2)
'()	'()	'()

■

El procedimiento de agregar nuevos estados y calcular nuevas transiciones, se repite hasta que no se pueden agregar más estados. Para saber cuándo ya no se pueden agregar más estados, es necesario determinar que la diferencia entre los estados obtenidos en las transiciones y los estados del actual conjunto de estados, es nula:

$$\texttt{(vacio? (difc DifC DQ))} \mapsto \texttt{\#t}$$

Como el proceso iterativo involucra tanto a los estados DQ como a las transiciones DT, es conveniente que como resultado de la iteración se devuelve una lista con ambos conjuntos.

Eventualmente el ciclo iterativo termina, pues el número de subconjuntos de estados es finito y en cada iteración se agregan nuevos conjuntos a DQ, con lo que la diferencia de conjuntos del rango de la relación de transiciones respecto al conjunto de estados cada vez aporta menos conjuntos.

```
; DS : el alfabeto en D
; DQ : conjunto actual de estados en D
; DT : transiciones actuales en D
(letrec ([defTQ (λ (Q T)
  (let ((DifC (difc (Ran T) Q)))
    (if (vacio? DifC) ; en #t se termina el proceso
      (list Q T) ; devuelve Q y T en una lista
      (defTQ (union Q DifC) ; se agregan estados a Q,
             ; paso 4: las nuevas transiciones a T
             (union T (apply union*
                 (map (λ (q)
                    (map (λ (s)
                         (list q s (send N Tr+ q s)))
                       DS))
                    DC))))))))])
   (defTQ DQ DT))
```

■ **Ejemplo 7.21** El estado actual del ejemplo que se ha estado construyendo es

	0	1
'(g1)	'()	'(g2 g3)
'(g2)	'(g3 g4)	'(g2)
'(g3)	'(g3)	'(g2 g4)
'(g4)	'()	'()
'(g2 g3)	'(g3 g4)	'(g2 g4)
'(g2 g4)	'(g3)	'(g2 g4)
'(g3 g4)	'(g3 g4)	'(g2)
'()	'()	'()

```
> (define DQ '((g1) (g2) (g3) (g4) (g2 g3) (g2 g4)
(g3 g4) ()))
> (define RanT '() (g2 g3) (g3 g4) (g2) (g3)
(g2 g4)))
> (difc DC DQ)
'()
>
```

En las interacciones `DQ` representa al conjunto actual de estados, `RanT` es el rango de la relación `T`. Como `(vacio? (difc RanT DQ))` $\mapsto$ `#t`, el proceso de agregar estados y transiciones termina. ■

Paso 5: Estado inicial y conjunto de estados aceptores

El estado inicial del nuevo autómata determinista `D` es el único estado unitario `q` en `DQ` que contiene al estado inicial $\texttt{Nq}_0$:

```
Dq0↤(car (filter (λ (q) (y (unitario? q)
                           (en? Nq0 q))) DQ))
```

☞ En términos convencionales se puede escribir $D_{q_0} = \{q \in D_Q | N_{q_0} \in q \wedge |q_0| = 1\}$.

■ **Ejemplo 7.22** El nuevo estado inicial:

```
> (define DQ '((g4) (g3) (g2) (g1) (g3 g4) () (g2 g3) (g2 g4)))
> DQ
'((g4) (g3) (g2) (g1) (g3 g4) () (g2 g3) (g2 g4))
> (car (filter (λ (q) (y (unitario? q) (en? (send N2 eini) q))) DQ))
'(g1)
>
```

■

	s_1	s_2	...	s_n
$'(q_1)$	(Tr+ $'(q_1)$ s_1)	(Tr+ $'(q_1)$ s_2)	...	(Tr+ $'(q_1)$ s_n)
$'(q_2)$	(Tr+ $'(q_2)$ s_2)	(Tr+ $'(q_2)$ s_2)	...	(Tr+ $'(q_2)$ s_n)
⋮	⋮	⋮		⋮
$>'(q_0)$	(Tr+ $'(q_0)$ s_2)	(Tr+ $'(q_0)$ s_2)	...	(Tr+ $'(q_0)$ s_n)
⋮	⋮	⋮		⋮
$'(q_m)$	(Tr+ $'(q_m)$ s_2)	(Tr+ $'(q_m)$ s_2)	...	(Tr+ $'(q_m)$ s_n)

Los estados aceptores del nuevo AFD% tienen al menos un aceptor en N.

DA↤(filter-not (λ (q) (vacio? (intersec q NA))) DQ)

El procedimiento filter-not actúa de manera similar que filter (ver página 32), filter-not selecciona aquellos elementos que no verifican el predicado.

☞ En términos convencionales, el conjunto de estados aceptores se determina con: $A_D = \{q \in Q_D | q \cap A_N \neq \emptyset\}$.

■ **Ejemplo 7.23** Considerando el ejemplo base, el único estado aceptor en N es 'g4. Los estados aceptores en N serán aquellos que que contengan a 'g4, así DA ↦ '((q4) (q2 q4) (q3 q4)) y la tabla de transiciones está completa.

	0	1
>'(g1)	'()	'(g2 g3)
'(g2)	'(g3 g4)	'(g2)
'(g3)	'(g3)	'(g2 g4)
•'(g4)	'()	'()
'(g2 g3)	'(g3 g4)	'(g2 g4)
•'(g2 g4)	'(g3)	'(g2 g4)
•'(g3 g4)	'(g3 g4)	'(g2)
'()	'()	'()

■

Finalmente se crea el nuevo objeto AFD% con los elementos conocidos: (new AFD% [tuplaDef (list DQ DS DT Dq0 DA)]).

Código 7.2: *Conversión de un AFN% a un AFD% equivalente*

```
; (afn->afd N) ↦ AFD%?
; N : AFN%?
(define afn->afd
  (λ (N) ; un afn
    (let* ((DQ (map (λ (q) (conj q)) (send N edos)))
           (DS (send N alfa))
           (DT (apply union* (map (λ (q)
                                    (map (λ (s)
                                           (list q s (send N Tr+ q s)))
                                         DS))
                                  DQ)))
           (X1 (letrec ([defTQ
                         (λ (Q T)
                           (let ((DifC (difc (Ran T) Q)))
                             (if (vacio? DifC)
                               (list Q T)
                               (defTQ (union Q DifC)
                                      (union T (apply union*
                                                      (map (λ (q)
                                                             (map (λ (s)
                                                                    (list q s (send N Tr+ q s)))
                                                                  DS))
                                                           DifC)))))))])
                 (defTQ DQ DT)))
           (DQ (car X1))
           (DT (cadr X1))
           (Dq0 (car (filter
    (λ (q) (en? (send N eini) q))
    DQ)))
           (DA (filter-not
    (λ (q) (vacio? (intersec q (send N acep))))
    DQ)))
      (new AFD% [tuplaDef (list DQ DS DT Dq0 DA)]))))
```

7.5.2 Autómatas equivalentes

Si AF1 y AF2 son autómatas, decimos que AF1 y AF2 son equivalentes si ambos aceptan exactamente el mismo lenguaje. En ocasiones los autómatas aceptan lenguajes infinitos, por lo que se requieren métodos analíticos para determinar su equivalencia. Sin embarvo es posible crear un acercamiento a la equivalencia cuando se consideran los lenguajes de cardinalidad finita, y esto se logra al comparar los lenguajes de cardinalidad finita de cada autómata, esto se puede escribir como (nAF=? AF1 AF2) y se evalúa como #t o #f de acuerdo a la siguiente regla:

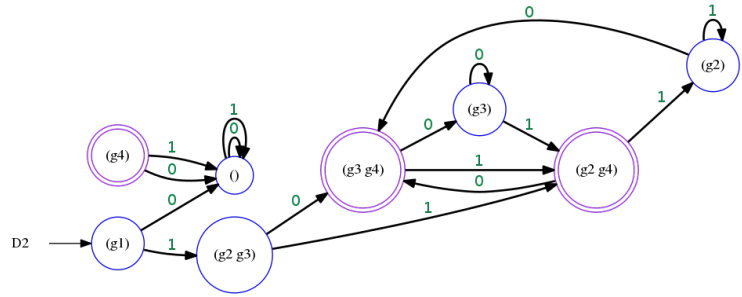


Figura 7.2: *Autómata determinísta* `D` *equivalente al* `AFN%` *de la figura 7.14.*

$$\texttt{(nAF=? AF1 AF2 [n])} \mapsto \begin{cases} \texttt{\#t} : \texttt{(l=? (AF1.nLeng n)} \\ \qquad\qquad \texttt{(AF2.nLeng n))} \mapsto \texttt{\#t} \\ \texttt{\#f} : \text{en otro caso,} \end{cases}$$

donde `(AF1.nLeng n)` y `(AF2.nLeng n)` son el lenguaje finito del autómata `AF1` y del autómata `AF2` respectivamente.

Código 7.3: *Autómatas equivalentes*

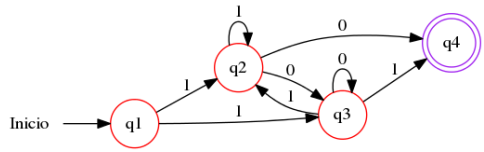
```
; (nAF=? AF1 AF2 [n]) ↦ booleano
; AF1 : AFD% o AFN% o AFE%
; AF2 : AFD% o AFN% o AFE%
; n : numero-entero-positivo = 10
(define nAF=?
  (λ (AF1 AF2 [n 10])
    (l=? (send AF1 nLeng n) (send AF2 nLeng n))))
```

Este procedimiento no explora todas las palabras que pudiera aceptar un autómata, dado que calcula las palabras que pertenecen al lenguaje de hasta longitud `n`. Para autómatas que aceptan un lenguaje infinito, o para hacer una prueba completa de la equivalencia, puede seguirse otro procedimiento [HU79].

■ **Ejemplo 7.24** Realice las siguientes actividades:

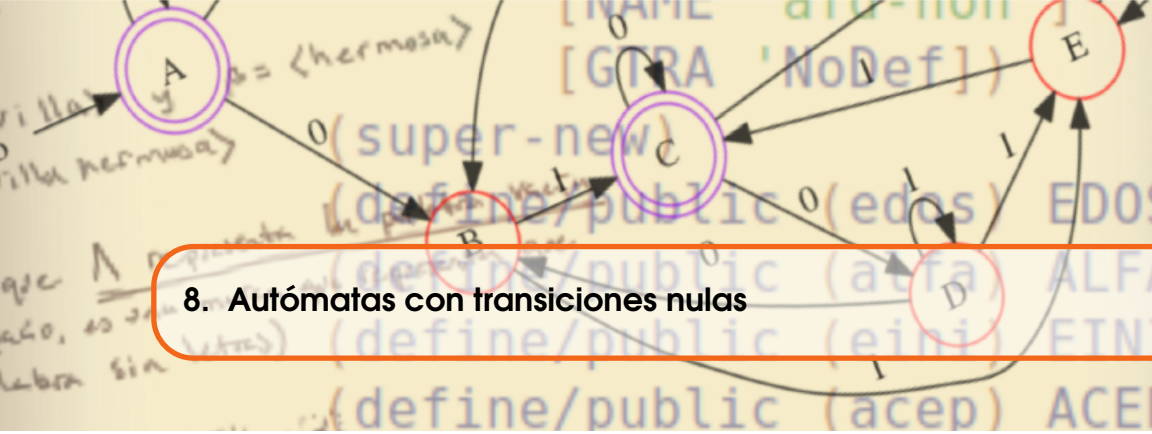
```
; Se define N como un AFN%:
> (define N (new AFN% [tuplaDef (list '(g1 g2 g3 g4) '(0 1) '((g1 1 g2)
(g1 1 g3) (g2 0 g3) (g2 0 g3) (g2 0 g4) (g2 1 g2) (g3 0 g3) (g3 1 g2)
(g3 1 g4)) 'g1 '(g4))]))
>
```

```
; Se obtiene su grafo de transiciones:
> (send N diagTr #:nom "N")
```




```
; Se define D como el AFD% equivalente
> (define D (afn->afd N))
; Se comparan los lenguajes de cardinalidad finita (n=10).
> (nAF=? N D)
#t
>
```

■



8. Autómatas con transiciones nulas

8.1 Presentación

En ocasiones al modelar un fenómeno usando autómatas finitos, se observan los estados en los que se muestra el fenómeno y las causas que ocasionan los cambios entre estados; cuando se crean las transiciones puede ocurrir alguna o algunas de las siguientes situaciones:

1. No se logran determinar todos los posibles estados.
2. No se logran determimar todas las posibles causas que ocasionan los cambios entre estados.
3. Se agregan funcionalidades independientes y no se desea modificar lo que ya se ha diseñado antes.

En cualquier caso, el comportamiento del modelo computacional puede ser indeterminado. En este capítulo se estudia el comportamiento de un autómata finito que cambia de estado aparentemente sin razón alguna. Este tipo de autómatas se llaman «autómatas finitos indeterministas con transiciones nulas [`AFE%`]». Además, se verá que hay una manera de transformar un `AFE%` a un `AFN%`, eliminando las transiciones nulas, de tal modo que acepte el mismo lenguaje que el lenguaje del original `AFE%`.

8.2 Definición de AFE%

Un AFE% es una clase de autómatas finitos indeterministas que son capaces de cambiar de estado aparentemente sin razón alguna. Debido a que no hay razón aparente, convencionalmente se ha denominado **transición nula**, otros nombres que designan el mismo comportamiento son **transición vacía** o **transición épsilon**, **ε-transición**, la letra griega /ε/ es por la primera letra de la palabra *empty* (*tr.* vacío); en ocasiones también puede ocurrir como λ-transiciones, donde /λ/ se ha establecido con el mismo propósito que /ε/, pero en este libro λ tiene el significado muy particular, de crear un procedimiento anónimo. Entonces, AFE% significa literalmente «autómata finito indeterminista con transiciones épsilon». Ya anteriormente se ha definido el símbolo ε para hacer referencia al símbolo nulo (ver página 63). Un AFE% es una quíntupla:

$$\texttt{AFE\%} \mapsto \langle \texttt{Q}\ \texttt{S}\ \texttt{T}\ q_0\ \texttt{A} \rangle$$

Donde:

1. Q↤(q1|Q') es un conjunto finito y no vacío de símbolos que representan los estados del autómata.
2. S↤(s1|S') es un conjunto finito de símbolos que representa el alfabeto del AFE%.
3. q_0 es el estado inicial. Considerando que (en? q_0 Q) $\mapsto$ #t. El estado inicial se selecciona entre los estados del AFE%.
4. A son los estados aceptores. Donde (subc? A Q) $\mapsto$ #t. Los estados aceptores son seleccionados del conjunto de estados del AFE%.
5. T es un conjunto finito de transiciones. Las transiciones son tuplas de la forma '(q s q_d), aquí s puede ser algun símbolo en S, o bien 'ε.

☞ El dominio de la función de transición Tr, incluye el símbolo /ε/ (ver páginas 63 y 65) que representa la transición nula 'ε. Así $Dom(Tr) = (Q \times S) \cup \{\varepsilon\}$. Note que '$\varepsilon$no es parte del alfabeto proporcionado por el modelador o diseñador del autómata, sino que es un símbolo que permite explicar el comportamiento observado del autómata al cambiar de uno a otro estado sin razón aparente. La función de transición está definida con la siguiente llave:

$$Tr : (Q \times S) \cup \{\varepsilon\} \to \mathbb{P}(Q)$$

Observe que el codominio de la función es el conjunto potencia del conjunto de estados, de modo que cada par de estado – símbolo se asocia a un subconjunto de estados.

■ **Ejemplo 8.1** Un centro de trabajo ha creado un código de 3 dígitos para sus empleados, con la siguiente configuración. Los dígitos empleados son el 0 y el 1. El primer dígito corresponde al turno, 0 si es matutino, o 1 si es vespertino; el turno matutino tiene tres departamentos, codificados '(1 0), '(0 0) y '(0 1); y el vespertino tiene solo dos departamentos '(1 0) y '(0 1).

La siguiente definición muestra un AFE% identificado como E1, que determina si una secuencia de tres dígitos corresponde a uno de sus empleados (en ejemplos posteriores se describe el funcionamiento).

```
Q↤'(r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14);
S↤'(0 1);
T↤'((r1 0 r14) (r1 1 r14) (r1 ε r2) (r2 1 r3)
(r2 ε r8) (r3 0 r5) (r3 1 r4) (r4 0 r6)
(r5 1 r6) (r6 ε r7) (r8 0 r9) (r9 0 r12)
(r9 0 r13) (r9 1 r10) (r10 0 r11) (r11 ε r7)
(r12 0 r11) (r13 1 r11) (r14 0 r14) (r14 1 r14))
q0↤'r1
A↤'(r7).
```

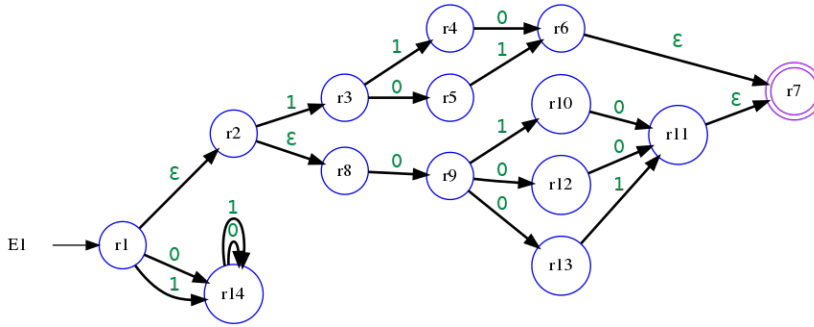
■

8.2.1 Grafo de transiciones de un AFE%

De manera similar que en los casos AFD% y AFN%, un AFE% se puede representar mediante un grafo de transiciones. Las reglas de construcción son las mismas que en los autómatas anteriores:

- Los estados se dibujan como círculos.
- El estado inicial es marcado con una flecha con la palabra INICIO, o con el identificador del autómata.
- Los estados aceptores se indican con círculos en doble línea.
- Para cada transición de la forma '(q s q_d), se dibuja un arco etiquetado con el símbolo s, que inicia en el estado q y llega al estado q_d.

■ **Ejemplo 8.2** El grafo de transiciones del ejemplo 8.1 es el que se muestra enseguida:



■

No se consideran las transiciones nulas desde un estado al mismo. No tiene sentido hacerlo, pues el autómata se encuentra en ese estado aún sin analizar evento alguno. Esta característica puede expresarse en términos del lenguaje como:

```
(existeUn (λ (q) (en? (tupla q 'ε q) T)) Q) ↦ #f
```

☞ En términos convencionales también puede expresarse como:

$$\exists q \in Q : (q, \varepsilon, q) \in T = \texttt{\#f}$$

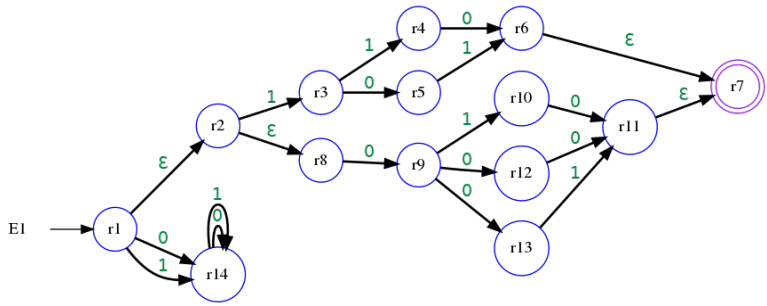
Otra observación es que, a partir de un estado cualquiera, pueden haber transiciones nulas hacia otros estados, pero más de una transición nula entre el mismo par de estados es redundante; se considerará solamente una de esas transiciones repetidas.

8.2.2 Tabla de transiciones de un AFE%

La tabla de transiciones es similar a la tabla de transiciones de un AFE%. La diferencia es una nueva columna identificada con 'ε, en la que se colocan las transiciones nulas.

■ **Ejemplo 8.3** La siguiente tabla de transiciones corresponde al autómata indeterminista con transiciones nulas identificada como E1, del ejemplo 8.1.

E1	0	1	'ε
>'r1	'(r14)	'(r14)	'(r2)
'r2	'()	'(r3)	'(r8)
'r3	'(r5)	'(r4)	'()
'r4	'(r6)	'()	'()
'r5	'()	'(r6)	'()
'r6	'()	'()	'(r7)
•'r7	'()	'()	'()
'r8	'(r9)	'()	'()
'r9	'(r12 r13)	'(r10)	'()
'r10	'(r11)	'()	'()
'r11	'()	'()	'(r7)
'r12	'(r11)	'()	'()
'r13	'()	'(r11)	'()
'r14	'(r14)	'(r14)	'()



■

8.2.3 AFE% como lista de transiciones

Al codificar un AFE% en formato de texto con listas de transiciones, se agrega una transición nula en cada renglón, junto con los estados alcanzables mediante 'ε. En el texto que codifica el AFE%, se considera el caracter /ε/ como si fuera parte del alfabeto del autómata, seguido de los estados alcanzados.

■ **Ejemplo 8.4** AFE% del ejemplo 8.1 en forma de texto.

```
>> r1 0 r14 1 r14 ε r2
-- r2 0 1 r3 ε r8
-- r3 0 r5 1 r4 ε
-- r4 0 r6 1 ε
-- r5 0 1 r6 ε
-- r6 0 1 ε r7
** r7 0 1 ε
-- r8 0 r9 1 ε
-- r9 0 r12 r13 1 r10 ε
-- r10 0 r11 1 ε
-- r11 0 1 ε r7
-- r12 0 r11 1 ε
-- r13 0 1 r11 ε
-- r14 0 r14 1 r14 ε
```

Aquí 'r1 es el estado inicial, el conjunto de estados aceptores es unitario '(r7). El alfabeto es '(0 1), incluye las transiciones nulas con /ε/. ■

8.3 Clase AFE%

Se crea la clase AFE% para identificar un autómata finito indeterminista con transiciones nulas. La tupla ⟨Q S T q_0 A⟩ indica el valor con que debes ser inicializados los atributos básicos del AFE%; todos los elementos son definidos de igual manera que en AFN%, excepto que la lista de transiciones contiene al menos una tupla que incluye 'ε.

Atributos de la clase AFE%

```
(define AFE%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    ; ... Métodos públicos y privados
    ))
```

■ **Ejemplo 8.5** Interacción con definiciones para crea un `AFE%`:

```
> (define Q '(r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14))
> (define S '(0 1))
> (define T '((r1 0 r14) (r1 1 r14) (r1 e r2) (r2 1 r3) (r2 e r8) (r3 0 r5)
(r3 1 r4) (r4 0 r6) (r5 1 r6) (r6 e r7) (r8 0 r9) (r9 0 r12) (r9 0 r13)
(r9 1 r10) (r10 0 r11) (r11 e r7) (r12 0 r11) (r13 1 r11) (r14 0 r14) (r14
    1 r14))
> (define q0 'r1)
> (define A '(r7))
> (define E1 (new AFE% [tuplaDef (list Q S T q0 A)]))
> E1
(object:AFE% ...)
```

■

8.3.1 Métodos informativos

Los métodos informativos son como en la clase `AFN%` anterior.

Métodos informativos de la clase ***AFE%***

```
(define AFE%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa) (filter-not
            (λ (s) (equal? s nulo)) S))
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
    ; ... Métodos públicos y privados
    ))
```

■ **Ejemplo 8.6** Considere nuevamente el `AFE%` del ejemplo base 8.1. Se crea una lista con toda la información que define al autómata `E1`.

```
> (list (send E1 edos) (send E1 alfa) (send E1 tran) (send E1 eini)
(send E1 acep))
'((r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14) (0 1) ((r1 0 r14)
(r1 1 r14) (r1 e r2) (r1 e r8) (r2 1 r3) (r3 0 r5) (r3 1 r4)
(r4 0 r6) (r5 1 r6) (r6 e r7) (r8 0 r9) (r9 0 r12) (r9 0 r13)
(r9 1 r10) (r10 0 r11) (r11 e r7) (r12 0 r11) (r13 1 r11)) r1 (r7))
>
```

■

Note el método informativo (alfa), que no recibe argumentos y devuelve la lista de símbolos diferentes a 'ε; de hecho la lectura literal debe ser, «aquellos símbolos s del alfabeto S, que no son iguales a 'ε ». Las siguientes expresiones son lógicamente equivalentes.

```
; S : el alfabeto
(filter-not (λ (s) (equal? s nulo)) S)

(filter (λ (s) (neg (equal? s nulo))) S)
```

8.4 Cambios de estado en el AFE%

Por tener transiciones nulas el autómata tiene un comportamiento diferente. Las transiciones nulas actúan al momento de alcanzar un estado y su efecto se propaga por las transiciones nulas adyacentes, alzanzando a más estados.

Para calcular una transición a partir de un estado actual q con un símbolo s, es necesario considerar que a partir de q pueden haber transiciones nulas a otros estados. El efecto es que el autómata está de hecho, en aquellos estados alcanzables mediante transiciones nulas y todos ellos son considerados estados actuales; una transición inicia en cada estado actual.

Luego, una vez calculados los estados actuales a causa de las transiciones nulas, se calcula la transición a causa de un símbolo del alfabeto y nuevamente los estados alcanzables debido a las transiciones nulas, para obtener el conjunto final de estados destino. En adelante, consideraremos el autómata E↤⟨Q,S,T,q_0,A⟩ un AFE%. Como es claro que se trata del autómata E, el conjunto de estados se denotará simplemente por Q, en lugar de EQ; de manera similar para cada atributo del autómata.

8.4.1 ε-Cerradura

La cerradura de transiciones nulas es una operación sobre un estado q del autómata E; el operador se identifica con eCerr. La eCerr de un estado q denotada (eCerr q), significa el conjunto de estados alcanzables utilizando exclusivamente transiciones nulas.

Se observa lo siguiente:

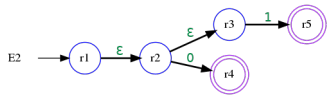
1. Si q es un estado en Q, (en? q (eCerr q))↦ #t, es decir, el estado q pertenece a su propia eCerr.
2. Si (en? (tupla q 'ε q_d) T) ↦ #t, entonces (en? q_d (eCerr q)) ↦ #t y el estado q_d es ε-alcanzable para q.
3. Si (en? p (eCerr q)) ↦ #t y q_d pertenece a (eCerr p), entonces también (en? q_d (eCerr q)) ↦ #t; y también q_d es 'ε-alzanzable para q.

Un error común a la hora de implementar un método para calcular la eCerr de un estado, es caer en un ciclo infinito cuando se observa una situación como en la figura 8.1b. Un procedimiento recursivo equivocado puede ser:

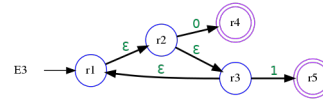
$$(\texttt{eCerr q}) \mapsto \begin{cases} (\texttt{agregar q '}\varepsilon()), \\ \text{cuando } (\texttt{Im (tupla q '}\varepsilon\texttt{) T}) \mapsto \texttt{'}\varepsilon() \\ \\ (\texttt{apply union* (map (}\lambda\texttt{ (p) (eCerr p))} \\ \qquad\qquad\qquad\qquad (\texttt{Im (tupla q '}\varepsilon\texttt{)) T})), \\ \text{cuando } (\texttt{Im (tupla q '}\varepsilon\texttt{) T}) \mapsto (\texttt{p}_1\texttt{| P'}) \end{cases}$$

Esto es equivocado porque en las situaciones como las de la figura 8.1b, al calcular la imagen de una `(tupla q 'ε )`, el resultado nunca es `vacio`, por lo que se incurre en un ciclo infinito como se explica enseguida.

El error puede apreciarse al observar la figura 8.1b, donde ocurren transiciones nulas que forman un ciclo, ya que al calcular la ε-Cerradura de `r1`, se agrega `r2`, luego `r3` y es cuando se observa que `(Im '(r3 'ε)) ↦ '(r1)`, que como no es vacío, el procedimiento continúa entrando a un ciclo eterno.



(a) *La cerradura de transiciones nulas no forman un ciclo.*



(b) *La cerradura de transiciones nulas forman un ciclo.*

Figura 8.1: *Situaciones comunes que se encuentran en las cerraduras de transiciones nulas: (8.1a) sin ciclo; (8.1b) con ciclo.*

☞ En la notación usual de matemáticas se puede escribir la primera aproximación a la `eCerr` como:

$$eCerr(q) = \begin{cases} \{q\} & : Tr(q) = \emptyset; \\ \\ \bigcup\limits_{p \in Tr(q,\varepsilon)} eCerr(p) & : e.o.c. \end{cases}$$

Donde $Tr(q,\varepsilon)$ es la imagen del estado q con el $'\varepsilon$ en la función Tr del autómata.

El error se corrige con un conjunto que coleccione progresivamente los estados que deben pertenecer a la `eCerr` detectados. Este conjunto tiene como valor inicial el conjunto unitario `(conj q)`. En cada iteración este conjunto debe agregar los estados que pertenecen a la diferencia entre la imagen del par formado por el estado y el símbolo nulo bajo la función de transición, con la `eCerr` calculada hasta el momento.

Una implementación en DrRacket para calcular la eCerr de un estado q en Q de un AFE%, considera además del estado inicial q, un conjunto de respuesta identificado con res, donde inicialmente res $\mapsto$ (conj q); donde iterativamente se encuentran los estados ε-alcanzables.

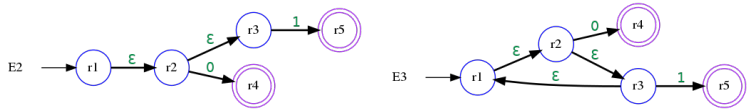
■ **Ejemplo 8.7** Considere los autómatas AFE% que se muestran en la figura 8.1.

- En E2, (eCerr r1) $\mapsto$ '(r1 r2 r3)
- En E2, (eCerr r2) $\mapsto$ '(r2 r3)
- En E2, (eCerr r3) $\mapsto$ '(r3)

En E3, los estados en '(r1 r2 r3) tienen la misma cerradura de transisciones nulas: (eCerr r1) $\mapsto$ '(r1 r2 r3) ■

```
; (eCerr q [res]) ↦ lista?
; q : Q? -- Un estado.
; res : lista? = (crearTupla q)
(define/public (eCerr q [res (conj q)])
   (let* ((x (tupla q ε)) (Tx (Im x T))
          (nc (difc Tx res)))
     (if (vacio? nc)
         res
         (apply union*
             (map (λ (p) (eCerr p (agregar p res)))
                  Tx)))))
```

■ **Ejemplo 8.8** Una vez agregado el método eCerr al AFE%, las siguientes interacciones calculan la ε-Cerradura de algunos estados tanto en E2 como en E3.



1. Calcular la cerradura épsilon del estado 'r2 en E2.
2. Calcular la cerradura épsilon del estado 'r1 en E2.
3. Calcular la cerradura épsilon del estado 'r1 en E3.
4. Calcular la cerradura épsilon del estado 'r3 en E3.

```
> (send E2 eCerr 'r2)
'(r2 r3)
> (send E2 eCerr 'r1)
'(r1 r2 r3)
> (send E3 eCerr 'r1)
'(r1 r2 r3)
> (send E3 eCerr 'r3)
'(r1 r2 r3)
>
```

■

La ε-Cerradura de un estado

```
(define AFE%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa)
            (filter-not (λ (s) (equal? s nulo)) S))
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
    (define/public (eCerr q [res (conj q)])
      (let* ((x (tupla q nulo))
             (Tx (Im x T))
             (nc (difc Tx res)))
        (if (vacio? nc)
            res
            (apply union*
               (map (λ (p) (eCerr p (agregar p res)))
                    Tx)))))
     ; otros métodos públicos y privados.
     ))
```

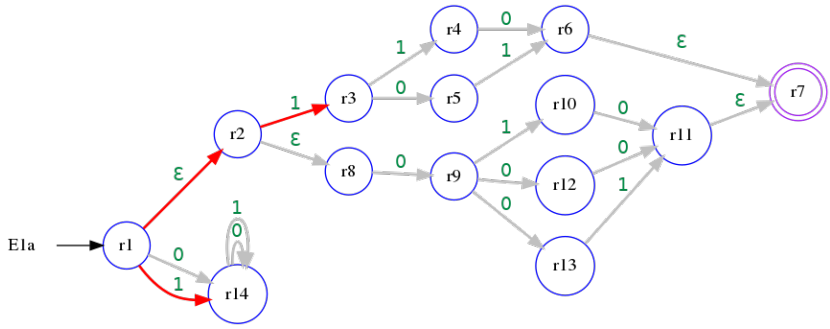
8.4.2 Transiciones del `AFE%`

El cálculo de una transición se realiza en tres pasos, considerando `q` como el estado actual del autómata y `s` como el símbolo que se está analizando:

1. Se calcula la ε-Cerradura de `q` mediante el método `(eCerr q)`, para obtener todos los estados que se deben considerar como *actuales*. De esto debe resultar un conjunto de estados que se llamará `P`.
2. Para cada estado `p` en `P`, se calcula la imagen bajo la función `T`, esto es `(Im (tupla p s) T)` (ver página 55), de esto resulta uno o más conjuntos, por lo que se deben unir para entregar un nuevo subconjunto de estados que se llamará `R`.
3. Para cada estado `r` en el resultado de las transiciones anteriores `R`, se debe realizar ahora `(eCerr r)`, para obtener todos los conjuntos finales y terminar con la unión de todos los conjuntos que resulten.

```
; (Tr q s) ↦ lista?
; q : Q? - un estado.
; s : S? - un símbolo del alfabeto (no 'e).
(define/public (Tr q s)
      (let* (; paso 1 :
             (P (eCerr q))
             ; paso 2 :
             (R (apply union*
                  (map (λ (p) (Im (tupla p s) T))
                       P))))
        ; paso 3 :
        (apply union* (map (λ (r) (eCerr r)) R))))
```

■ **Ejemplo 8.9** Observe el autómata `E1` de la figura del ejemplo 8.2. Si `E1` se encuentra en `'r1`, con un `1` el autómata quedará en el estado `'r14` por la transición inmediata y `'r3` por iniciar en `r2` a causa de una transición nula.




```
> (send E1 Tr 'r1 1)
'(r14 r3)
> (send E1 Tr 'r2 0)
'(r9)
>
```

De hecho, el autómata tiene como estados iniciales el conjunto `'(r1 r2 r8)`, `'r1` porque es el estado inicial definido para el autómata y `'(r2 r8)` son ε-alcanzables. Cualquier actividad del autómata que inicie en `'r1`, también iniciará en `'r2` y en `'r8`.

Sin embargo, cuando la transición se hace desde `'r2`, aunque los estados iniciales son `'r2` y `'r8`, solamente desde `'r2` hay una transición definida para el símbolo `1` y alcanza el estado `'r3`, que es el único estado del conjunto obtenido de `(Tr 'r2 1)`. ■

La transición entre estados del AFE%

```
(define AFE%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa)
          (filter-not (λ (s) (equal? s nulo)) S))
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
    (define/public (eCerr q [res (conj q)])
      (let* ((x (tupla q nulo))
             (Tx (Im x T))
             (nc (difc Tx res)))
        (if (vacio? nc)
            res
            (apply union*
    (map (λ (p) (eCerr p (agregar p res))) Tx)))))
     (define/public (Tr q s)
      (let* ((P (eCerr q))
             (R (apply union*
             (map (λ (p) (Im (tupla p s) T)) P))))
        (apply union* (map (λ (r) (eCerr r)) R))))
     ; otros métodos públicos y privados.
     ))
```

8.4.3 Transiciones extendidas

Del mismo modo que en los `AFN%`, en los `AFE%` se puede extender el método de transición `Tr` para analizar palabras de símbolos, o bien para hacer una transición simple a partir de un conjunto de estados.

1. **Extensión por símbolos**. Para el caso de la transición extendida desde un único estado inicial `q` y una palabra `w` de símbolos del alfabeto `S`:

```
(define/public (Tr* q w)
  (if (pVacia? w)
     (eCerr q)
     (apply union*
       (map (λ (es) (Tr* es (cdr w))) (Tr q (car w))))))
```

2. **Extensión por estados**. Para el caso de la transición extendida a partir de un conjunto `P` de estados iniciales y un único símbolo `s` del alfabeto `S`:

```
(define/public (Tr+ P s)
  (apply union* (map (λ (p) (Tr p s)) P)))
```

Las transiciones extendidas

```
(define AFE%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa)
          (filter-not (λ (s) (equal? s nulo)) S))
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
    (define/public (eCerr q [res (conj q)])
      (let* ((x (tupla q nulo))
             (Tx (Im x T))
             (nc (difc Tx res)))
        (if (vacio? nc)
            res
            (apply union*
    (map (λ (p) (eCerr p (agregar p res))) Tx)))))
     (define/public (Tr q s)
      (let* ((P (eCerr q))
             (R (apply union*
             (map (λ (p) (Im (tupla p s) T)) P))))
        (apply union* (map (λ (r) (eCerr r)) R))))
    ;--- Nuevas transiciones extendidas -----
    (define/public (Tr* q w)
      (if (pVacia? w)
        (eCerr q)
        (apply union* (map (λ (es) (Tr* es (cdr w)))
                           (Tr q (car w))))))
    (define/public (Tr+ P s)
      (apply union* (map (λ (p) (Tr p s)) P)))
      ; Otros métodos públicos y privados.
      ))
```

■ **Ejemplo 8.10** Considerando el autómata E1 definido en la página 156, determine cuáles son los estados finales después de analizar las siguientes palabras:

1. '(0), empezando desde los estados 'r8 y 'r1.
2. '(1 0 0), empezando desde el estado 'r5

```
> (apply union*
    (map (λ (q) (send E1 Tr* q '(0))) '(r1 r8)))
'(r14 r9)
> (send E1 Tr* 'r5 '(1 0 0))
'()
>
```

El segundo ejemplo muestra la posibilidad de que el autómata quede en ningún estado. Operativamente esto significa que el autómata no tiene definido algún estado para este caso. ∎

8.5 Lenguaje de los AFE%

El lenguaje de un AFE% consiste de las palabras aceptadas. El criterio es el mismo que en los AFN%: si después de analizar una palabra a partir del estado inicial, el autómata alcanza al menos un estado aceptor, entonces la palabra es aceptada, de otro modo la palabra no es aceptada.

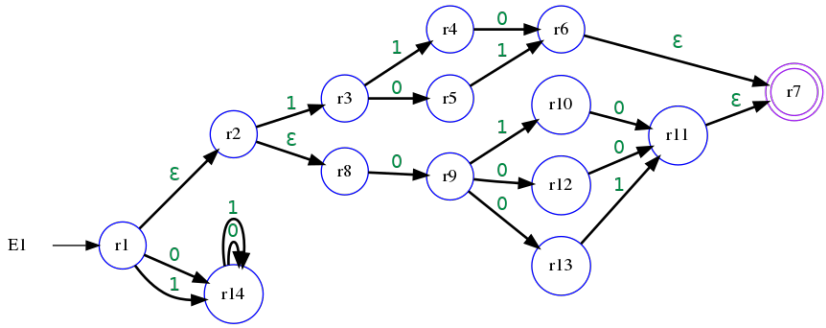
El conjunto de palabras de longitud finita que se pueden formar con los símbolos del alfabeto (Kleene* S), puede ser seccionado en dos conjuntos. El conjunto de las palabras aceptadas y el resto de las palabras. Debido a que los conjuntos forman una bipartición, no existe palabra alguna en (Kleene* S) que al mismo tiempo sea aceptada y que no lo sea.

Código 8.1: *Clase AFE%*

```
(define AFE%
  (class object%
    (init tuplaDef)
    (field [Q  (list-ref tuplaDef 0)]
           [S  (list-ref tuplaDef 1)]
           [T  (list-ref tuplaDef 2)]
           [q0 (list-ref tuplaDef 3)]
           [A  (list-ref tuplaDef 4)])
    (super-new)
    ;----------
    (define/public (edos) Q)
    (define/public (alfa)
          (filter-not (λ (s) (equal? s nulo)) S))
    (define/public (tran) T)
    (define/public (eini) q0)
    (define/public (acep) A)
    (define/public (eCerr q [res (conj q)])
      (let* ((x (tupla q nulo))
             (Tx (Im x T))
             (nc (difc Tx res)))
        (if (vacio? nc)
            res
            (apply union*
    (map (λ (p) (eCerr p (agregar p res))) Tx)))))
     (define/public (Tr q s)
      (let* ((P (eCerr q))
             (R (apply union*
             (map (λ (p) (Im (tupla p s) T)) P))))
        (apply union* (map (λ (r) (eCerr r)) R))))
    (define/public (Tr* q w)
      (if (pVacia? w)
        (eCerr q)
```

```
33        (apply union* (map (λ (es) (Tr* es (cdr w)))
34                           (Tr q (car w))))))
35    (define/public (Tr+ P s)
36      (apply union* (map (λ (p) (Tr p s)) P)))
37    ; -- Nuevos métodos agregados ------
38    (define/public (acepta? w) (existeUn
39          (λ (qf) (en? qf A)) (Tr* q0 w)))
40    (define/public (enLeng? w) (acepta? w))
41    (define/public (nLeng [n 10])
42      (if (<= n 10)
43        (filter (λ (w) (acepta? w)) (nKleene* S n))
44        (append (filter (λ (w) (acepta? w))
45                        (nKleene* S n)) (list '...))))
46    ))
```

■ **Ejemplo 8.11** Obtenga el lenguaje del autómata AFE% definido en la página 156, del cual se muestra su grafo de transiciones.



E1 puede analizar palabras de cualquier longitud, sin embargo solo unas cuantas palabras serán aceptadas.

Las palabras que teminen en el estado 'r14 no son aceptadas, aunque hay algunas que además de terminar en 'r14, también terminan en 'r7, las cuales sí son aceptadas.

Las dos palabras con prefijo con '(1) y con sufijo '(1 0) o '(0 1) son aceptadas.

Las tres palabras que inician con 0 y terminadas ya sea con '(1 0), o bien por '(0 0), o por '(0 1), también son aceptadas.

El autómata E1 acepta las palabras '((1 1 0) (1 0 1) (0 1 0) (1 0 0) (0 0 1)). Otro modo de decirlo es: «El lenguaje de E1 es '((1 1 0) (1 0 1) (0 1 0) (1 0 0) (0 0 1))».

```
> (send E1 nLeng 3)
'((0 0 0) (0 0 1) (0 1 0) (1 0 1) (1 1 0))
>
```

■

8.6 Equivalencia entre autómatas

Es posible crear un AFN% a partir de un AFE% y que acepte exactamente el mismo conjunto de palabras. En esta sección se desarrollará un método para transformar un AFE% en un AFN% que tiene el mismo lenguaje, **pero sin transiciones nulas**.

La ventaja de crear modelos basados en AFE% es que se tiene una mayor flexibilidad en el desarrollo y construcción del modelo, comparado con los AFN%. Esto es porque las transiciones nulas permiten agregar «nuevas funcionalidades» al autómata sin hacer muchos cambios.

La desventaja es que las transiciones nulas aportan información que en muchos casos resulta ser irrelevante, lo que provoca retrasos en el cálculo.

8.6.1 Conversión de un AFE% a un AFN%

El procedimiento requiere un AFE% que es precisamente el que se desea convertir. El proceso se realiza en tres pasos y produce un nuevo autómata de la clase AFN%.

Supondremos que el AFE% que se desea convertir es $E \mapsto \langle Q\ S\ T\ q_0\ A \rangle$, y el AFN% equivalente estará formalmente definido como $N \mapsto \langle Q\ S\ T\ q_0\ A \rangle$.

Se utilizará nuevamente una notación que antepone el nombre del autómata al atributo. Así EA se refiere al conjunto de estados aceptores del autómata E; mientras que NA es el conjunto de estados aceptores del AFN%.

■ **Ejemplo 8.12** Para seguir el método de conversión, se tomará el autómata E4, cuyas transiciones se describen a continuación y su grafo de transiciones se muestra en la figura 8.2:

```
>> r1 0 1 ε r2
-- r2 0 r4 1 ε r3
** r3 0 r4 1 r2 r5 ε
-- r4 0 r5 1 r1 ε
** r5 0 1 ε
```

```
> (define E4 (new AFN% [tuplaDef (list
 '(r1 r2 r3 r4 r5) '(0 1) '((r1 ε r2) (r2 0 r4)  (r2 ε r3)
 (r3 0 r4) (r3 1 r2) (r3 1 r5) (r4 0 r5) (r4 1 r1)) 'r1 '(r3 r5))]))
> E4
(object:AFN% ...)
>
```

■

El procedimiento para convertir un AFE% en un AFN% equivalente, se puede describir en tres pasos:

1. Establecer la información constante.
2. Obtener el nuevo conjunto de transiciones.
3. Crear el nuevo AFN%.

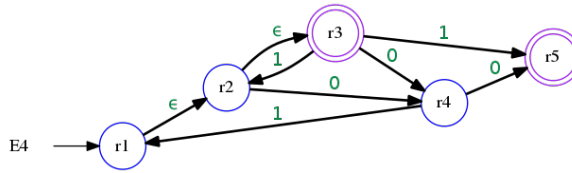


Figura 8.2: *Autómata finito indeterminista con transiciones nulas* `E4`*, tomado como ejemplo para construir un* `AFN%` *equivalente.*

8.6.2 `AFE%` → `AFN%`

Casi toda la información del `AFE%` define al nuevo `AFN%`, de hecho las diferencias son las transiciones, que deben ser de tal forma que no contengan la transición nula; y la otra diferencia es el conjunto de estados aceptores, que posiblemente se modifique a causa de las transiciones nulas; de modo que el conjunto de estados, el alfabeto y el estado inicial son pasados directamente al nuevo `AFN%`:

`NQ`$\leftarrow\!\!\shortmid$`EQ`
`NS`$\leftarrow\!\!\shortmid$`ES`
`Nq`$_0\leftarrow\!\!\shortmid$`Eq`$_0$

Es posible empezar a construir una función que haga la conversión. La nueva función puede llamarse `afe->afn` y recibe un objeto de la clase `AFE%` y deberá generar como resultado un nuevo objeto de la clase `AFN%`. El primer paso de la conversión queda como sigue:

```
; (afe->afn E) ↦ AFN%
; E : AFE%
(define afe->afn
  (λ (E)
    (let* (; Paso 1:
           (NQ (send E edos))
           (NS (send E alfa))
           (Nq0 (send E eini))
           ; :
           )
   ; :
   )))
```

La forma especial `let*` es como `let` [página 53], pero evalúa las expresiones una por una y tan pronto como se obtiene su valor se asocia a su identificador, permitiendo que esté disponible para el cálculo de otros valores dentro del mismo ámbito `let*`.

```
(let* ([id val-expr] ...) cuerpo ...+)
; id : identificador
; expr : Expresión que otorga un valor
```

Los nuevos estados aceptores

El estado inicial $\texttt{Nq}_0$ será aceptor siempre que su ε-Cerradura contenga un estado aceptor, esto es que `(intersec (eCerr Nq0) NA)`$\mapsto$ `(nq|NQ')` (ver la notación para un conjunto no vacío en la página 35).

☞ En la notación convencional se puede expresar como: $(eCerr(Nq_0) \cap NA) \neq \emptyset \rightarrow Nq_0 \in NA$.

Esto significa que al estar en el estado inicial y sin consumir ningún símbolo, el autómata también se encuentra en un estado aceptor. En el ejemplo, el estado inicial en el nuevo `AFN%` debe ser aceptor, porque la `eCerr` del estado inicial incluye un estado aceptor.

```
; (afe->afn E) ↦ AFN%?
; E : AFE%?
(define afe->afn
  (λ (E)
   (let* ((NQ (send E edos))
          (NS (send E alfa))
          (Nq0 (send E eini))
          (NA (if (vacio? (intersec (send E eCerr Nq0) (send E acep)))
                  (send E acep)
                  (union (send E acep) (conj Nq0))))
                   ;;
           )
   ; :
   )))
```

Las transiciones `NT` se obtienen a partir de los estados `NQ` y los símbolos `NA` previamente establecidos, siguiendo el siguiente protocolo:

1. Obtener la tabla de transiciones `tTr` para el nuevo `AFE%`, calculando el subconjunto de estados para cada par $\langle$`q s`$\rangle$, con `q` en `NQ` y `s` en `NS`. La información obtenida en `tTr` es un resultado previo para el cálculo final de las transiciones, de modo que `tTr` no aparece como información en `N`.
2. Crear la función `NT` a partir de los subconjuntos obtenidos en `tTr`, formando las tuplas de la forma `'(q s` $\texttt{q}_d$`)` que forman las transiciones en `N`.

Tabla de transiciones

Para cada estado `q` en `NQ` y para cada símbolo `s` en `NS`, se crea una tripleta `(tupla q s` $\texttt{Q}_d$`)`, con el estado `q`, el símbolo `s` y el subconjunto de estados $\texttt{Q}_d$ que resulta de hacer una transición en `E` a partir de `q` y `s`.

```
; NQ : (send E Q)
; NA : (send E A)
(append* (map (λ (q)
                  (map (λ (s)
                          (tupla q s (send E Tr q s)))
                  NA))
          NQ))
```

Una manera sencilla de ver el progreso del trabajo es con la tabla de transiciones de `N`.

Si `Q` $\mapsto$ `'(`q_1 ... q_m`)` y `S` $\mapsto$ `'(`s_1 ... s_n`)`, se puede crear una tabla donde se muestre con pequeños círculos la información que se necesita para definir en nuevo `AFN%`.

N	s_1	...	s_n
q_1	○	○	○
⋮	⋮	⋮	⋮
q_m	○	○	○

La información faltante se obtiene de las transiciones en `E`, utilizando cada par `(tupla q s)`, donde `(en? q NE)` $\mapsto$ `#t` y `(en? s NA)` $\mapsto$ `#t`.

Se debe obtener `(Tr q s)`, utilizando la transición definida en `E`, ya que se considera tanto la ε-Cerradura antes de la transición, la transición misma y la ε-Cerradura posterior a la transición.

El resultado de cada transición es un subconjunto de estados y será la imagen de `(tupla q s)` bajo la función `NT`.

■ **Ejemplo 8.13** La tabla de transiciones de `E4` [figura 8.2]:

E4	0	1	'ε
>>'r1	'()	'()	'(r1)
'r2	'(r4)	'()	'(r3)
**'r3	'(r4)	'(r2 r5)	'()
'r4	'(r5)	'(r1)	'()
**'r5	'()	'()	'()

La tabla de transiciones `tTr` generada a partir de `E4`:

tTr	0	1
>>'r1	(ETr 'r1 0) $\mapsto$ '(r4)	(ETr 'r1 1) $\mapsto$ '(r2 r3 r5)
'r2	'(r4)	'(r2 r3 r5)
**'r3	'(r4)	'(r2 r3 r5)
'r4	'(r5)	'(r1 r2 r3)
**'r5	'()	'()

■

Conjunto de transiciones

La información en la tabla de transiciones `tTr` del nuevo autómata `N`, contiene la información resumida; cada entrada de la tabla `tTr` se puede expresar como una 3-tupla de la forma `(tupla q s` Q_d`)`, donde Q_d $\mapsto$ `(ETr q s)`, sin embargo para efecto de obedecer cabalmente la definición, se requiere una lista de tuplas de la forma `(q s` q_d`)`, donde q_d es un estado (no un subconjunto de estados), de modo que se debe crear una tupla por cada elemento de Q_d, para cada subconjunto obtenido en `tTr`.

```
; TT : tabla de transiciones de N
(append*
  (map (λ (t)
         (map (λ (qd)
                (tupla (list-ref t 0)
                       (list-ref t 1)
                       qd))
              (last t)))
       tTr))
```

Resta crear el nuevo AFN%: N↤(new AFN% [tuplaDef (list NQ NS NT Nq0 NA)]).

Código 8.2: *afe->afn: Transforma un AFE% en un AFN%*

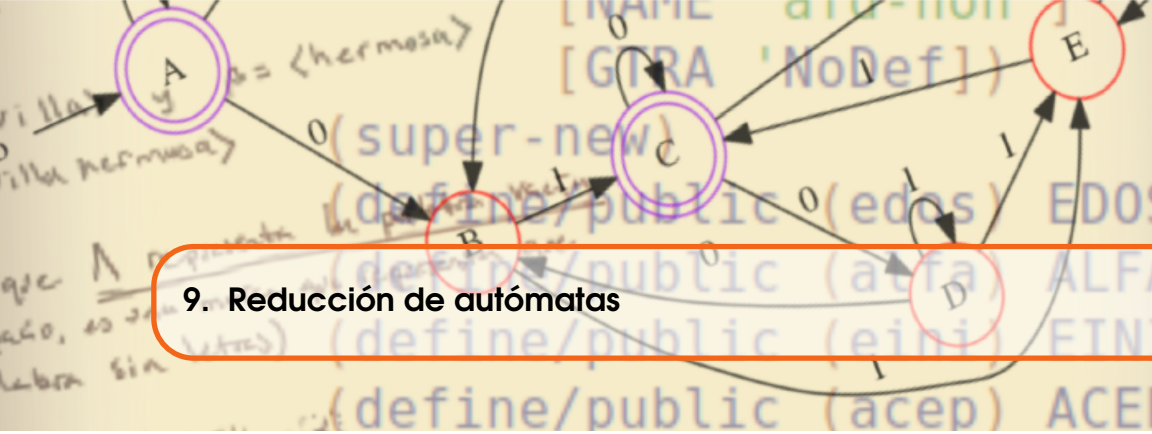
```
; (afe->afn E) ↦ AFN%?
; E : AFE%?
(define afe->afn
  (λ (E) ; <- un AFE%
    (let* ((NQ (send E edos))
           (NS (send E alfa))
           (Nq0 (send E eini))
           (NA (if (vacio? (intersec (send E eCerr Nq0)
                                     (send E acep)))
                   (send E acep)
                   (union (send E acep) (tupla Nq0))))
           (tTr (append*
                  (map (λ (q)
                         (map (λ (s)
                                (tupla q s (send E Tr q s)))
                              NS))
                       NQ)))
           (NT (append*
                 (map (λ (t)
                        (map (λ (qd)
                               (tupla (list-ref t 0)
                                      (list-ref t 1)
                                      qd))
                             (last t)))
                      tTr))))
      (new AFN% [tuplaDef (list NQ NS NT Nq0 NA)]))))
```

■ **Ejemplo 8.14** Describa los elementos del autómata finito indeterminista N4, equivalente al autómata con transiciones nulas E4 definido en la página 171.

```
> (list (send N4 edos) (send N4 alfa) (send N4 eini) (send N4 tran) (send
    N4 acep))
'((r1 r2 r3 r4 r5) (0 1) r1 ((r1 0 r4) (r1 1 r2) (r1 1 r3) (r1 1 r5)
(r2 0 r4) (r2 1 r2) (r2 1 r3) (r2 1 r5) (r3 0 r4) (r3 1 r2) (r3 1 r3)
(r3 1 r5) (r4 0 r5) (r4 1 r1) (r4 1 r2) (r4 1 r3)) (r3 r5 r1))
>
```

■

IV Manipulación de AF



9. Reducción de autómatas

9.1 Presentación

Cuando se diseña un modelo basado en autómatas finitos, sin duda es más fácil trabajar con autómatas indeterministas con transiciones nulas, pues esto evita considerar aquellos casos que pudieran ser imposibles, o intracendentes según el criterio del diseñador. Sin embargo, una consecuencia inherente es incluir situaciones para las que el autómata no tiene determinado algún estado, corriendo el riesgo de que el autómata quede estancado durante su operación.

La situación anterior se corrige transformando un autómata indeterminista a un AFD%, con lo que se garantiza la completa operación del autómata.

En la transformación de autómatas indeterministas a deterministas, ocasionalmente suelen ocurrir una o varias de las siguientes situaciones:

1. Cambia el alfabeto. Esto puede ocurrir cuando se cambia un sistema de codificación para identificar nuevos elementos.
2. Al transformar autómatas de indeterministas a deterministas, resulta una nomenclatura de estados muy complicada, por lo que se requiere reducir el nombre de los estados.
3. En la transformación suelen aparecer estados inaccesibles, por lo que es posible tener autómatas con una reducción en el número de estados.

4. En la transformación en ocasiones aparecen estados equivalentes, por lo que también es posible en ocasiones reducir el número de estados.

Como cada autómata indeterminista, aún con transiciones nulas, se puede transformar en un `AFD%`, se utiliza sólamente autómatas deterministas para estudiar los casos y escribir el código.

9.2 Autómatas homomorfos

Un homomorfismo es una función que permite transformar un objeto en otro, cambiando algunas propiedades pero preservando su estructura. La parte importante en un homomorfismo es que la transformación preserva la estructura del objeto original, aunque el objeto resultante tiene características diferentes.

Los autómatas homomorfos, o más precisamente autómatas finitos homomorfos, son el resultado de aplicar un homomorfismo a un autómata finito. El autómata resultante de una transformación homomorfa, puede tener un alfabeto diferente si se ha aplicado un homomorfismo de alfabeto, o bien puede tener un conjunto diferente de estados si se aplica un homomorfismo de estados, en cualquier caso se debe garantizar que la estructura general del autómata es preservada.

En esta sección se estudiarán dos tipos de homomorfismos, el primero transforma el alfabeto y el segundo transforma el conjunto de estados. Las razones para hacer esto redundan en tener autómatas más simples en su aspecto general, pero que tengan las mismas propiedades que el autómata original.

9.2.1 Homomorfismos de alfabeto

En autómatas finitos, un homomorfismo de alfabeto es una función que toma un autómata y genera otro autómata de la misma clase pero con un alfabeto diferente, aunque preservando toda la demás información. El propósito de aplicar un homomorfismo de alfabeto a un autómata, es sustituir cada símbolo de un alfabeto, por un símbolo de otro alfabeto. Si `F`$\leftarrow\!\shortmid\langle$`Q S T` q_0 `A`$\rangle$ es un autómata finito, con alfabeto `FS`$\leftarrow\!\shortmid$`'(`s_1 `...` s_n`)`, se desea crear un nuevo autómata `G`$\leftarrow\!\shortmid\langle$`Q S T` q_0 `A`$\rangle$ donde:

- `GQ` tenga los mismos estados que `FQ`
- `GS` tenga símbolos diferentes de `FS`, tomado de un alfabeto alterno `S-alt`.
- `GT` con las mismas transiciones que `FT`, observando el cambio de símbolos.
- `Gq0` tenga el mismo estado inicial que `Fq0`
- `GA` tenga los mismos estados aceptores que `FA`

■ **Ejemplo 9.1** Si `F` y `G` son autómatas homomorfos por alfabeto, donde `FS` $\mapsto$ `'(0 1)` y como resultado del homomorfismo se obtiene `GS` $\mapsto$ `'(a b)` haciendo `0` $\mapsto$ `'a` y `1` $\mapsto$ `'b`. Suponiendo que `'(0 1 0 0 1)` es aceptada por `F`, entonces `'(a b a a b)` debería ser aceptada por `G`. ■

Aunque el alfabeto alterno `S-alt` contiene los nuevos símbolos para el nuevo autómata `G`, se requiere crear una asociación biyectiva que transforme un símbolo de `FS` en un símbolo de `S-alt`.

El homomorfismo se crea como una función explícita con dominio en `FS` y codominio en `S-alt`, de la misma cardinalidad por ser una biyección. Así para cada símbolo en `S`, le corresponde un símbolo en `S-alt` y todos los símbolos quedan cubiertos.

```
(define Smorf (map (λ (s t) (tupla s t)) S S-alt))
```

■ **Ejemplo 9.2** Si `S` $\mapsto$ `'(0 1 2)` y `S-alt` $\mapsto$ `'(a b c)`, una función homomorfa puede ser

$$\texttt{Smorf} \mapsto \texttt{'((0 a) (1 b) (2 c))},$$

aunque pueden arbitrariamente definirse otros mapeos. ■

Así `Smorf` $\mapsto$ `'(`$(s_1\ t_1)$ `...+)` es una lista de pares, con el primer elemento en `S` y el segundo elemento en `S-alt` y es un homomorfismo usado para obtener el nuevo alfabeto `NS`. El nuevo alfabeto `NS` se obtiene mapeando para cada símbolo `s` del alfabeto original `FS` su asociación respecto del homomorfismo `Smorf`.

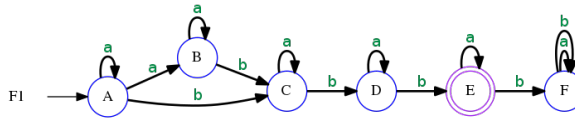
```
(define NS (map (λ (s) (car (assoc s Smorf))) S))
```

El procedimiento `assoc` busca en la lista de pares `lst`, el primer par encabezado con el elemento igual a `v` y devuelve como salida un `#f` si no se encuentra algún par de esa característica, o bien devuelve el par encontrado.

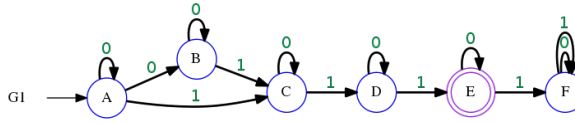
```
; v : cualquier
; lst : lista de tuplas
; is-equal? : (any/c any/c ->any/c)
(assoc v lst [is-equal?]) ↦  (or/c par? #f )
```

```
> (assoc 2 '((0 a) (1 b) (2 c) (3 d)))
'(2 c)
> (cadr '(2 c))
'c
>
```

Una vez que se tiene la función homomorfa `Smorf` y el nuevo alfabeto `NS`, es necesario transformar también las tuplas de las transiciones para que cada transición se realice considerando los nuevos símbolos. Para obtener un nuevo conjunto de transiciones, se realiza un mapeo donde para cada transición `t`, que es de la forma `'(q s` q_d`)`, se debe obtener una nueva transición de la forma `'(q t` q_d`)`, donde `t` $\mapsto$ `(Smorf s)`.



(a) *Autómata generador* $F1 \mapsto \langle Q\ S\ T\ q_0\ A \rangle$



(b) *Autómata resultante* $G1 \mapsto \langle Q\ S\ T\ q_0\ A \rangle$ *homomorfo a* $F1$

Figura 9.1: *Aplicación de un homomorfismo de alfabeto con* $Smorf \mapsto$ `'((a 0) (b 1))`, *en (a) el autómata generador* $F1$, *en (b) el autómata resultante* $G1$.

```
(define NT
  (map (λ (t)
     (tupla (list-ref t 0)
            (cadr (assoc (list-ref t 1) Smorf))
            (list-ref t 2))) T))
```

Código 9.1: *Homomorfismo de alfabeto*

```
1 (define S-morf
2   (λ (AF [S-alt '()])
3     (let* ((NQ (send AF edos))
4            (S  (send AF alfa))
5            (SI (build-list (card S) values))
6            (Smorf (if (vacio? S-alt)
7                       (map (λ (s t) (tupla s t)) S SI)
8                       (map (λ (s t) (tupla s t)) S S-alt)))
9           (NS (map (λ (s) (cadr (assoc s Smorf))) S))
10          (NT (map (λ (t) (tupla
11                           (list-ref t 0)
12                    (cadr (assoc (list-ref t 1) Smorf))
13                           (list-ref t 2)))
14                   (send AF tran)))
15           (Nq0 (send AF eini))
16           (NA (send AF acep)))
17       (new
18        (cond ((is-a? AF AFD%) AFD%)
19              ((is-a? AF AFN%) AFN%)
20              ((is-a? AF AFE%) AFE%))
21        [tuplaDef (list NQ NS NT Nq0 NA)])))))
```

■ **Ejemplo 9.3** Defina `F1` como el `AFN%` de la figura 9.1a, aplique el homomorfismo de alfabeto `Smorf` $\mapsto$ `'((a 0) (b 1))` para obtener el nuevo alfabeto y

nuevas transiciones de G1, luego compare las transiciones del autómata generador F1 con las del autómata resultante G1.

```
> (define F1 (new AFN% [tuplaDef (list '(A B C D E F) '(a b) '((A a A)
(A a B) (A b C) (B a B) (B b C) (C a C) (C b D) (D a D) (D b E) (E a E)
(E b F) (F a F) (F b F)) 'A '(E))]))
> (define G1 (S-morf F1))
> (map (λ (AF) (send AF alfa)) (list F1 G1))
'((a b) (0 1))
> (map (λ (AF) (send AF tran)) (list F1 G1))
'(((A a A) (A a B) (A b C) (B a B) (B b C) (C a C)
(C b D) (D a D) (D b E) (E a E) (E b F) (F a F)
(F b F)) ((A 0 A) (A 0 B) (A 1 C) (B 0 B) (B 1 C)
(C 0 C) (C 1 D) (D 0 D) (D 1 E) (E 0 E) (E 1 F)
(F 0 F) (F 1 F)))
>
```

■

9.2.2 Homomorfismo de estados

Durante las transformaciones de autómatas, especialmente de AFN% a AFD%, el autómata determinista resultante tiene un conjunto de estados conformado por subconjuntos de estados de AFN%, esto supone mayor espacio para dibujar el diagrama de transiciones; pero es mejor que el tratar de interpretar conjuntos de símbolos para identificar un nuevo estado.

Estas u otras razones pueden dar origen a renombrar los estados y para hacer esto se aplica un homomorfismo de estados. De nuevo, un homomorfismo es una función biyectiva que asocia los elementos de un conjunto con elementos de otro conjunto. Como antes, es necesario crear una regla que determine qué nuevo nombre debe tener cada estado y posteriormente aplicar esa regla en un nuevo autómata tanto en su conjunto de estados, como en sus transiciones.

Si F↤⟨Q S T q0 A⟩ es el autómata generador, se debe construir un autómata resultante G↤⟨Q S T q0 A⟩, con la misma estructura general, excepto por un renombramiento de estados que afecta en la definición del autómata, también a las transiciones, al estado inicial y al conjunto de estados aceptores.

Considere que FQ↤'(q_q ... q_m) es el conjunto de estados del autómata generador y NQ↤'(p_1 ... p_m) un nuevo conjunto de símbolos, que son aquellos por los que se quiere hacer la sustitución. Los nuevos estados NQ tienen identificadores generados arbitrariamente, aunque se sugiere:

1. NQ ↦ '(1 ... m); una secuencia de números, o bien
2. NQ ↦ '(l_1 ... l_m); un conjunto de m identificadores.

Se define el homomorfismo de estados Qmorf como una función explícita que enlista pares de símbolos, donde el primer símbolo de cada par pertenece a FQ y el segundo símbolo de cada par pertenece a NQ.

```
(define Qmorf  (map (λ (q p) (list q p)) Q NQ))
```

Las transiciones se deben transformar, utilizando el criterio `Qmorf` para establecer los estados en cada transición de la forma `'(q s` q_d`)`, en transiciones de la forma `'(p s` p_d`)`. Si `T` son transiciones, cada nueva transición `t` en `NT`, es creada sustituyendo los estados por estados obtenidos por `Qmorf`, sin cambiar el símbolo del alfabeto.

```
(define NT
  (map (λ (t)
    (list (cadr (assoc (list-ref t 0) Qmorf))
          (list-ref t 1)
          (cadr (assoc (list-ref t 2) Qmorf)))) T))
```

El nuevo estado inicial `Nq0` es el último término de la asociación que inicia con `Fq0`.

```
(define Nq0 (last (assoc q0 Qmorf)))
```

El nuevo subconjunto de estados aceptores `NA` se obtiene de transformar cada estado `q` en `FQ` en algún estado de `NQ`, considerando el criterio de transformación `Qmorf`.

```
(define NA (map (λ (q) (last (assoc q Qmorf))) A)))
```

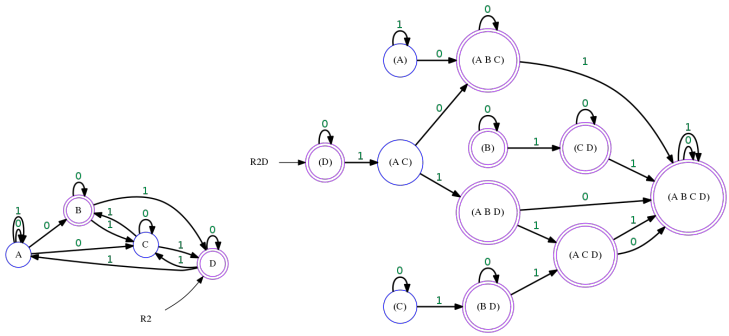
Donde `A` es el subconjunto de estados aceptores en el autómata generador. El alfabeto permanece sin cambio.

Código 9.2: *Homomorfismo de estados*

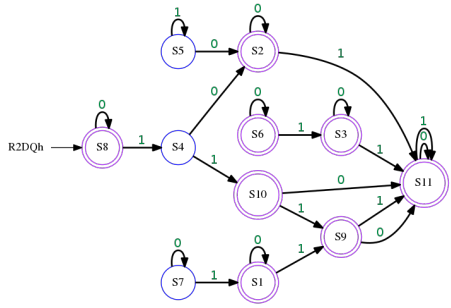
```
1 ; (Q-morf AF [lit]) ↦ (or AFD% AFN% AFE%)
2 ; AF : (or AFD% AFN% AFE%)
3 ; lit = "q": string?
4 (define Q-morf
5   (λ (AF [lit "q"]) ; Un prefijo para el nombre de estados.
6     (let* ((Q  (send AF edos))
7            (NQ0 (map (λ (i) (format "~a~s" lit i))
8             (build-list (card Q) (λ (i) (+ i 1)))))
9            (opB (if (paraTodo (λ (q)
10                   (string->number q)) NQ0)
11                       string->number
12                       string->symbol))
13           (NQ  (map (λ (p) (opB p)) NQ0))
14           (Qmorf  (map (λ (q p) (list q p)) Q NQ))
15           (NS  (send AF alfa))
16           (NT  (map (λ (t)
17            (list
18                (last (assoc (list-ref t 0) Qmorf))
19                (list-ref t 1)
20                (last (assoc (list-ref t 2) Qmorf)))) (send AF tran)))
21           (Nq0 (last (assoc (send AF eini) Qmorf)))
22           (NA  (map (λ (q) (last (assoc q Qmorf))) (send AF acep))))
23       (new
24        (cond ((is-a? AF AFD%) AFD%)
25              ((is-a? AF AFN%) AFN%)
```

```
26              ((is-a? AF AFE%) AFE%))
27        [tuplaDef (list NQ NS NT Nq0 NA)])))))
```

■ **Ejemplo 9.4** R2 es el autómata que se muestra enseguida y R2D, la conversión de AFN% a AFD% de R2:



Después del homomorfismo de estados en R2D y de renombrar los estados con el prefijo "S" se tiene R2DQh:




```
> (define R2 (new AFN% [tuplaDef (list '(A B C D)
'(0 1) '((A 0 A) (A 0 B) (A 0 C) (A 1 A) (B 0 B)
(B 1 C) (B 1 D) (C 0 C) (C 1 B) (C 1 D) (D 0 D)
(D 1 A) (D 1 C)) 'D '(B D)))]))
> (define R2D (afn->afd R2))
> (define R2DQh (Q-morf R2D "S"))
> R2DQh
(object:AFN% ...)
> (list (send R2D edos) (send R2Qh edos))
'(((B D) (A B C) (C D) (A C) (A) (B) (C) (D) (A C D)
(A B D)(A B C D))(S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11))
>
```

■

9.3 Estados alcanzables

Otra manera de reducir un autómata es considerar únicamente los estados alcanzables. Un estado `q` es un estado **alcanzable** cuando existe al menos una secuencia de eventos reconocibles por el autómata tal que, a partir del estado inicial, el autómata en algún momento llegue a encontrarse en el estado `q`. Un estado que no es alcanzable, es **inalcanzable**.

El estado inicial de cualquier autómata es alcanzable.

Un modo de detectar los estados alcanzables es mediante el grafo de transiciones; quitando los autociclos, observando y seleccionando aquellos nodos [estados] que tienen valencia de entrada mayor que `0`, incluyendo el estado inicial en este conjunto; sin embargo pueden haber algunos nodos [estados] con valencia de entrada mayor que `0` y que no son alcanzables.

■ **Ejemplo 9.5** En el autómata finito determinista `R2DQh` de la figura 9.2, los estados `'S5`, `'S6` y `'S7` son inalcanzables, porque los nodos que los representan tienen valencia `0` [no reciben alguna arista], pero los nodos que representan a los estados `'S3` y `'S1` también son inalcanzables, aunque tienen valencia de entrada mayor que `0`. ■

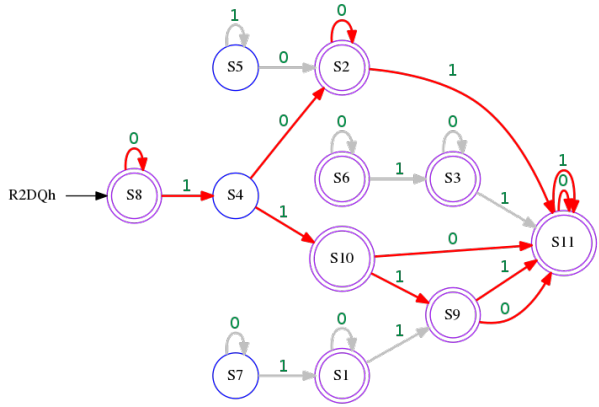


Figura 9.2: *Cálculo de estados accesibles desde el estado inicial.*

Para determinar los estados alcanzables se puede seguir una estrategia de búsqueda primero en amplitud [CLRS09]. Esta estrategia también se utilizó al definir el método `eCerr` para encontrar la `'`ε-Cerradura de un estado [ver página 163].

A diferencia de `eCerr`, donde solamente se consideraron las transiciones nulas, para determinar los estados alcanzables, es necesario considerar todos los símbolos del alfabeto.

Digamos que AF↤⟨Q S T q_0 A⟩ es un autómata finito de cualquier tipo ; con q ↦ AFq_0 el estado inicial, res ↦ ⟨q⟩ la tupla unitaria con el estado inicial; y (en? 'e S) ↦ #t para garantizar que se consideran las transiciones vacías.

$$(\texttt{q0Alc AF [q } q_0\texttt{] [res (conj q)]}) \mapsto \begin{cases} \texttt{res, cuando (vacio? nc)} \mapsto \texttt{\#t} \\ \\ \texttt{(apply union*} \\ \quad \texttt{(map (}\lambda\texttt{(p)} \\ \quad\quad \texttt{(q0Alc AF p (agregar p res)))} \\ \quad\quad \texttt{nc))}, \\ \texttt{cuando (vacio? nc)} \mapsto \texttt{\#f}. \end{cases}$$

Aquí, nc representa los estados aún no son considerados, que es la diferencia entre el conjunto de todas las imágenes de las transiciones al considerar todos los símbolos del alfabeto, respecto de los estados ya considerados en res, esto es: nc ↦ (difConjuntos (apply union* (λ (s) (Im ⟨q s⟩ T)) S) res).

Código 9.3: *Selección de estados alcanzables de un autómata*

```
; (q0Alc AF [q] [res]) ↦ (listade Q?)
; AF : (or AFD% AFN% AFE%)
; q = (send AF eini) : Q?
; res = (tupla q) : lista?
(define q0Alc
  (λ (AF [q (send AF eini)] [res (tupla q)])
    (let* ((T (send AF tran))
           (S (union (send AF alfa) '(e)))
           (Tx (apply union*
           (map (λ (s) (Im (tupla q s) T)) S)))
           (nc (difConjuntos Tx res)))
      (if (vacio? nc)
          res
          (apply union*
    (map (λ (p) (q0Alc AF p (agregar p res))) nc))))))
```

■ **Ejemplo 9.6** Si R2DQh es el autómata de la figura 9.2, calcular los estados alcanzables y los inalcanzables.

```
> (q0Alc R2DQh)  ; Los estados alcanzables.
'(S10 S11 S2 S4 S8 S9)
> (difc (send R2DQh edos) (q0Alc R2DQh)) ; Los estados inalcanzables.
'(S7 S6 S5 S3 S1)
>
```

■

Para reconstruir el autómata se consideran las transiciones con estados accesible. Los aceptores son estados aceptores accesibles.

Código 9.4: *AF con reducción de estados inalcanzables*

```
; (Q-alc AF) ↦ (or AFD% AFN% AFE%)?
; AF : (or AFD% AFN% AFE%)
(define Q-alc
  (λ (AF)
    (let ((Qalc (q0Alc AF)))
      (new (cond ((is-a? AF AFD%) AFD%)
                 ((is-a? AF AFN%) AFN%)
                 ((is-a? AF AFE%) AFE%))
       [tuplaDef (list Qalc (send AF alfa)
                 (filter (λ (t)
                   (en? (car t) Qalc)) (send AF tran))
                        (send AF eini)
                 (filter (λ (q) (en? q Qalc))
                   (send AF acep)))] ))))
```

■ **Ejemplo 9.7** Defina `R2DQhQa` como el `AFD%` equivalente, sin los estados inaccesibles de `R2DQh` de la figura 9.3.

```
> (define R2DQhQa (Q-alc R2DQh))
> R2DQhQa
(object:AFD% ...)
> (send R2DQhQa edos)
>
```

■

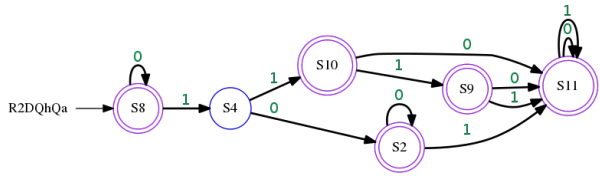


Figura 9.3: *`AFD%` sin estados inaccesibles equivalente al de la figura 9.2.*

9.4 Estados distinguibles

Otro criterio para minimizar autómatas se basa en el mismo propósito del autómata, que es el de diferenciar o hacer distinguir las palabras creadas con el alfabeto en dos clases: las palabras aceptadas y claro, las palabras que no son aceptadas.

Cuando una palabra es aceptada, no importa cuál fué el estado aceptor alcanzado. En este sentido los estados aceptores son indistinguibles [Law03] entre ellos mismos y son distinguibles del resto de estados.

La idea general es identificar los estados que bajo algún criterio resulten distinguibles; para luego reunir en un solo conjunto los que resulten indistinguibles; el proceso se repite para determinar la menor cantidad de conjuntos de estados indistinguibles, para luego definir de un nuevo autómata con (posiblemente) un menor número de estados.

Hay diferentes algoritmos para minimizar autómatas [Hop71, HU79, Jak15], aquí se utilizará un algoritmo basado en la creación de clases de equivalencia, que se conoce como el **algoritmo de llenado de tabla** [*table-filling algorithm*] [HU79, Jak15]. El algoritmo produce una lista de pares de estados distinguibles, los pares no generados son de estados indistinguibles (equivalentes). Se considerará `D` $\mapsto \langle$`Q S T` q_0 `A`$\rangle$, un `AFD%`.

Si `p` y `q` son estados en `Q`, `p` y `q` son **distinguibles** si hay una palabra `w` en `S*` que:

$$\texttt{(ox (en? (Tr* p w) A) (en? (Tr* q w) A))} \mapsto \texttt{\#t},$$

al analizar `w` desde cada estado, uno de los casos acepta la palabra y en el otro no.

El algoritmo utiliza una tabla donde cada renglón y cada columna es etiquetado con un estado. La entrada `'(p q)` representa la comparación de los estados `p` con `q`; se pone una marca en esa entrada si los estados son distinguibles. Se omiten las entradas simétricas porque se representan conjuntos. Las entradas reflejadas también se omiten ya que cada estado es indistinguible de sí mismo.

Las entradas de la tabla en cuestión pueden verse como una lista de pares de estados, que se puede llamar `LPQ`. Cada elemento de `LPQ` es un conjunto de la forma `'(p q)`, donde `p` es diferente a `q` lo que define una relación asimétrica.

Sobre los elementos de `LPQ` se define la relación **0-distinguible**, denotada por `0dis?`, cuando `p` y `q` son distinguibles mediante una palabra de longitud `0`:

$$\texttt{(0dis? p q)} \mapsto \begin{cases} \texttt{\#t}, \text{cuando } \texttt{(ox (en? p A) (en? q A))} \mapsto \texttt{\#t} \\ \texttt{\#f}, \text{en otro caso.} \end{cases}$$

Una bipartición de `LPQ` tiene los subconjuntos `QDi` (estados 0-distinguibles) y `QIn` (los que son indistinguibles).

■ **Ejemplo 9.8** Suponga el autómata finito determinista `D5` de la figura 9.4.

```
LPQ ↦ '((q2 q1) (q3 q1) (q3 q2) (q4 q1) (q4 q2) (q4 q3)
     (q5 q1) (q5 q2) (q5 q3) (q5 q4) (q6 q1) (q6 q2)
     (q6 q3) (q6 q4) (q6 q5))
QDi ↦ '((q4 q1) (q4 q2) (q4 q3) (q5 q1) (q5 q2) (q5 q3)
     (q6 q4) (q6 q5))
QIn ↦ '((q6 q3) (q6 q2) (q6 q1) (q5 q4) (q3 q2) (q3 q1)
     (q2 q1))
```

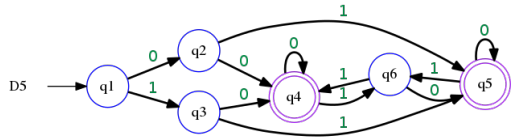


Figura 9.4: *Los estados* `'q2` *y* `'q4` *son estados 0-distinguibles, porque uno de ellos es aceptor mientras el otro no lo es. Los estados* `'q4` *y* `'q5` *son 0-distinguibles, porque ambos son aceptores; y los estados* `'q2` *y* `'q3` *también son 0-distinguibles porque ambos no son aceptores.*

'q2					
'q3					
'q4	•0	•0	•0		
'q5	•0	•0	•0		
'q6				•0	•0
	'q1	'q2	'q3	'q4	'q5

■

De los conjuntos en `QIn` ambos son aceptores, o bien ambos son no aceptores, pero pueden ser distinguibles al considerar otras palabras. Con palabras de longitud `1`, los estados `p` y `q` son distinguibles, si al hacer una transición desde cada estado y algún símbolo, se alcanza un conjunto distinguible. Dos estados son distinguibles cuando `(Tr p s)` es aceptor o bien `(Tr q s)` es aceptor, con `(en? s A)` $\mapsto$ `#t`.

$$\texttt{(1dis? p q)} \mapsto \begin{cases} \texttt{\#t}, \text{cuando } \texttt{(ox(en?(Tr p s)A)} \\ \qquad\qquad\qquad\qquad \texttt{(en? (Tr q s)A)} \mapsto \texttt{\#t}. \\ \texttt{\#f}, \text{en otro caso}, \end{cases}$$

que de manera equivalente puede ser reescrito como

$$\texttt{(1dis? p q)} \mapsto \begin{cases} \texttt{\#t}, \text{cuando} \\ \qquad \texttt{(en?} \langle \texttt{(Tr p s) (Tr q s)} \rangle \texttt{ QDi)} \mapsto \texttt{\#t} \\ \\ \texttt{\#f}, \text{en otro caso.} \end{cases}$$

El predicado `1dis?` produce el conjunto `1dis` con los conjuntos de estados distinguibles con palabras de longitud `1`, tomados de los pares de estados en `QIn`.

$$\texttt{1dis} \mapsto \texttt{(filter (}\lambda\texttt{ (pq) (1dis? (car p) (cadr q))) QIn)}$$

Es necesario ahora actualizar los conjuntos `QDi` agregando los pares en `1dis`; y quitando los mismos del conjunto `QIn`:

`QDi` ↦ `(union QDi 1dis)` y
`QIn` ↦ `(difConjuntos QIn QDi)`

■ **Ejemplo 9.9** Siguiendo con el ejemplo 9.8, se agregan a `QDis` los pares de estados en `1dis`, que deben ser removidos de `QIn`.

'q2	$\bullet_1$				
'q3	$\bullet_1$				
'q4	$\bullet_0$	$\bullet_0$	$\bullet_0$		
'q5	$\bullet_0$	$\bullet_0$	$\bullet_0$		
'q6	$\bullet_1$			$\bullet_0$	$\bullet_0$
	'q1	'q2	'q3	'q4	'q5

■

Una vez conocidos los pares de estados 1-distinguibles, pueden ser agregados a `QDi`; así el proceso se extiende para encontrar los pares de estados 2-distinguibles, para agregarlos a `QDi`; luego el mismo procedimiento con los 3-distinguibles y así en adelante hasta que ya no se puedan encontrar nuevos estados distinguibles.

Los predicados como `0dis?`, `1dis?`, `2dis?` pueden generalizarse en un único predicado `dis?`, que toma un autómata finito determinista `D`; un par de estados `p` y `q` en `DQ`; y una lista de referencia de pares de estados distinguibles `QDi`, que contiene los pares de estados distinguibles detectados hasta el momento; y devuelve `#t` si existe un par de estados distinguibles `ab` en `QDi`, en el que para algún símbolo `s` en `DS` se cumpla:

`(c=? (conj (Tr p s) (Tr q s)) ab)` ↦ `#t`

☞ En la notación matemática usual, la condición es escrita como:

$$\{\delta(p,\sigma),\delta(q,\sigma)\} = \{a,b\}; \text{con } \{a,b\} \in \texttt{QDi}; \sigma \in \Sigma.$$

Código 9.5: *Verifica si un par de estados son distinguibles*

```
1 ; (dis? D q1 q2 QDi) ↦ booleano?
2 ; D : AFD% -- Un autómata determinista.
3 ; p : DQ? -- Un estado.
4 ; q : DQ? -- Un estado.
5 ; QDi : (listade lista?)
6 (define dis?
7   (λ (D p q QDi)
8     (existeUn
9       (λ (ab)
10        (existeUn (λ (s)
11       (c=?  (conj (send D Tr p s) (send D Tr q s))
12                ab)) (send D alfa)))  QDi)))
```

Claramente es posible crear un procedimiento recursivo que permita obtener el conjunto de estados distinguibles:

Código 9.6: *Lista de estados distinguibles*

```
; (qDist D QDi QIn) ↦ (listade lista?)
; D : AFD%
; QDi : (listade lista?)
; QIn : (listade lista?)
(define qDist
  (λ (D QDi QIn)
    (let ((nvosDis
       (filter (λ (ee)
                 (dis? D (car ee) (cadr ee) QDi))
               QIn)))
    (cond ((vacio? QIn) QDi)
          ((vacio? nvosDis) QDi)
          (else (qDist D
                    (union QDi nvosDis)
                    (difConjuntos QIn nvosDis)))))))
```

Cuando ya no es posible encontrar nuevos estados distinguibles, los conjuntos que permanecen en `QIn` marcan los pares de estados que son equivalentes entre ellos. En la tabla, los pares de estados que son equivalentes `Qeq` se destacan por no haber sido marcados en la etapa anterior.

`Qeq` ↦ `(difc LPQ (qDist D QDi QIn))`

■ **Ejemplo 9.10** Siguiendo con el ejemplo 9.8, cuando ya no se agregan nuevos pares al conjunto `QDis`, los pares en `QIn` son indistinguiblen entre ellos.

'q2	$\bullet_1$				
'q3	$\bullet_1$	≅			
'q4	$\bullet_0$	$\bullet_0$	$\bullet_0$		
'q5	$\bullet_0$	$\bullet_0$	$\bullet_0$	≅	
'q6	$\bullet_1$	≅	≅	$\bullet_0$	$\bullet_0$
	'q1	'q2	'q3	'q4	'q5

`Qeq` ↦ `'((q3 q2) (q5 q4) (q6 q2) (q6 q3))`

Los conjuntos `'(q3 q2)`, `'(q5 q4)`, `'(q6 q2)` y `'(q6 q3)`, contienen estados equivalentes. ■

La lista de pares de estados equivalentes `Qeq` puede verse como una lista de clases de equivalencia, donde por el momento cada una de esas clases contiene exactamente dos estados. Se puede crear un procedimiento `reduceClases` para reducir el número de clases de equivalencia, uniendo conjuntos de estados equivalentes de acuerdo al criterio:

Si `(conj p q)` y `(conj q r)` están en `Qeq` [son estados equivalentes], entonces `p`, `q` y `r` son equivalentes entre sí.

El procedimiento `reduceClases` funciona con una lista de clases de equivalencia `LK` y una lista reducida de clases de equivalencia `res`, que inicialmente es

vacía y que iterativamente se le agregarán nuevas clases de equivalencia, para entregar como resultado la lista `LKred` donde:

$$\texttt{(<= (card LKred) (card LK))} \mapsto \texttt{\#t}$$

El procedimiento termina cuando se han analizado todas las clases inicialmente dadas en `LK`:

$$\texttt{(vacio? LK)} \mapsto \texttt{\#t}.$$

Un caso recursivo se da cuando la intersección del primer conjunto en la lista `LK` con cada uno de los conjuntos en `res` es vacío. En este caso, el primer conjunto en `LK` se agrega a la lista reducida `res` y se recursa el procedimiento con el resto de los conjuntos de `LK` junto con la modificación a la lista reducida.

```
(reduceClases (cdr LK) (agregar (car LK) res))
```

El otro caso recursivo es cuando la intersección del primer conjunto en `LK` con algún conjunto en `res` no ha sido vacío.

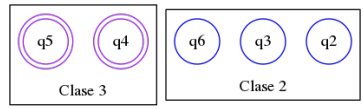
En este otro caso, es necesario seleccionar el primer conjunto en la lista reducida `res`, cuya intersección con el conjunto en cuestión `(car LK)` no ha sido vacía. Digamos que `K1` es tal conjunto. Por otro lado, digamos que `K2` contiene al resto de los conjuntos, exceptuando a `K1`; es necesario crear un tercer conjunto `K3` con los conjuntos en `K2` junto con la unión de `(car LK)` con `K1`; para iniciar una nueva recursión considerando ahora el resto de los conjuntos por analizar `(cdr LK)` y el nuevo conjunto `K3`.

Código 9.7: *Reduce clases de equivalencia*

```
; (reduceClases LK [res]) ↦ (listade lista?)
; LK : (listade lista?)
; res : (listade lista?) = '()
(define reduceClases
  (λ (LK [res '()])
    (cond ((vacio? LK) res)
          ((paraTodo (λ (K)
                (vacio? (intersec (car LK)
                                  K)))
                    res)
           (reduceClases (cdr LK)
                         (agregar (car LK) res)))
          (else (let*
             ((K1 (car (filter-not
                    (λ (K)
                      (vacio? (intersec (car LK) K)))
                    res)))
              (K2 (difc res (list K1)))
              (K3 (agregar (union (car LK) K1) K2)))
      (reduceClases (cdr LK) K3))))))
```

■ **Ejemplo 9.11** Supónga que el algoritmo ha detectado que los estados en los pares siguiente son equivalentes. Se debe reducir las clases de equivalencia.

```
> (reduceClases '((q3 q2) (q5 q4) (q6 q2) (q6 q3)))
'((q6 q3 q2) (q5 q4))
>
```



■

Los conjuntos en `LKred` no contienen a los estados que son indistinguibles de todos los demás, los cuales conforman clases de equivalencia unitarias.

Las clases de equivalencia unitarias se obtienen de los estados en `Q` y las clases que deben agregarse son aquellas cuya intersección con cada conjunto en la lista de clases de equivalencia es vacá. Si la lista de clases unitarias del conjunto `Q` de estados es

$$\texttt{Kunit} \mapsto \texttt{(map (}\lambda\ \texttt{(q) (conj q)) Q)},$$

otro método para lograr incluir aquellas clases de equivalencia unitarias, es unir `Kunit` con `Qeq`, que es la lista de conjuntos de estados equivalentes; y reducir nuevamente las clases de equivalencia.

Código 9.8: *Lista de clases de estados equivalentes*

```
; (qEqv AF) ↦ (listade lista?)
; AF : AFD%
(define Q-Keq
  (λ (AF)
    (let* ((LPQ (spCart (send AF edos)))
           (A   (send AF acep))
           (QDi (filter
      (λ (p) (ox (en? (send AF Tr* (car p)  '()) A)
                 (en? (send AF Tr* (cadr p) '()) A)))
              LPQ))
           (QIn (difc LPQ QDi))
           (Qeq (difc LPQ (qDist AF QDi QIn)))
      (Kunit (map (λ (q) (conj q)) (send AF edos))))
     (reduceClases (union Keq Qeq)))))
```

■ **Ejemplo 9.12** Estados equivalentes del autómata `D5` de la figura 9.4:

```
> (Q-Keq D5)
'((q6 q3 q2) (q1) (q5 q4))
>
```

■

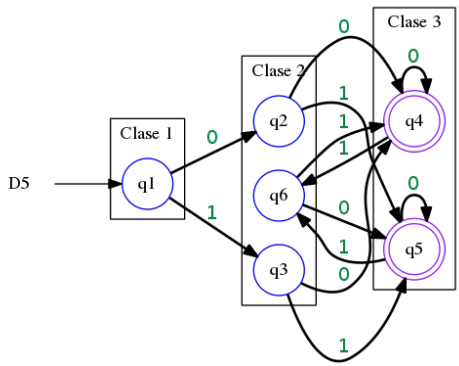


Figura 9.5: *Reestructura del autómata determinista* `D5` *(página 188), considerando los estados equivalentes. Observe que el estado destino de las transiciones para cada estado, es la misma clase para cada símbolo del alfabeto.*

9.4.1 Construcción del `AFD%` reducido

Si `D` es el `AFD%` que se desea reducir y `NQ` $\mapsto$ `(Q-Keq D)` es la lista de conjuntos equivalentes, es posible generar el nuevo `AFD%`.

La construcción del nuevo estado contempla cinco elementos, cada uno de los cuales corresponde a cada elelemento que conforma la definición del autómata.

Los nuevos estados

Los nuevos estados `NQ` se obtienen de los conjuntos de estados equivalentes obtenidos mediante el proceso descrito anteriormente.

`NQ` $\mapsto$ `(Q-Keq D)`.

El nuevo alfabeto

El nuevo alfabeto debe contener exactamente los mismos símbolos que el autómata generador.

`NS` $\mapsto$ `DS`.

El nuevo conjunto de transiciones

Las nuevas transiciones `NT`, son calculadas a partir de los nuevos estados que ya han sido calculados, creando una tupla para cada nuevo estado `nq` en el conjunto de estados `NQ` y para cada símbolo `s` del nuevo alfabeto `NS`.

Las transiciones tienes la forma $\langle$`nq s (Kde (Tr (car nq) s) NQ)`$\rangle$, donde `(Kde (Tr (car nq) s) NQ)` es la clase del estado que alcanza el autómata al hacer la transición desde un estado representante `nq` con el símbolo `s`; con más detalle:

- El `(car nq)` es un representante de la clase `nq` [ver la figura 9.5], ya que los miembros del conjunto `nq` son estados equivalentes.
- La transición `Tr` se hace en el autómata generador `D`, utilizando un estado en `DQ` y un símbolo `s` en `DS`, con lo que se obtiene un estado destino e_d que es elemento de alguna clase de equivalencia en `NQ`, por lo que es necesario utilizar `Kde` para obtener la clase de tal estado.
- `Kde` sirve para obtener la clase de estados equivalentes que contiene a un miembro específico `q`; lo que se obtiene al obtener una lista de todas las clases de equivalencia que contienen a `q`. Como las clases de estados equivalentes definen una partición del conjunto de estados, es garantía de que la lista de clases es unitaria, por lo que la clase se obtiene con la primera clase de la lista.

Código 9.9: *Clase de equivalencia de un miembro*

```
; (Kde q LK) ↦ lista?
; q : cualquier
; LK : (listade lista?) -- lista de clases de equivalencia
(define Kde
  (λ (q LK) (car (filter (λ (K) (en? q K)) LK))))
```

■ **Ejemplo 9.13** Seleccionar las clases de `'q2`, `'q6` y `'q4` en la colección de clases `'((q6 q3 q2) (q1) (q5 q4))`.

```
> (define NQ '((q6 q3 q2) (q1) (q5 q4)))
> (Kde 'q2 NQ)
'(q6 q3 q2)
> (Kde 'q6 NQ)
'(q6 q3 q2)
> (Kde 'q4 NQ)
'(q5 q4)
>
```

■

Así las nuevas transiciones `NT` se logran haciendo:

```
NT↦ (append*
      (map (λ (nq)
             (map (λ (s) (tupla nq  s  (Tr (car nq) s))) NS))
      NQ))
```

9.4.2 Los nuevos estados

El nuevo estado inicial

El nuevo estado inicial `Nq0` es la clase de equivalencia que contiene al estado inicial en `D`.

```
Nq0 ↦ (car (filter (λ (k) (en? Dq0  k)) NQ))
```

Como la función `filter` devuelve una lista, es necesario obtener el primer elemento de esa lista, que contiene la clase de equivalencia del estado inicial.

Los nuevos estados aceptores

En el nuevo `AFD%`, un estado es aceptor si contiene al menos un estado aceptor del autómata generador `D`. El nuevo conjunto de estados aceptores `NA` es el subconjunto de nuevos estados cuya intersección con los aceptores en `D` no es vacia:

$$\texttt{NA} \mapsto \texttt{(filter-not (}\lambda\texttt{ (vacio?) (intersec k DA)) NQ)}$$

Reuniendo los nuevos elementos se crea un nuevo objeto de la clase `AFD%`: `(new AFD% [tuplaDef (list NQ NS NT Nq0 NA)])`

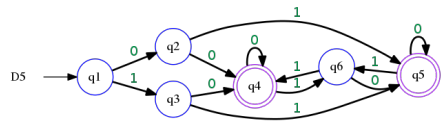


Figura 9.6: *Autómata generador* $\texttt{D5} \mapsto \langle \texttt{Q S T } q_0 \texttt{ A} \rangle$

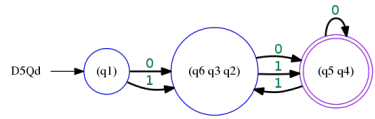
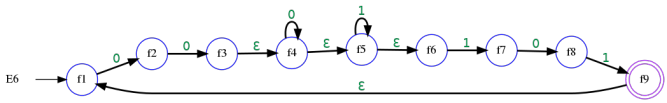


Figura 9.7: *Autómata resultante* $\texttt{D5Qd} \mapsto \langle \texttt{Q S T } q_0 \texttt{ A} \rangle$ *equivalente a* `D5`*, con clases de estados equivalentes.*

Código 9.10: *`AFD%` con clases de estados equivalentes*

```
1 ; (Q-dis D) ↦ AFD%?
2 ; D : AFD%?
3 (define Q-dis
4   (λ (D)
5     (let* ((NQ (Q-Keq D)) ; Los nuevos estados.
6            (NS (send D alfa)) ; El mismo alfabeto.
7            (NT (append* (map (λ (nq)
8                                (map (λ (s)
9                                       (tupla nq s (Kde (send D Tr (car nq) s) NQ)))
10                                     NS)) NQ)))
11           (Nq0 (car (filter (λ (k) (en? (send D eini) k)) NQ)))
12           (NA (filter-not  (λ (k) (vacio? (intersec k (send D acep)))) NQ)
      ))
13       (new AFD% [tuplaDef (list NQ NS NT Nq0 NA)])))
```

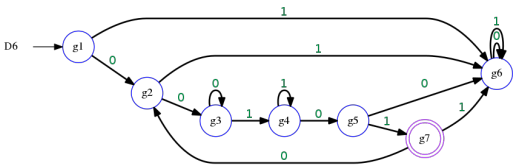
■ **Ejemplo 9.14** Obtenga el autómata reducido del AFE% identificado como E6, mostrado en la siguiente figura:



Para realizar la tarea se harán las siguientes operaciones:

1. Convertir el AFE% a un AFN% equivalente.
2. Convertir el autómata resultante anterior a un AFD% equivalente.
3. Obtener un nuevo AFD% que contenga únicamente los estados q_0-alcanzables.
4. Renombrar los estados del AFD% en el punto anterior.
5. Obtener un nuevo AFD% con clases de estados equivalentes, considerando el AFD% obtenido en el paso anterior.
6. Renombrar las clases de equivalencia
7. Verificar la igualdad de los lenguajes de ambos autómatas.

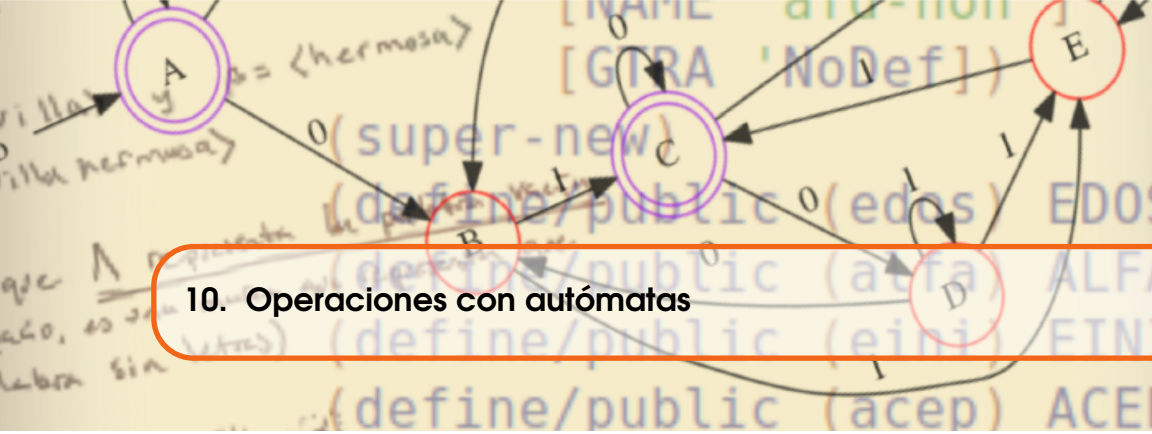
```
> (define E6 (new AFE% [tuplaDef (list '(f1 f2 f3 f4
f5 f6 f7 f8 f9) '(0 1) '((f1 0 f2) (f2 0 f3)
(f3 ε f4) (f4 0 f4) (f4 ε f5) (f5 1 f5) (f5 ε f6)
(f6 1 f7) (f7 0 f8) (f8 1 f9) (f9 ε f1)) 'f1 '(f9))]))
> (define D6 (Q-dis (Q-morf (Q-alc (afn->afd (afe->afn E6))) "g")))
> (define D6 (Q-morf D6 "g"))
> (c=? (send E6 nLeng 10) (send D6 nLeng 10))
#t
>
```



■

Código 9.11: *Reduce un AFE% a un AFD% mínimo*

```
1 ; (afe->afd AF #:pref [pref "d"]) ↦ AFD%
2 ; AF : (or AFE% AFN%)
3 ; pref : string? = "d- Prefijo para el nombre de estados.
4 (define afe->afd
5   (λ (AF  #:pref [pref "d"])
6     (let ((N (if (is-a? AFE% AF) (afe->afn AF) AF)))
7       (Q-morf (Q-dis (Q-alc (Q-morf (afn->afd N)))) pref))))
```



10. Operaciones con autómatas

10.1 Presentación

Una vez que se tiene ya una buena herramienta para crear y manipular autómatas finitos, es posible definir algunas operaciones para combinar autómatas, de modo que se puedan producir reconocedores para lenguajes más complejos.

Existe una estrecha relación entre los lenguajes (ver capítulo 5 en la página 89) y los autómatas finitos (capítulos 6, 7, 8). Los lenguajes son un conjuntos de palabras, escogidas de acuerdo a un criterio arbitrario y los autómatas son una herramienta que nos permite determinar si una palabra pertenece o no a cierto lenguaje, aunque hay otros criterios para generar lenguajes [HU79].

En este capítulo se establecerá de manera práctica la relación entre los lenguajes regulares y los autómatas finitos deterministas. Gracias a los patrones fijos que muestran los lenguajes regulares, es posible describir una manera de nombrar a los autómatas en función del lenguaje que aceptan; y a diseñar autómatas en función del nombre que se les ha asignado. Supondremos en lo que resta del capítulo, que los autómatas utilizados corresponden a la mínima expresión del autómata, es decir que contiene el mínimo número de estados, aunque esto no es una restricción.

10.2 Lenguajes regulares

Se dice que un lenguaje es **regular** si las palabras que contienen, ofrecen características que son el resultado de un criterio dwe construcción preestablecido, es decir, hay una regla para construir las palabras del lenguaje. La regla aludida se puede expresar de tal modo que no deje lugar a dudas acerca de las palabras que debe contener el lenguaje.

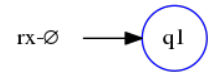
Las reglas que describen el contenido de los lenguajes regulares [LR] se llaman **expresiones regulares**. Con el fin de hacer diferencia entre las expresiones regulares y cualquier otro tipo de expresiones, se escribirá una expresión regular con el prefijo `rx-`, como en:

```
rx-10(a+b)*
```

El prefijo `rx-` indica que se trata de una expresión regular y el sufijo `10(a+b)*` expresa una regla de construcción para las palabras que debe aceptar el autómata, como se estudiará en las siguientes secciones.

10.2.1 El lenguaje vacío basado en autómatas

El lenguaje vacío (ver página 92) es el lenguaje que no contiene palabra alguna. Es posible construir un autómata finito determinista que acepte el mismo lenguaje (ver la figura 10.1). Observe que el autómata no tiene estados aceptores.




```
> (send rx-∅ nLeng)
'()
>
```

Figura 10.1: *Autómata finito determinista que modela el lenguaje vacio.*

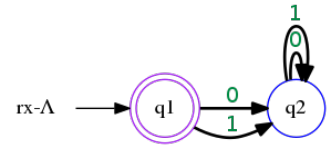
Una expresión regular para denotar al autómata que modela el lenguaje vacío es `rx-∅`, aprovechando que ∅ es un símbolo amplicamente conocido que representa al conjunto vacío. Así `rx-∅` se refiere al lenguaje vacío, o bien al autómata cuyo lenguaje es vacío.

De hecho, cualquier autómata donde el subconjunto de estados aceptores $A \mapsto$ `'()`, sirve para modelar el lenguaje vacío.

Formalmente `rx-∅` está definido con la tupla de definición

```
tuplaDef↤'((q1) () () 'q1 ()),
```

(a) *Autómata finito determinista con alfabeto vacío y estado inicial como estado aceptor, asociado con la* `rx-Λ`.



(b) *Autómata finito determinista con alfabeto* `'(0 1)` *y estado inicial como estado aceptor, asociado con la* `rx-Λ`.

Figura 10.2: *Autómatas finitos deterministas que modelan el mismo lenguaje, el lenguaje vacio.*

es decir, un autómata de un solo estado; con alfabeto vacío; un conjunto vacío de transiciones (ya que no tiene símbolos); un estado inicial y con un conjunto vacío de estados aceptiores.

10.2.2 Lenguaje unitario de un autómata

El autómata que acepta el lenguaje que contiene únicamente la palabra vacía es el lenguaje identidad (ver página 98). Se identifica usualmente por la expresión regular `rx-Λ` y puede ser modelado mediante algún autómata que acepte únicamente la palabra vacía.

En la figura 10.2 se muestran dos opciones para modelar el lenguaje que acepta únicamente la palabra vacía. El autómata de la izquierda es un autómata determinista con alfabeto vacío. De este modo `rx-Λ` representa el lenguaje que acepta una sola palabra, de modo que es un lenguaje unitario. El autómata a la derecha en la misma figura 10.2, está modelado con un autómata que reconoce el alfabeto `'(0 1)`, aunque no reconoce mas que la palabra vacía.

Si `S↤'(s)` es un alfabeto unitario, `rx-s` es una expresión regular que se refiere al autómata que acepta la palabra unitaria `'(s)`. El prefijo `rx-` indica que se trata de una expresión regular, y el resto indica el lenguaje aceptado por el autómata.

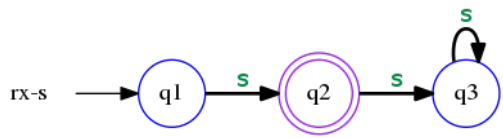
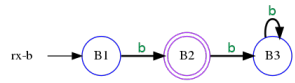


Figura 10.3: `rx-s` *denota el lenguaje unitario* `'((s))`*, se muestra un* `AFD%` *que acepta el mismo lenguaje.*

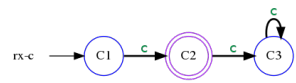
■ **Ejemplo 10.1** Si Sa ↦ '(a) y Sb ↦ '(b) son alfabetos unitarios, se pueden crear dos autómatas, uno para que acepte Sa y el otro que acepte unicamente Sb:

```
> (define rx-a (new AFD% [tuplaDef '((A1 A2 A3) (a) ((A1 a A2) (A2 a A3)
(A3 a A3)) A1 (A2))]))
> (send rx-a nLeng)
'((a))
> (define rx-b (new AFD% [tuplaDef '((B1 B2 B3) (b) ((B1 b B2) (B2 b B3)
(B3 b B3)) B1 (B2))]))
> (send rx-b nLeng)
'((b))
>
```

■



(a) *rx-b es una ER, donde b ↦ '((b))*



(b) *rx-c es una ER, donde c ↦ '((c))*

Figura 10.4: *Tanto rx-b como rx-c son expresiones regulares que identifican a los autómatas en 10.4a y 10.4b; a su vez denotan los lenguajes unitarios '((b)) y '((c)) respectivamente.*

10.3 Unión de autómatas

Si B ↦ ⟨Q S T q_0 A⟩ y C ↦ ⟨Q S T q_0 A⟩ son dos autómatas, con lenguajes (Leng B) ↦ '((b)) y (Leng C) ↦ '((c)), es posible unirlos para generar un AFE% (y eventualmente un AFD%) que acepte las palabras en el lenguaje de B o en el lenguaje de C.

Lo que se busca es un AFE% que permita aceptar las palabras en la unión del lenguaje de B con el lenguaje de C. Al analizar una palabra desde el estado inicial, el autómata generado aceptará una palabra, si se alcanza un estado aceptor en B o en C. El autómata generado se identifica mediante a expresión regular

rx-B+C

Para crear un AFE% que acepte la unión del lenguaje de B con el lenguaje de C:

1. **Se agrega un nuevo estado**. El nuevo conjunto de estados es

   NQ↤(agregar 'ii (union BQ CQ)),

2. **Se actualiza el alfabeto**. El nuevo alfabeto es

   NS↤(union BS CS),

ya que se debe garantizar el analisis de las palabras de ambos lenguajes.

3. **Se actualizan las transiciones**. El nuevo conjunto de transiciones, además de incluir las transiciones de ambos autómatas generadores, debe incluir ε-transiciones del nuevo estado inicial al inicio de cada autómata.

```
tib↤'(ii 'ε Bq0)
tic↤'(ii 'ε Cq0)
NT↤(union* BT CT (conj tib tic))
```

4. **Se actualiza el estado inicial**. El nuevo estado inicial `Nq0` es `'ii`, recién agregado al conjunto de estados.

```
Nq0↤'ii
```

5. **Actualizar el subconjunto de estados aceptores**. El nuevo subconjunto de estados aceptores será la unión de `BA` con `CA`

```
NA↤(union BA CA)
```

6. **Crear el nuevo autómata**. Finalmente se crea un nuevo `AFE%` con la información anterior:

```
(new AFE% [tuplaDef (list NQ NS Nq0 NT NA)])
```

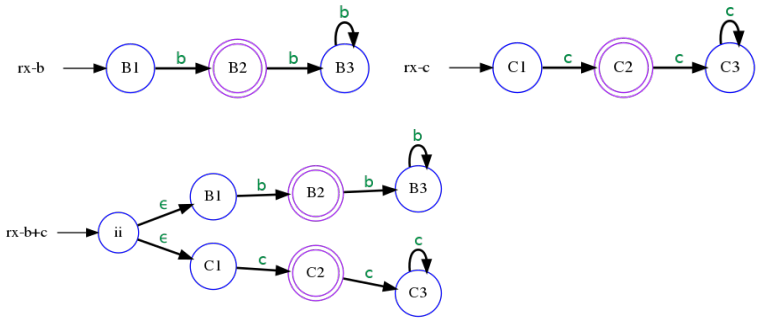
Si `rx-B` y `rx-B` son expresiones regulares regulares (expresiones que se refieren a conjuntos regulares de palabras) entonces:

```
rx-B+C ↦ (union (nLeng rx-B) (nLeng rx-C))
```

Código 10.1: *Unión de autómatas finitos*

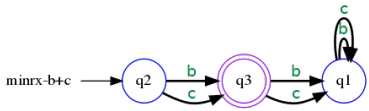
```
; (AF-union B C [pref]) ↦ (o AFD% AFE%)
; B : (o AFE% AFN% AFD%)
; C : (o AFE% AFN% AFD%)
; pref ="q": string? -- Un prefijo para renombrar estados.
(define AF-union
  (λ (B C [pref "q"])
    (let* ((B1 (Q-morf B "B"))
           (C1 (Q-morf C "C"))
           (tib (tupla 'ii 'e (send B1 eini)))
           (tic (tupla 'ii 'e (send C1 eini)))
           (NQ (agregar 'ii (union (send B1 edos)
                                   (send C1 edos))))
           (NS (union (send B1 alfa) (send C1 alfa)))
           (NT (union* (send B1 tran) (send C1 tran)
                       (conj tib tic)))
           (Nq0 'ii)
           (NA (union (send B1 acep) (send C1 acep)))
           (NE (new AFE%
                    [tuplaDef (list NQ NS NT Nq0 NA)])))
      (if (equal? pref "")
          NE
          (afe->afd NE #:pref pref)))))
```

■ **Ejemplo 10.2** Sean `rx-b` y `rx-c` los autómatas de la figura 10.4. Construya `rx-b+c`, un nuevo `AFE%` con la unión de ambos autómatas. El nuevo `AFE%` resultante está definido por el diagrama de transiciones:




```
> (define rx-b+c (AF-union rx-b rx-c ""))
> (send rx-b+c nLeng)
'((b) (c))
>
```

Con las herramientas de reducción de autómatas se puede obtener la mínima expresión para el mismo autómata. Al indicar el prefijo `"q"`, se indica hacer la conversión `afe->afd` prefiriendo cambiar el nombre de los estados a uno que inicie con `"q"`:




```
> (define minrx-b+c (AF-union rx-b rx-c "q"))
> (send minrx-b+c nLeng)
'((b) (c))
>
```

■

10.3.1 Unión extendida de autómatas

De manera similar que la unión extendida de conjuntos (página 42), es posible crear un procedimiento que tome cualquier cantidad de autómatas y genere un nuevo autómata que acepte el lenguaje de la unión extendida de los lenguajes generadores.

El procedimiento debe funcionar con una lista indeterminada de autómatas. Si la lista es vacía, el procedimiento debe retornar el autómata vacío `rx-∅`; pero si $B_1 \ldots B_k$ son autómatas, la unión extendida de ellos debe ser el nuevo `AFE%`:

rx-B1+...+Bk $\mapsto$ (AF-union B_k (AF-union ... (AF-union B_1 rx-Λ) ...))

Código 10.2: *Unión extendida de autómatas*

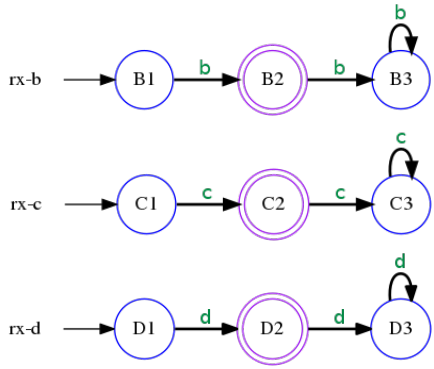
```
; (AF+ B1 ...+) ↦ AFD%
; B1 ...+ : (listade (or AFE% AFN% AFD%))
(define AF+
  (λ (#:pref  [pref ""] . LA)
    (letrec ([AF+aux
              (λ (LAF res)
                (if (vacio? LAF)
                  res
                  (AF+aux (cdr LAF)
                    (AF-union res (car LAF) pref))))])
      (if (vacio? LA)
          rx-∅ (AF+aux (cdr LA) (car LA))))))
```

Los argumentos tipo *rest* se utilizan en procedimientos en los que se requiere una lista indeterminada de argumentos, pero debe haber al menos un argumento obligatorio. La lista `x . LA` en la lista de argumentos de `AF+`

```
(define AF+ (λ (#:pref [pref "q"] x . LA) ...))
```

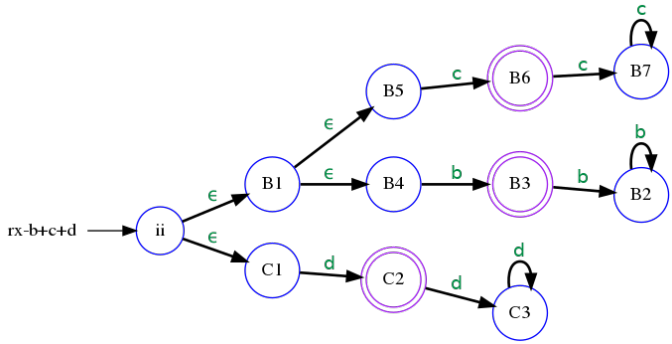
indica que debe proporcionarse un argumento obligatorio asociado con la literal `x` y una lista indeterminada de argumentos que se asocian con el identificador `LA`. En el código 10.2, el único argumento obligatorio es el argumeto clave `pref`, que tiene como valor por defecto la cadena vacía.

■ **Ejemplo 10.3** Sean `rx-b`, `rx-c` y `rx-d` los autómatas mostrados enseguida, que servirán como autómatas generadores:

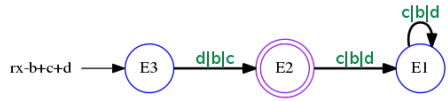


La unión extendida de los tres autómatas generadores es:

```
> (define rx-b+c+d (AF+ #:pref "" rx-b rx-c rx-d))
> rx-b+c+d
(object:AFE% ...)
```



Con el fin de mejorar la lectura del autómata, es posible reducir a su mínima expresión y resumir la información de las transiciones comunes:



■

10.4 Concatenación de autómatas

La concatenación de autómatas está directamente relacionada con la concatenación de lenguajes (página 97). Se requiere ahora construir un autómata que acepte las palabras en la concatenación de dos lenguajes. Cada uno de los lenguajes que sirven para hacer la concatenación está determinado por un autómata.

Sean `rx-B` $\mapsto \langle$`Q S T` q_0 `A`$\rangle$ y `rx-C` $\mapsto \langle$`Q S T` q_0 `A`$\rangle$ dos autómatas finitos que pueden ser deterministas o indeterministas, en los que para hacer referencia al lenguaje se utiliza el mismo identificador, así:

`rx-B` $\mapsto$ `(Leng B)`
`rx-C` $\mapsto$ `(Leng C)`

Se construye un nuevo `AFE%` denominado `rx-BC`, que eventualmente puede ser transformado en un `AFD%` y que acepte el lenguaje `(lconcat rx-B rx-C)`. El nuevo `AFE%` asociado con la expresión regular `rx-BC` se constituye del siguiente modo:

1. **El conjunto de estados**. El nuevo conjunto de estados `NQ`$\leftarrow\!\shortmid$`(union BQ CQ)`.
2. **El alfabeto**. El conjunto de símbolos se obtiene mediante `NS`$\leftarrow\!\shortmid$`(union BS CS)`.
3. **Las transiciones**. Cada palabra aceptada en `rx-BC` tiene prefijo aceptado por `rx-B` y sufijo aceptado por `rx-C`. De cada estado aceptor en `rx-B` se

crea una ε-transición al estado inicial en `C`. Así `tfb↤(map (λ (ef) ⟨ef 'e Cq0⟩) BA)`, entonces

`NT↤(union* BT CT tfb)`

4. **El estado inicial**. El estado inicial para el nuevo autómata es el estado inicial del autómata `B`, así `Nq0↤Bq0`
5. **Los aceptores**. El conjunto de estados aceptores son tomados del autómata `C`, de modo que `NA↤CA`.

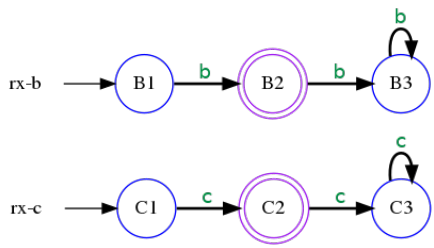
Con la información anterior se genera el `AFE%`:

`NE↤(new AFE% [tuplaDef (list NQ NS Nq0 NT NA)]).`

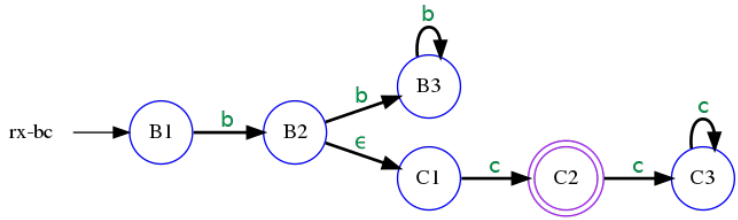
Código 10.3: *Concatenación de autómatas*

```
; (AF-concat B C [pref]) ↦ AFD%
; B : (or AFE% AFN% AFD%)
; C : (or AFE% AFN% AFD%)
; pref : string? = "q" -- Un prefijo para renombrar estados.
(define AF-concat
  (λ (B C [pref "q"])
    (let* ((B1 (Q-morf B "B"))
           (C1 (Q-morf C "C"))
           (NQ (union (send B1 edos) (send C1 edos)))
           (NS (union (send B1 alfa) (send C1 alfa)))
           (tfb (map (λ (ef)
                       (tupla ef 'e (send C1 eini)))
                     (send B1 acep)))
           (NT (union* (send B1 tran)
                       (send C1 tran)
                       tfb))
           (Nq0 (send B1 eini))
           (NA (send C1 acep))
           (NE (new AFE%
                    [tuplaDef (list NQ NS NT Nq0 NA)])))
      (if (equal? pref "")
          NE
          (afe->afd NE #:pref pref)))))
```

■ **Ejemplo 10.4** Sean `rx-b` y `rx-c` los autómatas mostrados

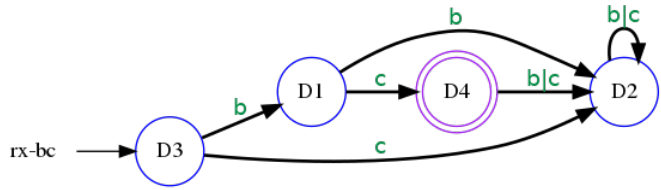


La concatenación de `rx-b` con `rx-c` es el nuevo `AFE%`:




```
> (define rx-bc (AF-concat rx-b rx-c ""))
> (send rx-bc nLeng)
'((b c))
>
```

que puede ser reducido a un `AFD%` de mínima expresión:



■

10.4.1 Concatenación extendida de autómatas

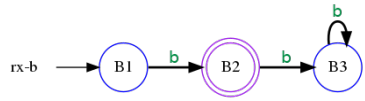
La concatenación extendida de autómatas brinda la posibilidad de concatenar por derecha un número indefinido de autómatas.

Observe que cuando no se proporciona algún autómata para ser concatenado, la respuesta es el autómata vacío `rx-∅`; y cuando se proporciona un único autómata, la concatenación extendida es ese mismo autómata.

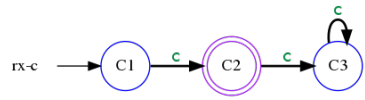
Código 10.4: *Concatenación extendida de autómatas*

```
1 ; (AF* B1 ...+) ↦ (or AFE% AFD%)
2 ; B1 ...+ : (listade (or AFE% AFN% AFD%))
3 (define AF*
4   (λ (#:pref [pref "q"] . LA)
5     (letrec ([AF*aux
6               (λ (LAF res)
7                 (if (vacio? LAF)
8                   res
9                   (AF*aux (cdr LAF)
10                    (AF-concat res (car LAF) pref)))
11                  )])
12       (if (vacio? LA)
13         rx-∅
14         (AF*aux (cdr LA) (car LA))))))
```

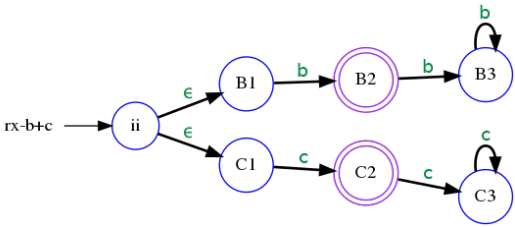
■ **Ejemplo 10.5** Sea `rx-b` el autómata que acepta el lenguaje `'((b))`,



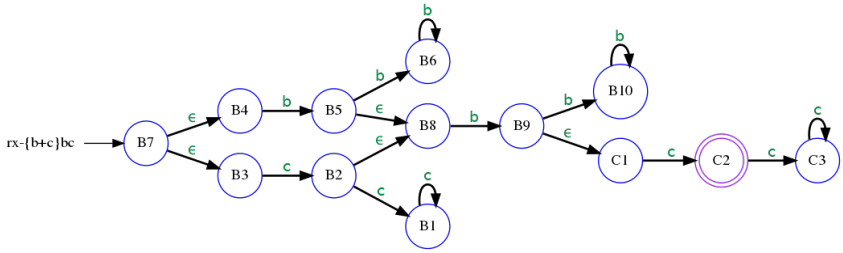
`rx-c` el autómata que acepta el lenguaje `'((c))`,



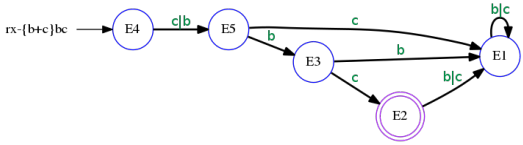
y `rx-b+c` el autómata que acepta el lenguaje `'((b) (c))`



`rx-{b+c}bc` es el lenguaje que contiene las palabras que inician con prefijo `'(b)` o `'(c)` y que terminan con sufijo `'(b c)`. `rx-{b+c}bc` es el autómata con transiciones nulas:



que puede ser reducido a un `AFD%` de expresión mínima:




```
> (define rx-<b+c>bc (AF* rx-b+c rx-b rx-c ""))
> (send rx-<b+c>bc nLeng)
'((b b c) (c b c))
>
```

■

10.5 Potencia de un autómata

La potencia de un autómata está estrechamente relacionada con la potencia de un lenguaje (ver página 100). El objetivo es crear una máquina de estados finitos que reconozca el lenguaje generado por la `n`-ésima potencia de un lenguaje, donde `n` es un número entero no negativo.

Observe que la potencia `0` de cualquier lenguaje es el lenguaje identidad (página 98), que es el lenguaje unitario que contiene a `pVacia` (página 74).

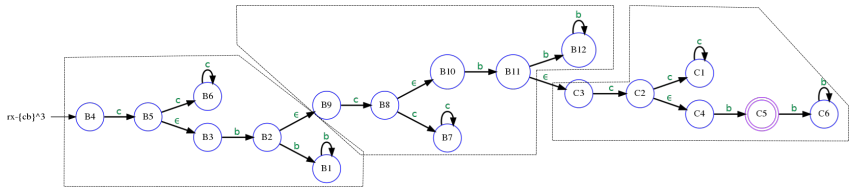
Código 10.5: *Potencia de un autómata finito*

```
; (AF-pot A n [pref] [res]) ↦ AFD%
; A : (or AFE% AFN% AFD%)
; n : numero-entero-no-negativo?
; pref : string = "q- Un prefijo para renombrar estados.
; res : AFD% -- El autómata resultante.
(define AF-pot
  (λ (A n [pref ""] [res A])
    (cond ((= n 0) rx-Λ)
          ((= n 1) res)
          (else (AF-pot A (- n 1)
                pref (AF-concat res A pref))))))
```

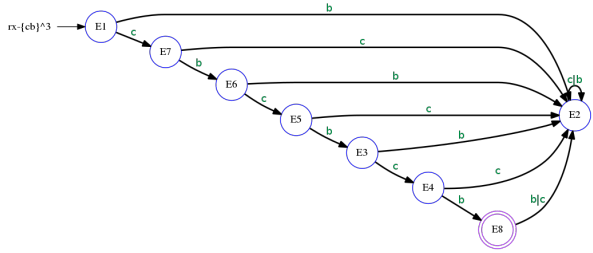
■ **Ejemplo 10.6** Calcule el autómata correspondiente a `rx-{cb}`^{3}.

El autómata `rx-{cb}`^{3} acepta la potencia `3` del lenguaje `'((c b))`, esto es `'((c b c b c b))`. Primero se hace la concatenación `rx-c` con `rx-b` y luego la tercera potencia.

```
> (define rx-<cb>^3 (AF-pot (AF* #:pref "" rx-c rx-b) 3 ""))
>
```



que puede ser reducido a un `AFD%` de expresión mínima:



■

10.6 Cerradura de Kleene para un autómata

La cerradura de Kleene para un autómata se puede nombrar como `AF-Kleene*` y es una operación que recibe un autómata `AF` que pudiera ser un `AFE%`, un `AFN%` o incluso un `AFD%`; y genera un `AFE%` (eventualmente un `AFD%` equivalente) que acepta

```
(union* lvacio (Leng (AF-pot AF 1))
               (Leng (AF-pot AF 2)) ...)
```

Convencionalmente la cerradura de Kleene aplicada a un lenguaje regular R, se denota como R^* y se define como:

$$\begin{aligned} R^* &= \Lambda \cup \Lambda R^1 \cup R^1 R^1 \cup R^2 R^1 \ldots \\ &= \Lambda \cup R^1 \cup R^2 \cup R^3 \ldots \\ &= \bigcup_{k=0}^{\infty} R^k \end{aligned}$$

La función `(AF-Kleene* AF)` genera un nuevo autómata, en cuyo lenguaje se encuentran las palabras que corresponden al lenguaje de la unión de todas las potencias de `AF`.

Para generar el autómata que modela el lenguaje de la cerradura de Kleene para un autómata generador `AF`:

1. **El conjunto de estados**. El nuevo autómata tendrá los mismos estados que el autómata generador:

   `NQ`$\leftarrow\!\!\shortmid$`AFQ`

2. **El alfabeto**. El nuevo autómata tendrá el mismo alfabeto del autómata `AF`

   `NS`$\leftarrow\!\!\shortmid$`AFS`

3. **Las transiciones**. El conjunto de transiciones tiene como base las mismas transiciones de `AF`, agregando transiciones nulas desde los estados aceptores al estado inicial. Para cada estado marcado como aceptor en `AF`, se crea una transición nula hacia el estado inicial y se agregan a las transiciones del autómata.

   `Tf`$\leftarrow\!\!\shortmid$`(map (`λ `(ea) (tupla ea '`ε `AFq`$_0$`)) AFA)`

   entonces

   `NT`$\leftarrow\!\!\shortmid$`(union AFA Tf)`

4. **El estado inicial**. Para el nuevo autómata, el estado inicial es el estado inicial de `AF`

   `Nq0`$\leftarrow\!\!\shortmid$`AFq`$_0$

5. **Los estados aceptores**. El conjunto de estados aceptores en el nuevo autómata serán los estados aceptores en AF, junto con el estado inicial si es necesario

$$\texttt{NA} \leftarrow (\texttt{agregar AF}q_0\ \texttt{AFA}).$$

6. Finalmente se genera un nuevo autómata indeterminista AFE%:

$$\texttt{NE} \leftarrow (\texttt{new AFE\% [tuplaDef (list NQ NS NT Nq0 NA)]})$$

Código 10.6: *Cerradura de Kleene para autómatas*

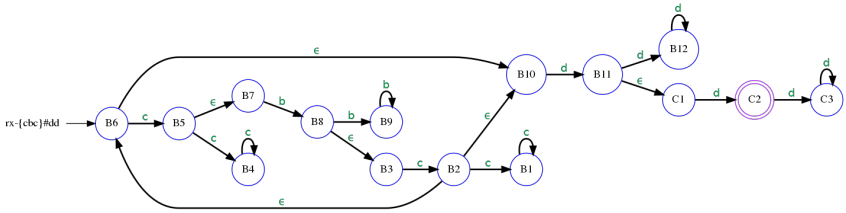
```
; (AF-Kleene* AF [pref]) ↦ (or AFE% AFD%)
; AF : (or AFE% AFN% AFD%)
; pref": string? = "q -- Un prefijo para renombrar estados.
(define AF-Kleene*
  (λ (AF [pref "K"])
    (let* ((NQ (send AF edos))
           (NS (send AF alfa))
           (Tf (map (λ (ea)
        (tupla ea 'e (send AF eini))) (send AF acep)))
           (NT (union (send AF tran) Tf))
           (Nq0 (send AF eini))
          (NA (agregar (send AF eini) (send AF acep)))
           (NE (new AFE% [tuplaDef
              (list NQ NS NT Nq0 NA)])))
      (if (equal? pref "")
          NE
          (afe->afd NE #:pref pref)))))
```

■ **Ejemplo 10.7** Calcular el autómata que corresponde a la expresión regular rx-{cbc}*dd.

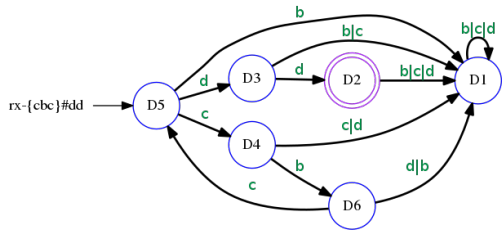
Antes de escribir el autómata, un breve análisis de la expresión regular rx-{cbc}*dd muestra que es un autómata que involucra las operaciones de concatenación extendida y la cerradura de Kleene. Después de hacer la cerradura de Kleene para rx-cbc, habrá que concatenarlos con rx-dd. Aunque rx-dd equivale a $\texttt{rx-d}^2$, que es la segunda potencia de rx-d, es más simple trabajarlos dentro de la misma concatenación extendida. Así

$$\texttt{rx-\{cbc\}*dd} \mapsto \texttt{(AF* (AF-Kleene* (AF* rx-c rx-b rx-c)) rx-d rx-d)}$$

```
> (define rx-<cbc>*dd (AF* #:pref "" (AF-Kleene* (AF* #:pref "" rx-c rx-b
    rx-c) "") rx-d rx-d))
>
```



que puede ser reducido a un AFD% de expresión mínima:



■

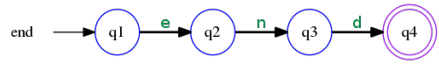
10.7 Aplicación

En esta sección se describe una de las más comunes aplicaciones para los autómatas finitos, se trata de un reconocedor de palabras clave [RPC]. Un RPC se utiliza comunmente en el análisis de códigos fuente para detectar palabras mal escritas, o construcciones gramaticales equivocadas.

Las palabras utilizadas en la construcción de programas de computadora, obedecen las reglas de construcción dictaminadas en el diseño del lenguaje; son precisamente esas reglas las que hacen que el lenguaje sea regular.

Supónga que se desea analizar un texto para determinar si una palabra clave está correctamente escrita, digamos las palabras "`begin`", "`end`", que son palabras clave que suelen tener los lenguajes de programación. Se puede construir un AFE% para reconocer exactamente esas palabras:

(a) *AFN% para reconocer la palabra "`begin`"*



(b) *AFN% para reconocer la palabra "`end`"*

Figura 10.5: *Generación de AFN% para reconocer palabras clave.*

De hecho es posible crear un procedimiento efectivo para construir un AFN% a partir de una palabra entrecomillada, tal como "`begin`", "`end`", etc.

El procedimiento `pal->afn` recibe una palabra entrecomillada `str` y devuelve un `AFE%`, para lo cual se require:

1. El número de caracteres, `n`↤`(string-length str)`. El número de caracteres se requiere para crear los `(+ n 1)` estados necesarios.
2. El nuevo conjunto de estados `NQ` es una lista de símbolos `'(`q_1 ... q_{n+1}`)`. La etiqueta para los estados es irrelevante, pues puede ser modificada mediante un homomorfismo de estados (página 181).
3. Los nuevos símbolos `NS` se toman del conjunto de letras que forman la palabra `str`. Observe que en `DrRacket`, las palabras unitarias entrecomilladas no son equivalentes a los símbolos, por ejemplo "a" no es lo mismo que `'a`, por lo que se requiere un procedimiento `str->alfa` que transforme una cadena de caracteres a un conjunto de símbolos, así por ejemplo `(str->alfa "bool")` ↦ `'(b o l)`.
4. Las transiciones `NT` se crean uniendo cada par de estados con el símbolo en secuencia. Como se observa en la figura 10.5.
5. El estado inicial `Nq0`↤`q1`
6. El conjunto de estados aceptores es unitario: `NA`↤`'(`q_{n+1}`)`
7. Finalmente se consigue un nuevo `AFN%` con

```
(new AFN% [tuplaDef (list NQ NS NT Nq0 NA)])
```

Código 10.7: *Obtiene un alfabeto de una cadena de caracteres*

```
; (str->simb str) ↦ (listade simbolo?)
; str : string?
(define str->simb
  (λ (str)
    (let* ((ls (map (λ (s) (format "~a" s)) (string->list str))))
      (map (λ (s) (if (equal? #f (string->number s))
                      (string->symbol s)
                      (string->number s))) ls))))

; (str->alfa str) ↦ (listade simbolo?)
; str : string?
(define str->alfa
  (λ (str)
    (alfP (str->simb str))))
```

■ **Ejemplo 10.8** Obtenga el alfabeto de la palabra "begin".

```
> (str->alfa "begin")
'(n i g e b)
>
```

■

Una vez que se tiene el alfabeto, se genera un `AFE%` que acepte una palabra entrecomillada. El conjunto de estados es fácilmente obtenido ya que se construye

una cadena de estados de longitud `(+ (long str) 1)`, donde `str` es la palabra entrecomillada, numerando los nuevos estados a partir del numeral `1` de prefijo `'q`, así `'q1` será el estado inicial y q_{n+1} será el único estado aceptor. Las transiciones se generan con cada letra de la palabra `str`.

Código 10.8: *`AFN%` a partir de una cadena de caracteres*

```
; (pal->afn str) ↦ AFN%
; str : string?
(define pal->afn
  (λ (str)
    (let* ((n (string-length str))
           (NQ (build-list (+ n 1)
                 (λ (i) (string->symbol
                          (string-append "q"
                        (number->string (+ i 1)))))))
           (NS (str->alfa str))
       (NT (map (λ (t s) (tupla (car t) s (last t)))
                  (map (λ (x y) (list x y))
                       (drop-right NQ 1) (cdr NQ))
                  (str->simb str)))
           (Nq0 (car NQ))
           (NA (conj (last NQ))))
      (new AFN% [tuplaDef (list NQ NS NT Nq0 NA)]))))
```

Es conveniente que el símbolo utilizado para identificar las transiciones nulas, sea un símbolo no utilizado en el alfabeto. En el siguiente ejemplo se ha utilizado `'`ϵ como el símbolo para las trancisiones nulas.

■ **Ejemplo 10.9** Obtenga un `AFN%` que acepte únicamente la palabra "`begin`" y muestre sus transiciones (figura 10.5a).

```
> (define begin (pal->afn "begin"))
> (send begin tran)
'((q1 b q2) (q2 e q3) (q3 g q4) (q4 i q5) (q5 n q6))
>
```

■

Finalmente la idea es un procedimiento que tome una lista de palabras y genere un `AFE%` con la unión de los `AFN%` de cada palabra de entrada.

```
> (define comp1 (apply AF+ (map (λ (w) (pal->afn w)) '("begin" "for"
  "while" "define" "end" "endf" "endw"))))
>
```

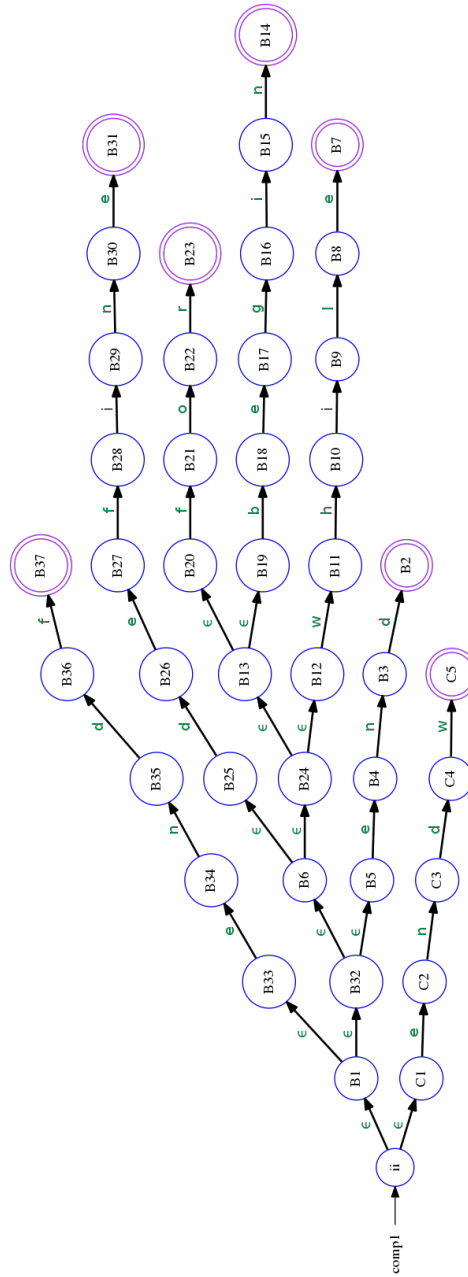
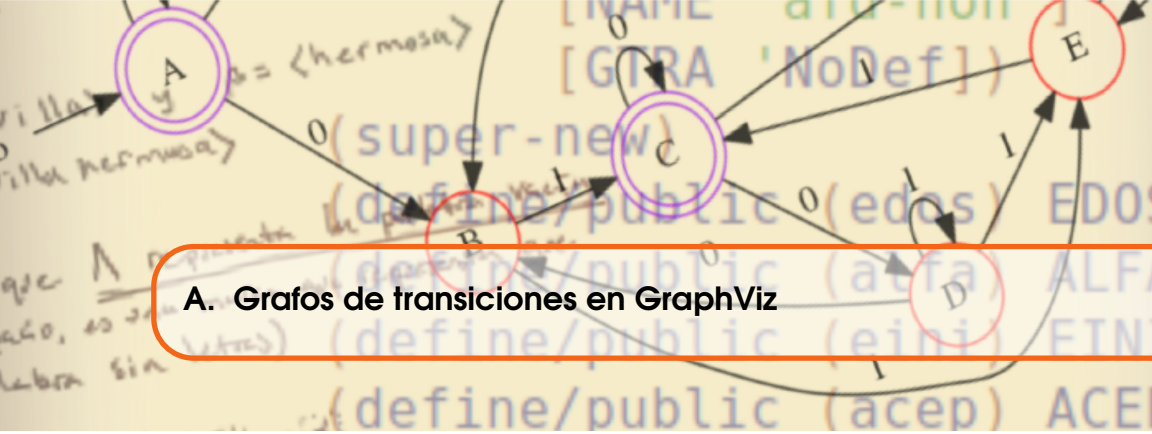


Figura 10.6: *AFE% que acepta el lenguaje '("begin" "for" "while" "define" "end" "endf" "endw").*



A. Grafos de transiciones en GraphViz

Todas las imágenes de diagramas de transición de los autómatas en este libro han sido generadas en GraphViz. GraphViz es un programa de computadora OpenSource, que sirve para visualizar la información estructural de los grafos. Actualmente puede ser descargado desde `http://www.graphviz.org/` para las plataformas de cómputo más comunes.

GraphViz toma la descripción de un grafo utilizando un lenguaje propio y construye una imagen con alguno de los algoritmos que tiene. Cada algoritmo ofrece una manera especial de dibujar el grafo. El resultado es una imagen que puede estar en alguno de una gran variedad de formatos, por ejemplo `PNG`, `PDF`, `SVG` y muchos otros.

GraphViz genera la imagen a partir de una descripción codificada en el lenguaje `DOT`, este código es analizado por un motor de transformación que puede ser por ejemplo `dot`, `neato` y `twopi`, de acuerdo con la siguiente expresión que debe ser leída por el sistema operativo:

```
$ dot -Tpng nom-grafo.gv -o nom-grafo.png
```

En la expresión anterior:

`dot` : Es el nombre del motor de construcción que se utiliza. GraphViz ha creado diferentes motores que generan grafos con diferentes aspectos.

`-Tpng` : Indica el tipo de archivo que se genera a la salida. Esta directiva genera archivos de tipo `PNG` (*Portable Networks Graphics*), pero `GraphViz` puede generar salidas de diferentes formatos.

`nom-grafo.gv` : Es el nombre de un archivo con el código fuente en el lenguaje `DOT`, que contiene la estructura general del grafo que se desea dibujar, por ejemplo:

```
digraph finite_state_machine {
  rankdir = LR;
  bgcolor = transparent;
  "a1" [shape = circle, color = blue];
  "a2" [shape = doublecircle, color = purple];
  "Inicio" [label = "A", shape = none];
  "Inicio" -> "a1";
  "a1" -> "a2" [color = "black", label = "a"];
  overlap=false
}
```

`-o` : Indica que se va a establecer un archivo de salida.

`nom.grafo.png` : Es el nombre del archivo de salida.

Para generar los grafos de transiciones desde `DrRacket`, es necesario crear una función que tome como entrada la información de un autómata finito y haga las siguientes tareas:

1. Cree y abra un archivo de texto para escritura. El nombre puede ser obtenido como valor dado por el usuario con un argumento clave. Suponiendo que "`nombre-grafo.gv`" es el nombre del archivo, este nombre debe ser asociado con un identificador:

   `nomgrafo`↤"`nombre-grafo.gv`"

2. Escriba en `nomgrafo` el código que se refiere a la parte invariante del grafo, por ejemplo:

   ```
   digraph finite_state_machine {
     rankdir = LR;
     bgcolor = transparent;
   ```

   Para escribir un texto en un archivo de texto, es necesario utilizar un puerto de salida y asociarlo con la escritura de un archivo de texto, la primitiva `open-output-file` asocia un nombre de archivo con una variable en `DrRacket`, especificando que debe ser un archivo de escritura y lo que debe hacer si el archivo ya existe.

3. Escriba en `nomgrafo` la información de cada nodo, considerando en primer lugar los nodos regulares, luego los nodos aceptores y al final un nodo especial que servirá como marca de `INICIO`.

   ```
   "a1" [shape = circle, color = blue];
   "a2" [shape = doublecircle, color = purple];
   "Inicio" [label = "A", shape = none];
   ```

4. Escriba en nomgrafo la información de las aristas, considerando en primer lugar la arista del nodo INICIO al nodo inicial del autómata; luego las aristas deben ser tomadas del conjunto de transiciones del autómata:

```
"Inicio" -> "a1";
"a1" -> "a2" [color = "black", label = "a"];
```

5. Finalmente se escribe la parte invariable del final del archivo y se cierra.

```
    overlap=false
}
```

Una vez que se tiene el archivo .gv, DrRacket debe enviarlo al motor DOT por medio de una solicitud al sistema operativo, para que pueda ser interpretado y procesado.

DrRacket puede interactuar con el sistema operativo al incluir la biblioteca de funciones racket/system, aunque también será necesaria la biblioteca que pertmite manipular imágenes 2htdp/image. Para incluir estas bibliotecas se debe agregar la solicitud al inicio del código fuente.

```
(require racket/system
         2htdp/image)
```

El siguiente código es una función que toma como entrada la información de un autómata y despliega una imagen de formato png del grafo de transiciones. Se invita al lector para personalizar el procedimiento.

```
(define dtran
  (λ (Q S T e0 A
        #:nom [nombre "afn"]
        #:motor [motor "dot"])
    ; Se prepara un archivo de texto para escritura
    (let* ((nomin (string-append nombre ".gv"))
           (nomout2 (string-append nombre ".png"))
           (out (open-output-file nomin #:mode 'text #:exists 'replace))
           )
      (fprintf out "digraph finite_state_machine {~n")
      (fprintf out "rankdir = LR;~n") ; preferencia de izq. a derecha.
      ; Prepara los nodos para ser codificados.
      (map
        (λ (e)  ; Se determina la forma de los nodos.
          (cond ((and (equal? e e0) ; Si es aceptor-inicial.
                      (en? e A))
                 (fprintf out "~s [shape = doublecircle,color=purple]; ~n"
                          (format "~s" e0)))
                ((equal? e e0) ; Si es inicial.
                 (fprintf out "~s [shape = circle,color=blue]; ~n"
                          (format "~s" e0) ))
                ((en? e A) ; Si es aceptor.
                 (fprintf out "~s [shape = doublecircle,color=purple];~n"
                          (format "~s" e)))
                (else ; Todos los demas son normales.
                 (fprintf out "~s [shape = circle,color=blue];~n"
                          (format "~s" e) ))))
        Q)
      (fprintf out "~s [shape = none];~n" "INICIO")
      ; Prepara las transiciones.
      (fprintf out "~s -> ~s; ~n" "INICIO" (format "~s" e0))
```

```
32      (for-each
33        (λ (t) ; Para cada transición.
34          (fprintf out "~s -> ~s [label = \"~s\", penwidth=2];~n"
35               (format "~s" (car t)); El nodo inicial.
36               (format "~s" (last t))  ; El nodo final.
37               (cadr t))); la etiqueta
38          T)
39      ; La parte invariante final.
40      (fprintf out "overlap=false~n")
41      (fprintf out "}~n" )
42      (close-output-port out)
43      ; Se prepara una solicitud al sistema operativo.
44      (system (string-append motor " -Tpng " nomin  " -o " nomout2))
45      ; Se dibuja la imagen como un bitmap.
46      (bitmap/file nomout2))))
```

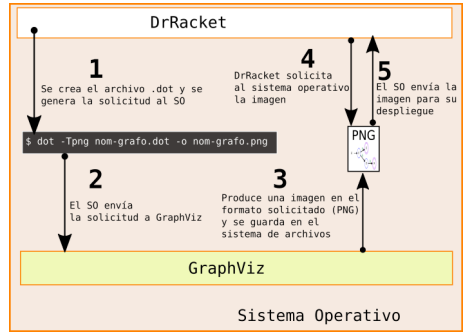


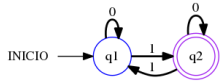
Figura A.1: *Proceso para utilizar* `GraphViz` *en la generación de los grafos de transiciones de los autómatas finitos.*

Suponga que se tiene un autómata que tiene la siguiente información y se desea crear una imagen para ver el diagrama de transiciones que corresponde:

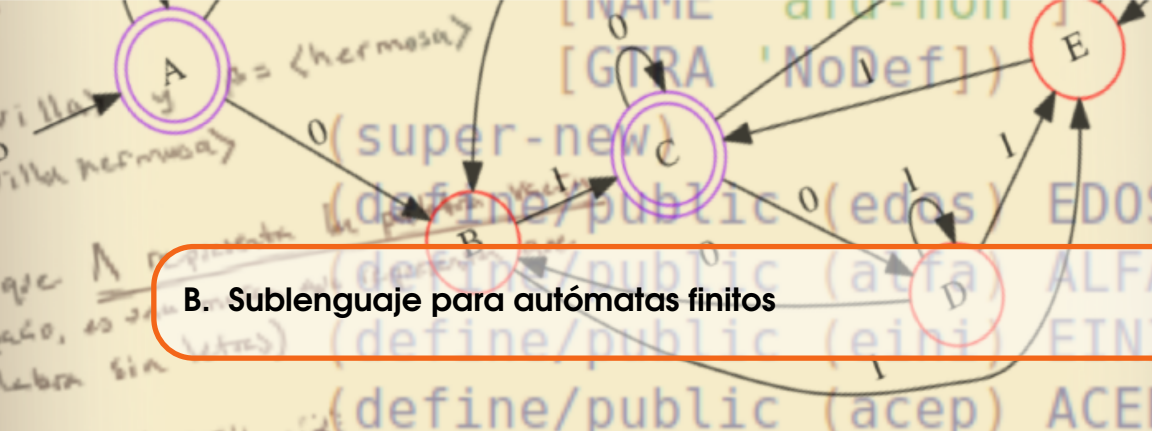
```
Q ↦ '(q1 q2)
S ↦ '(0 1)
T ↦ '((q1 1 q2) (q1 0 q1) (q2 1 q1) (q2 0 q2))
q0 ↦ 'q1
A ↦ '(q2)
```

Para generar la imagen, se invoca la función `dtran`:

```
> (dtran '(q1 q2) '(1 0) '((q1 1 q2) (q1 0 q1) (q2 1 q1) (q2 0 q2))
  'q1 '(q2) #:nom "Apnd1")
```




```
>
```



B. Sublenguaje para autómatas finitos

Código fuente completo que aparece en el libro. Se ha dividido el código de acuerdo a los capítulos en los que aparecen.

Código fuente para la inclusión de bibliotecas

```
#lang racket
(require racket/system
         2htdp/image)
```

Código fuente para el capítulo 1

```
; (neg p) ↦ booleano?
; p : booleano?
; Ej (neg #t) ↦ #f : La negación de #t significa #f.
(define neg
  (λ (p)
    (if p #f #t)))

; (y p q) ↦ booleano?
; p : booleano?
; q : booleano?;
; (y #t #f) ↦ #f
(define y
  (λ (p q)
    (if p q #f)))
```

```
15
16 ; (o p q) ↦ booleano?
17 ; p : booleano?
18 ; q : booleano?; (o #t #f) ↦ #t
19 (define o
20   (λ (p q)
21     (if p #t q)))
22
23 ; (y* LPred) ↦ booleano?
24 ; Lpred : booleano? ...
25 ; Ej>(y* #t #t #t #t #t #t) ↦ #t
26 (define y*
27   (λ LPred
28     (define y*aux
29       (λ (LP)
30         (cond ((empty? LP) #t)
31               ((car LP) (y*aux (cdr LP)))
32               (else #f))))
33     (if (empty? LPred)
34         #t
35         (y*aux LPred))
36     ))
37
38 ; (o* LPred) ↦ booleano?
39 ; LPred : booleano? ...
40 (define o*
41   (λ LPred
42     (define o*aux
43       (λ (LP)
44         (cond ((empty? LP) #f)
45               ((car LP) #t)
46               (else (o*aux (cdr LP))))))
47     (if (empty? LPred)
48         #f
49         (o*aux LPred))
50     ))
51
52 ; (ox p q) ↦ booleano?
53 ; p : booleano?
54 ; q : booleano?
55 (define ox
56   (λ (p q)
57     (if p (neg q) q)))
58
59 ; (->p q) ↦ booleano?
60 ; p : booleano?
61 ; q : booleano?
62 ; Ej. (->(= 4 4) (= 4 3)) ↦ #t.
63 (define ->
64   (λ (p q)
65     (if p q #t)))
66
67 ; (paraTodo proc lst ...+) ↦ lista?
68 ; proc : procedure? ; es el predicado
69 ; lst : lista? ; es la lista de dominios
70 (define paraTodo andmap)
71
72 ; (existeUn pred lst ...+) ↦ boolean?
73 ; pred : predicado?
74 ; lst : lista? ; requiere una o más listas de la misma cardinalidad
75 (define existeUn ormap)
```

Código fuente para el capítulo 2

```
76 ; Crea un conjunto con los elementos dados (igual que list).
77 (define conj list)
78
79 ; Se utiliza la lista vacía como el conjunto vacío.
80 ; Se crea un nombre adecuado para llamarlo.
81 (define vacio '()); Se define el conjunto vacío.
82
83 ; Se define un sinónimo para verificar el conjunto vacío.
84 ; (vacio? A) ↦ booleano?
85 ; A : conjunto?
86 (define vacio? empty?)
87
88 ; (en? e A) ↦ booleano?
89 ; e : cualquier
90 ; A : conjunto?
91 ; (en? 'a '(e d s a w)) ↦ #t
92 (define en?
93   (λ (e A)
94     (existeUn (λ (x) (equal? e x)) A)))
95
96 ; (card A [res]) ↦ numero?
97 ; A : conjunto?
98 ; res : numero? = 0 -- es un parámetro opcional, con valor 0 en omisión
99 (define card
100   (λ (A [res 0])
101     (if (vacio? A)
102         res
103         (card (cdr A) (+ res 1)))))
104
105 ; (unitario? C) ↦ booleano?
106 ; A : conjunto?
107 (define unitario?
108   (λ (A)
109     (= (card A) 1)))
110
111 ; (subc? A B) ↦ booleano?
112 ; A : conjunto?
113 ; B : conjunto?
114 (define subc?
115   (λ (A B)
116     (paraTodo (λ (a) (en? a B)) A)))
117
118
119 ; (subcPropio? A B) ↦ booleano?
120 ; A : conjunto?
121 ; B : conjunto?
122 (define subcPropio?
123   (λ (A B) ;<-- dos conjuntos en forma de lista
124     (y (subc? A B)
125        (existeUn (λ (b) (neg (en? b A))) B))))
126
127
128 ; (c=? A B) ↦ booleano?
129 ; A : conjunto?
130 ; B : conjunto?
131 (define c=?
132   (λ (A B)
133     (y (subc? A B)
134        (subc? B A))))
135
136
```

```
137 ; (agregar e A) ↦ conjunto?
138 ; e : cualquier
139 ; A : conjunto?
140 (define agregar
141   (λ (e A #:chk [chk #t])
142     (if (y chk (en? e A))
143         A
144         (cons e A))))
145
146
147 ; Unión de dos conjuntos
148 ; (union A B [res]) ↦ conjunto?
149 ; A : conjunto?
150 ; B : conjunto?
151 (define union
152   (λ (A B)
153     (cond ((empty? A) B) ; Caso 1.
154           ((en? (car A) B) (union (cdr A) B)) ; Caso 2.
155           (else (union (cdr A) (agregar (car A) B #:chk #f)))))); Caso 3.
156
157
158 ; Criterio de orden para símbolos.
159 ; (symb<? sy1 sy2) ↦ booleano?
160 ; sy1 : simbolo?
161 ; sy2 : simbolo?
162 (define symb<?
163   (λ (sy1 sy2)
164     (string<? (symbol->string sy1) (symbol->string sy2))))
165
166
167 ; Unión extendida.
168 ; (union* conj ...) ↦ conjunto?
169 ; conj : (listade conjunto?)
170 (define union*
171   (λ LC ; Una lista indefinida.
172     (if (vacio? LC)
173         '()
174         (sort (union*aux LC)
175               (λ (c1 c2) (cond ((y (symbol? c1)
176                                    (symbol? c2)) (symb<? c1 c2))
177                                ((y (number? c1)
178                                    (number? c2)) (< c1 c2))
179                                (else #f)))))))
180
181 ; Función auxiliar para la unión extendida.
182 ; (union*aux LC [res]) ↦ conjunto?
183 ; LC : (listade conjunto?)
184 ; res : conjunto? = '()
185 (define union*aux
186   (λ (LC [res '()])
187     (if (vacio? LC)
188         res
189         (union*aux (cdr LC) (union res (car LC))))))
190
191 ; Intersección de dos conjuntos.
192 ; (intersec A B) ↦ conjunto?
193 ; A : conjunto?
194 ; B : conjunto?
195 ; res : conjunto? = '()
196 (define intersec
197   (λ (A B)
198     (filter (λ (b) (en? b A)) B)))
199
```

```
; Intersección extendida de conjuntos.
; (intersec* A ...) ↦ conjunto?
; A : conjunto?
(define intersec*
  (λ LC
    (letrec ([intersec*aux
              (λ (LC [res (car LC)])
                (if (vacio? LC)
                    res
                    (intersec*aux (cdr LC)
                                  (intersec res (car LC)))))])
      (cond ((vacio? LC) vacio)
            ((vacio? (cdr LC)) (car LC))
            (else (intersec*aux LC))))))

; Diferencia entre dos conjuntos.
; (difc A B) ↦ conjunto?
; A : conjunto?
; B : conjunto?
(define difc
  (λ (A B)
    (filter-not (λ (a) (en? a B)) A)))

; Agrega un elemento al inicio de cada lista.
; (agregaEnCada e LC) ↦ (listade lista?)
; e : cualquier
; LC : (listade lista?)
(define agregaEnCada
  (λ (e LC)
    (map (λ (C) (agregar e C)) LC)))

; Conjunto potencia de un conjunto.
; (cPot A) ↦ (listade conjunto?)
; A : lista
(define cPot
  (λ (A [res '(())])
    (if (empty? A)
        res
        (cPot (cdr A) (append res (agregaEnCada (car A) res))))))

; Crea una tupla con una lista indeterminada de elementos.
; (tupla lst-elem ... ) ↦ lista?
; lst-elem : cualquier...
(define tupla
  (λ lst ; Es una lista indeterminada de elementos.
    lst )) ; Devuelve los valores en una lista determinada.

; Verifica que el argumento sea una tupla.
; (tupla? t) ↦ booleano?
; t : cualquier
(define tupla?
  (λ (t)
    (list? t)))

; Calcula la longitud de una tupla.
; (tlong t) ↦ numero-entero-no-negativo?
; t : tupla?
(define tlong
  (λ (t)
    (card t)))
```

```
260 ; Verifica la tupla vacía.
261 ; (tvacia? t) ↦ booleano?
262 ; t : lista
263 (define tvacia?
264   (λ (t)
265     (and (tupla? t) (tvacia? t))))
266
267 ; Verifica la tupla unitaria.
268 ; (tunit? t) ↦ booleano?
269 ; t : lista?
270 (define tunit?
271   (λ (t)
272     (and (tupla? t) (= (tlong t) 1))))
273
274 ; Calcula el producto cartesiano de dos conjuntos
275 ; (pCart A B) ↦ conjuntoDe/par?
276 ; A : conjunto?
277 ; B : conjunto?
278 (define pCart
279   (λ (A B)
280     (append* (map (λ (a)
281                     (map (λ (b)
282                            (cons a (if (tupla? b) b (tupla b))))
283                          B))
284                   A))))
285
286 ; Calcula el producto cartesiano de cero o más conjuntos.
287 ; (pCart* lst ...) ↦ (listade lista?)
288 ; lst : lista?
289 (define pCart*
290   (λ LC
291     ;--------------------------
292     (define pCart*aux
293       (λ (LC [res '(())])
294         (if (vacio? LC)
295             res
296             (pCart*aux (cdr LC) (pCart res (car LC))))))
297     ;--------------------------
298     (if (vacio? LC)
299         '()
300         (pCart*aux LC))))
301
302
303 ;; Verifica que R sea una relación.
304 ; (relacion? R A [B A]) ↦ booleano?
305 ; R : (listade par?)
306 ; A : conjunto?
307 ; B : conjunto? = A
308 (define relacion?
309   (λ (R A [B A]) ; R : A ↦ B
310     (subc? R (pCart A B)))) ; si R ⊆ A × B
311
312 ; Calcula el dominio de una relación.
313 ; (Dom R) ↦ conjunto?
314 ; R : (listade tupla?)
315 (define Dom
316   (λ (R)
317     (let ((dom (remove-duplicates (map (λ (t) (drop-right t 1)) R))))
318       (if (paraTodo (λ (d) (tunit? d)) dom)
319           (map (λ (x) (car x)) dom)
320           dom))))
321
322
```

```
323 ; Calcula el rango de una relación.
324 ; (Ran R) ↦ conjunto?
325 ; R : (listade tupla?)
326 (define Ran
327   (λ (R)
328     (remove-duplicates (map (λ (t) (car (take-right t 1))) R))))
329
330 ; Calcula la imagen de un elemento en una relación.
331 ; (Im a R) ↦ conjunto?
332 ; a : cualquier
333 ; R : conjuntoDe/par?
334 (define Im
335   (λ (a R)
336     (let ((eDom (if (list? a) a (list a))))
337       (filter (λ (c) (en? (append eDom (list c)) R)) (Ran R)))))
338
339 ; Calcula la imagen de un conjunto de elementos en una relación.
340 ; (Im* SD R) ↦ conjunto?
341 ; SD : conjunto?
342 ; R : (listade par?)
343 (define Im*
344   (λ (SD R) ; SD es un subconjunto del dominio de R
345     (apply union* (map (λ (e) (Im e R)) SD))))
346
347 ; Verifica si es una función.
348 ; (funcion? lsts) ↦ booleano?
349 ; lsts : (listade par?)
350 (define funcion?
351   (λ (R [dom (Dom R)])
352     (let ((imgns (map (λ (e) (Im e R)) dom)));<la imagen de cada elem
353       (paraTodo (λ (C) (unitario? C)) imgns))))
354
355 ; Evalúa un elemento en una función.
356 ; (evalf e F) ↦ cualquier
357 ; e : cualquier
358 ; F : (listade tupla?)
359 (define evalf
360   (λ (e F)
361     (let ((ev (Im e F)))
362       (if (vacio? ev)
363           (error "Error en los argumentos de la funcion")
364           (car (Im e F))))))
```

Código fuente para el capítulo 3

```
365 ; se define nulo como el símbolo 'ε
366 (define nulo 'ε)
367
368 ; Verifica que el argumento sea el símbolo nulo.
369 ; (nulo? s) ↦ booleano?
370 ; s : cualquiera?
371 (define nulo?
372   (λ (s)
373     (equal? s nulo)))
374
375 ; Verifica que el argumento sea un símbolo válido.
376 ; (simbolo? s) ↦ boolean?
377 ; s : cualquier
378 (define simbolo?
379   (λ (s)
380     (or (number? s)
381         (symbol? s)
382         (string? s)
383         (boolean? s))))
384
385 ; Genera un alfabeto de símbolos.
386 ; (alf S ...) ↦ (or (listade simbolo?) #f)
387 ; S ... : cualquiera
388 (define alf
389   (λ S
390     (if (paraTodo (λ (s) (simbolo? s)) S) S #\f)))
391
392 ; Define el alfabeto vacío como un conjunto sin símbolos.
393 (define alfaVacio '())
394
395 ; Verifica que el argumento sea el conjunto vacío.
396 ; (alfaVacio? S) ↦ boolean?
397 ; S : lista?
398 (define alfaVacio? empty?)
399
400 ; Verifica que el argumento sea un alfabeto.
401 ; (alf? S) ↦ boolean?
402 ; S : lista?
403 (define alf?
404   (λ (S)
405     (o (alfaVacio? S) ; puede ser el alfabeto vacío o puede ser
406        (and (list? S) ; ... una lista
407             (< (card S) +inf.0) ; ... finita
408             (paraTodo (λ (s) (simbolo? s)) S))))) ; ... de símbolos.
409
410 ; Verifica que un símbolo esté en un alfabeto.
411 ; (enS? s S) ↦ booleano?
412 ; s : simbolo?
413 ; S : alf?
414 (define enS?
415   (λ (s S)
416     (o (equal? s nulo)
417        (en? s S))))
418
```

```
; Verifica que un alfabeto sea subalfabeto de otro alfabeto.
; (subalf? S1 S2) ↦ booleano?
; S1 : alf?
; S2 : alf?
(define subalf?
  (λ (S1 S2)
    (paraTodo (λ (s) (enS? s S2)) S1)))

; Verifica que dos alfabetos sean iguales.
; (S=? S1 S2) ↦ booleano?
; S1 : lista?
; S2 : lista?
(define S=?
  (λ (S1 S2)
    (y (subalf? S1 S2) (subalf? S2 S1))))
```

Código fuente para el capítulo 4

```
; Genera una palabra con una secuencia indeterminada de símbolos.
;(pal S ...) ↦ pal?
; S : simbolo
(define pal
  (λ S
    (if (paraTodo (λ (s) (simbolo? s)) S) S #f)))

; Verifica que el argumento sea una palabra.
;(pal? w) ↦ booleano?
; w : cualquier
(define pal?
  (λ (w)
    (if (y (list? w)
           (paraTodo (λ (s) (simbolo? s)) w)))))

; Verifica que una palabra pertenezca a un alfabeto.
; (penS? w S) ↦ booleano?
; w : pal?
; S : alf?
(define penS?
  (λ (w [S w])
    (y  (list? w) ; w es una lista
        (paraTodo (λ (s) (en? s S)) w))))

; Calcula la longitud de una palabra.
; (long w [res]) ↦ numero?
; w : pal?
; res : numero? = 0 -- Es un argumento opcional, con valor 0 en omisión.
(define long
  (λ (w [res 0])
    (cond ((vacio? w) res)
          ((equal? (car w) nulo) (card (cdr w) res))
          (else (card (cdr w) (+ res 1))))))


; Define la palabra vacía como una palabra sin símbolos.
(define pVacia '())

; Verifica que el argumento sea la palabra vacía.
; (pVacia? w) ↦ booleano?
; w : cualquiera
(define pVacia?
  (λ (w)
    (o (equal? w pVacia) (= (long w) 0))))

; Crea una palabra unitaria válida.
; (punit s) ↦ pal?
; s : simbolo?
(define punit
  (λ (s)
    (if (y (simbolo? s) (neg (equal? s nulo)))
      (tupla s)
      'ERR-noSimbolo-o-nulo)))

```

```
489 ; Calcula el alfabeto generador de una palabra.
490 ; (Sgen w [res '()]) ↦ conjunto?
491 ; w : pal?
492 ; res : conjunto = '()
493 (define Sgen
494   (λ (w [res '()])
495     (cond ((pVacia? w) res)
496           ((en? (car w) res) (Sgen (cdr w) res))
497           (else (Sgen (cdr w) (agregar (car w) res #:chk #f))))))
498
499 ; Verifica que dos palabras sean iguales.
500 ; (p=? v w) ↦ booleano?
501 ; v : pal?
502 ; w : pal?
503 (define p=?
504   (λ (v w)
505     (equal? v w)))
506
507 ; Verifica que dos palabras sean diferentes.
508 ; (p!=? v w) ↦ booleano?
509 ; v : pal?
510 ; w : pal?
511 (define p!=?
512   (λ (v w)
513     (neg (p=? v w))))
514
515 ; Anexa una lista indeterminada de palabras
516 ; (*p ps ...) ↦ pal?
517 ; ps : pal? ...
518 (define *p
519   (λ ps
520     (append* ps)))
521
522 ; Escribe un símbolo en un extremo de la palabra.
523 ; (escribe s w [#:der #t]) ↦ pal?
524 ; s : simbolo?
525 ; w : pal?
526 ; der : booleano? = #t
527 (define escribe
528   (λ (s w #:der [der #t])
529     (cond ((nulo? s) w)
530           (der (concat w (punit s)))
531           (else (concat (punit s) w)))))
532
533 ; Genera la palabra inversa.
534 ; (pInv w) ↦ pal?
535 ; w : pal?
536 (define pInv reverse)
537
538 ; Calcula n-ésima la potencia de una palabra.
539 ; (**p w n [res]) ↦ pal?
540 ; w : pal?
541 ; n : numero-entero-no-neg
542 ; res : pal? = '()
543 (define **p
544   (λ (w n [res '()])
545     (if (= n 0)
546         res
547         (**p w (- n 1) (*p res w)))))
548
```

```
549 ; Verifica que una palabra sea prefijo de otra palabra.
550 ; (prefijo? a w) ↦ booleano?
551 ; a : pal?
552 ; w : pal?
553 (define prefijo?
554   (λ (a w)
555     (y (<= (long a) (long w))
556        (p=? a (take w (long a))))))
557
558 ; Calcula todos los prefijos de una palabra.
559 ; (prefijos w) ↦ (listade lista?)
560 ; w : lista?
561 (define prefijos
562   (λ (w)
563     (map (λ (n) (take w n)) (build-list (+ 1 (long w)) values))))
564
565 ; Verifica que una palabra sea sufijo de otra palabra.
566 ; (sufijo? c w) ↦ booleano?
567 ; c : pal?
568 ; w : pal?
569 (define sufijo?
570   (λ (c w)
571     (y (<= (long c) (long w))
572        (p=? c (take-right w (long c))))))
573
574 ; Calcula todos los sufijos de una palabra.
575 ; (sufijos w) ↦ (listade lista?)
576 ; w : lista?
577 (define sufijos
578   (λ (w)
579     (map (λ (n) (drop w n))  (build-list (+ 1 (long w)) values))))
580
581 ; Verifica que una palabra sea subpalabra de otra palabra.
582 ; (subpalabra? z w) ↦ booleano?
583 ; z : pal?
584 ; w : pal?
585 (define subpalabra?
586   (λ (z w)
587     (existeUn (λ (pref) (sufijo? z pref)) (prefijos w))))
588
589 ; Calcula todas las subpalabras de una palabra.
590 ; (subpalabras w) ↦ (listade lista?)
591 ; w : lista?
592 (define subpalabras
593   (λ (w)
594     (apply union* (map (λ (p) (sufijos p)) (prefijos w)))))
```

Código fuente para el capítulo 5

```
595 ; Verifica que el argumento sea un lenguaje.
596 ; (lenguaje? L) ↦ booleano?
597 ; L : (listade lista?)
598 (define lenguaje?
599   (λ (L)
600     (paraTodo (λ (w) (penS? w)) L)))
601
602 ; Calcula el alfabeto generador de un lenguaje.
603 ; (SgenL L) ↦ lenguaje
604 ; L : (listade pal?)
605 (define SgenL
606   (λ (L)
607     (apply union* (map (λ (w) (Sgen w)) L))))
608
609 ; El lenguaje vacío.
610 (define lvacio '())
611
612 ; Verifica que el argumento sea el lenguaje vacío.
613 ; (lvacio? L) ↦ booleano?
614 ; L : (listade pal?)
615 (define lvacio?
616   (λ (L)
617     (equal? L lvacio)))
618
619 ; Genera el lenguaje unitario de un alfabeto.
620 ; (lunit S) ↦ lenguaje
621 ; S : alf?
622 (define lunit
623   (λ (S)
624     (if (alf? S)
625       (map (λ (s) (punit s)) S)
626       'ERR-no-simb)))
627
628 ; Calcula la cardinalidad de un lenguaje.
629 ; (lcard L) ↦ numero-entero-no-negativo?
630 ; L : (listade lista?)
631 (define lcard card)
632
633 ; Verifica que una palabra pertenezca a un lenguaje.
634 ; (enl? w l) ↦ booleano?
635 ; w : pal?
636 ; L : lenguaje
637 (define enl?
638   (λ (w L)
639     (existeUn (λ (p) (p=? p w)) L)))
640
641 ; Verifica que un lenguajes sea un sublenguaje de otro.
642 ; (subl? L M) ↦ booleano?
643 ; L : (listade lista?)
644 ; M : (listade lista?)
645 (define subl?
646   (λ (L M)
647     (paraTodo (λ (w) (enl? w M)) L)))
648
```

```
649 ; Verifica que dos lenguajes sean iguales.
650 ; (l=? L M) ↦ boolean?
651 ; L : (listade lista?)
652 ; M : (listade lista?)
653 (define l=?
654   (λ (L M)
655     (y (subl? L M) (subl? M L))))
656
657 ; Calcula la concatenación de dos lenguajes.
658 ; (*l L M) ↦ lenguaje
659 ; L : lenguaje?
660 ; M : lenguaje?
661 (define *l
662   (λ (L M)
663     (append* (map (λ (a) (map (λ (b) (*p a b)) M)) L))))
664
665 ; El lenguaje identidad
666 (define lid '(()))
667
668 ; Verifica que un lenguaje sea el lenguaje identidad.
669 ; (lId? L) ↦ booleano?
670 ; L : (listade lista?)
671 (define lid?
672   (λ (L)
673     (l=? L lid)))
674
675 ; Calcula la concatenación extendida de lenguajes.
676 ; (*L LLs ...+) ↦ lenguaje?
677 ; LLs : (listade lenguaje?)
678 (define *L
679   (λ LLs
680     (letrec ([lccat-aux  (λ (LLs [res lid])
681                            (if (lvacio? LLs)
682                                res
683                                (lccat-aux (cdr LLs)
684                                           (*l res (car LLs))))))])
685       (if (lvacio? LLs)
686           lvacio
687           (lccat-aux LLs)))))
688
689 ; Calcula la n-ésima potencia de un lenguaje.
690 ; (**l L n) ↦ lenguaje?
691 ; M : lenguaje?
692 ; n : numero-entero-no-negativo?
693 (define **l
694   (λ (L n [res lid])
695     (if (= n 0)
696         res
697         (**l L (- n 1) (*l res L)))))
698
699 ; Calcula la cerradura finita de Kleene para un alfabeto.
700 ; (nKleene* S [n]) ↦ lenguaje
701 ; S : alf?
702 ; n : numero-entero-no-negativo?
703 (define nKleene*
704   (λ (S [n 10])
705     (let ((L    (lunit S)) ; El lenguaje unitario de S.
706           (pots (build-list (+ n 1) values))) ; Lista '(0 1 ... n).
707       (append* (map (λ (i) (**l M i)) pots)))))
708
```

```
; Verifica que una palabra pertenezca a la cerradura finita de Kleene.
; (Kleene*? x S) ↦ booleano?
; x : (or pal? lenguaje?)
; S : alf?
(define Kleene*?
  (λ (x S)
    (cond ((penS? x S) #t)
          ((y (lenguaje? x) (paraTodo (λ (w) (penS? w S)) x)) #t)
          (else #f))))
```

Código fuente para el capítulo 6

```
718 ; Clase para los autómatas finitos deterministas.
719 ; Requiere tuplaDef : 5-tupla como constructor.
720 (define AFD%
721   (class object%
722     (init tuplaDef)
723     (field [Q  (list-ref tuplaDef 0)] ; Los estados.
724            [S  (list-ref tuplaDef 1)] ; El alfabeto.
725            [T  (list-ref tuplaDef 2)] ; Las transiciones.
726            [q0 (list-ref tuplaDef 3)] ; El estado inicial.
727            [A  (list-ref tuplaDef 4)]) ; Los estados aceptores.
728     (super-new)
729     ;----------
730     (define/public (edos) Q)
731     (define/public (alfa) S)
732     (define/public (tran) T)
733     (define/public (eini) q0)
734     (define/public (acep) A)
735
736     (define/public (Tr q s) (evalf (tupla q s) T))
737     (define/public (Tr* q w)
738       (if (pVacia? w)
739           q
740           (Tr* (Tr q (car w)) (cdr w))))
741     (define/public (acepta? w) (en? (Tr* q0 w) A))
742     (define/public (enLeng? w) (acepta? w))
743     (define/public (nLeng [n 10])
744       (if (< n 10)
745          (filter (λ (w) (acepta? w)) (nKleene* S n))
746          (append (filter (λ (w) (acepta? w)) (nKleene* S 10))
747                  (list '...)))))))
```

Código fuente para el capítulo 7

```
749 ; Clase para los autómatas finitos indeterminístas.
750 ; Requiere tuplaDef : 5-tupla como constructor.
751 (define AFN%
752   (class object%
753     (init tuplaDef)
754     (field [Q  (list-ref tuplaDef 0)] ; Los estados.
755            [S  (list-ref tuplaDef 1)] ; El alfabeto.
756            [T  (list-ref tuplaDef 2)] ; Las transiciones.
757            [q0 (list-ref tuplaDef 3)] ; El estado inicial.
758            [A  (list-ref tuplaDef 4)]) ; Los estados aceptores.
759     (super-new)
760     ;----------
761     (define/public (edos) Q)
762     (define/public (alfa) S)
763     (define/public (tran) T)
764     (define/public (eini) q0)
765     (define/public (acep) A)
766     (define/public (Tr q s) (Im (tupla q s) T))
767     (define/public (Tr* q w)
768       (if (pVacia? w)
769           (tupla q)
770           (apply union* (map (λ (es) (Tr* es (cdr w))) (Tr q (car w))))))
771     (define/public (Tr+ P s)
772       (apply union* (map (λ (p) (Tr p s)) P)))
773     (define/public (acepta? w) (existeUn (λ (qf) (en? qf A)) (Tr* q0 w)))
774     (define/public (enLeng? w) (acepta? w))
775     (define/public (nLeng [n 10])
776       (if (< n 10)
777          (filter (λ (w) (acepta? w)) (nKleene* S n))
778          (append (filter (λ (w) (acepta? w)) (nKleene* S 10))
779                  (list '...))))))
```

```
781 ; Transforma un afn en un afd.
782 ; (afn->afd N) ↦ AFD%
783 ; N : AFN%
784 (define afn->afd
785   (λ (N) ; un afn
786     (let* ((DQ (map (λ (q) (agregar q '())) (send N edos)))
787            (DS (send N alfa))
788            (DT (apply union*
789                 (map (λ (q)
790                    (map (λ (s) (list q s (send N Tr+ q s)))
791                         DS))
792                      DQ)))
793            (X1 (letrec ([defTQ (λ (Q T)
794                 (let ((DifC (difConjuntos (Ran T) Q)))
795                   (if (vacio? DifC)
796                      (list Q T)
797                      (defTQ (union Q DifC)
798                             (union T (apply union*
799                                (map (λ (q)
800                                   (map (λ (s)
801                                          (list q s (send N Tr+ q s)))
802                                        DS))
803                                 DifC))))) ))])
804                  (defTQ DQ DT)))
805            (DQ (car X1))
806            (DT (cadr X1))
807            (Dq0 (car (filter (λ (q) (y (unitario? q)
808                                        (en? (send N eini) q)))
809                              DQ)))
810            (DA (filter-not
811                   (λ (q) (vacio? (intersec q (send N acep))))
812                  DQ)))
813       (new AFD% [tuplaDef (list DQ DS DT Dq0 DA)])))))
814
815 ; Verifica que dos autómatas sean equivalentes.
816 ; (nAF=? AF1 AF2 [n]) ↦ booleano?
817 ; AF1 : (or AFD% AFN% AFE%)
818 ; AF2 : (or AFD% AFN% AFE%)
819 ; n : numero-entero-positivo? = 10
820 (define nAF=?
821   (λ (AF1 AF2 [n 10])
822     (l=? (send AF1 nLeng n) (send AF2 nLeng n))))
```

Código fuente para el capítulo 8

```
824 ; Clase para los autómatas finitos con ε-transiciones.
825 ; Requiere tuplaDef : 5-tupla como constructor.
826 (define AFE%
827   (class object%
828     (init tuplaDef)
829     (field [Q  (list-ref tuplaDef 0)] ; Los estados.
830            [S  (list-ref tuplaDef 1)] ; El alfabeto.
831            [T  (list-ref tuplaDef 2)] ; Las transiciones.
832            [q0 (list-ref tuplaDef 3)] ; El estado inicial.
833            [A  (list-ref tuplaDef 4)]) ; Los estados aceptores.
834     (super-new)
835     ;----------
836     (define/public (edos) Q)
837     (define/public (alfa) (filter-not (λ (s) (equal? s nulo)) S))
838     (define/public (tran) T)
839     (define/public (eini) q0)
840     (define/public (acep) A)
841     (define/public (eCerr q [res (tupla q)])
842       (let* ((x (tupla q nulo))
843              (Tx (Im x T))
844              (nc (difConjuntos Tx res)))
845         (if (vacio? nc)
846             res
847             (apply union* (map (λ (p) (eCerr p (agregar p res)))
848                                Tx)))))
849     (define/public (Tr q s)
850       (let* ((P (eCerr q))
851              (R (apply union* (map (λ (p) (Im (tupla p s) T)) P))))
852         (apply union* (map (λ (r) (eCerr r)) R))))
853     (define/public (Tr* q w)
854       (if (pVacia? w)
855           (eCerr q)
856           (apply union* (map (λ (es) (Tr* es (cdr w)))
857                              (Tr q (car w))))))
858     (define/public (Tr+ P s)
859       (apply union* (map (λ (p) (Tr p s)) P)))
860     (define/public (acepta? w) (
861              existeUn (λ (ef) (en? ef A)) (Tr* q0 w)))
862     (define/public (enLeng? w) (acepta? w))
863     (define/public (nLeng [n 10])
864       (if (< n 10)
865          (filter (λ (w) (acepta? w)) (nKleene* S n))
866          (append (filter (λ (w) (acepta? w)) (nKleene* S 10))
867                  (list '...)))) ))
868
```

```
; Transforma un AFE% en un AFN% equivalente.
; E : AFE%
(define afe->afn
  (λ (E)
    (let* ((NQ (send E edos))
           (NS (send E alfa))
           (Nq0 (send E eini))
           (NA (if (vacio? (intersec (send E eCerr Nq0)
                                     (send E acep)))
                   (send E acep)
                   (union (send E acep) (conj Nq0))))
           (TT (append* (map (λ (q)
                               (map (λ (s) (tupla q
                                                  s
                                                  (send E Tr q s)))
                                    NS))
                             NQ)))
           (NT (append* (map (λ (t)
                               (map (λ (qd) (tupla (list-ref t 0)
                                                   (list-ref t 1)
                                                   qd))
                                    (last t)))
                             TT))))
      (new AFN% [tuplaDef (list NQ NS NT Nq0 NA)]))))
```

Código fuente para el capítulo 9

```
894 ; Renombra los símbolos del alfabeto de un autómata.
895 ; (S-morf AF [S-alt]) ↦ (or AFD% AFN% AFE%)
896 ; AF : (or AFD% AFN% AFE%)
897 ; S-alt : lista? = '()
898 (define S-morf
899   (λ (AF [S-alt '()])
900     (let* ((NQ (send AF edos))
901            (S  (send AF alfa))
902            (SI (build-list (card S) values))
903            (Smorf (if (vacio? S-alt)
904                       (map (λ (s t) (tupla s t)) S SI)
905                       (map (λ (s t) (tupla s t)) S S-alt)))
906            (NS (map (λ (s) (last (assoc s Smorf))) S))
907            (NT (map (λ (t) (tupla
908                                (list-ref t 0)
909                                (last (assoc (list-ref t 1) Smorf))
910                                (list-ref t 2)))
911                     (send AF tran)))
912            (NqO (send AF eini))
913            (NA (send AF acep)))
914       (new
915        (cond ((is-a? AF AFD%) AFD%)
916              ((is-a? AF AFN%) AFN%)
917              ((is-a? AF AFE%) AFE%))
918        [tuplaDef (list NQ NS NT NqO NA)])))))
919
920 ; Renombra los símbolos de los estados de un autómata.
921 ; (Q-morf AF [lit]) ↦ (or AFD% AFN% AFE%)
922 ; AF : (or AFD% AFN% AFE%)
923 ; lit : string? = "r"
924 (define Q-morf
925   (λ (AF [lit "q"]) ; la literal deseada
926     (let* ((Q  (send AF edos)) ; los estados
927            (NQO (map (λ (i) (format "~a~s" lit i))
928                      (build-list (card Q) (λ (i) (+ i 1)))))
929            (opB (if (paraTodo (λ (q) (string->number q)) NQO)
930                     string->number  string->symbol))
931            (NQ  (map (λ (p) (opB p)) NQO))
932            (Qmorf  (map (λ (q p) (list q p)) Q NQ))
933            (NS  (send AF alfa))
934            (NT  (map (λ (t)
935                        (list
936                           (last (assoc (list-ref t 0) Qmorf))
937                           (list-ref t 1)
938                           (last (assoc (list-ref t 2) Qmorf))))
939                      (send AF tran)))
940            (NqO (last (assoc (send AF eini) Qmorf)))
941            (NA  (map (λ (q) (last (assoc q Qmorf)))
942                      (send AF acep))))
943       (new
944        (cond ((is-a? AF AFD%) AFD%)
945              ((is-a? AF AFN%) AFN%)
946              ((is-a? AF AFE%) AFE%))
947        [tuplaDef (list NQ NS NT NqO NA)])))))
948
```

```
; Calcula el conjunto de estados alcanzables desde el estado inicial.
; (qOAlc AF [q] [res]) ↦ (listade Q?)
; AF : (or AFD% AFN% AFE%)
; q : Q? = (send AF eini)
; res : lista? = (tupla q)
(define qOAlc
  (λ (AF [q (send AF eini)] [res (tupla q)])
    (let* ((T (send AF tran))
           (S (union (send AF alfa) '(e)))
           (Tx (apply union* (map (λ (s) (Im (tupla q s) T)) S)))
           (nc (difConjuntos Tx res))) ; para evitar ciclos
      (if (vacio? nc)
          res
          (apply union*
             (map (λ (p) (qOAlc AF p (agregar p res))) nc))))))

; Remueve los estados inalcanzables desde el estado inicial.
; (AF-reduc1 AF)
; AF : (or AFD% AFN% AFE%)
(define Q-alc
  (λ (AF)
    (let ((Qalc (qOAlc AF)))
      (new
       (cond ((is-a? AF AFD%) AFD%)
             ((is-a? AF AFN%) AFN%)
             ((is-a? AF AFE%) AFE%))
       [tuplaDef (list Qalc ; los estados alcanzables
                       (send AF alfa) ; el alfabeto
                       (filter (λ (t) (en? (car t) Qalc))
                               (send AF tran))
                       (send AF eini)
                       (filter (λ (q) (en? q Qalc))
                               (send AF acep)))])))))

; Verifica que dos estados sean distinguibles.
; (dis? D q1 q2 QDi) ↦ booleano?
; AD : AFD; q1 : D.Q? -- Un estado.
; q2 : D.Q? -- Un estado.
; QDi : (listade Q?) -- Estados distinguibles de referencia.
(define dis?
  (λ (D q1 q2 QDi)
    (existeUn (λ (pq)
                (existeUn (λ (s) (c=? (tupla (send D Tr q1 s)
                                             (send D Tr q2 s))
                                      pq))
                          (send D alfa)))
              QDi)))

; Calcula el conjunto de estados distinguibles.
; (qDist AF QDi QIn) ↦ (listade conjunto?)
; D : AFD%
; QDi : (listade lista?) -- Pares de estados distinguibles.
; QIn : (listade lista?) --Pares de estados indistinguibles.
(define qDist
  (λ (D QDi QIn)
    (let ((nvosDis
           (filter (λ (ee) (dis? D (car ee) (cadr ee) QDi))
                   QIn)))
    (cond ((vacio? QIn) QDi)
          ((vacio? nvosDis) QDi)
          (else (qDist D
                       (union QDi nvosDis)
                       (difConjuntos QIn nvosDis)))))))
```

```
1012
1013 ; Genera una relación irreflexiva, asimetrica, antisimetrica y transitiva.
1014 ; (spCart A i j [res]) ↦ (listade lista?)
1015 ; A : lista?
1016 ; i : numero? = 0
1017 ; j : numero? = 1
1018 ; res : (listade lista?) = '()
1019 ;>(spCart '(1 2 3 4)) ↦'((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))
1020 (define spCart
1021   (λ (A [i 1] [j 0] [res '()])
1022     (let ((n (card A)))
1023       (cond ((= i n) (reverse res))
1024             ((= j i) (spCart A (+ i 1) 0 res))
1025             (else (spCart A i (+ j 1) (cons (list (list-ref A i)
1026                                                   (list-ref A j))
1027                                             res)))))))
1028
1029 ; Reduce clases de equivalencia.
1030 ; (reduceClases LK [res]) ↦ (listade lista?)
1031 ; LK : (listade lista?) -- La lista de las clases.
1032 ; res : (listade lista?) = '() -- La lista resultante de las clases.
1033 (define reduceClases
1034   (λ (LK [res '()])
1035     (cond ((vacio? LK) res)
1036           ((paraTodo (λ (K) (vacio? (intersec (car LK) K))) res)
1037                      (reduceClases (cdr LK)
1038                                    (agregar (car LK) res)))
1039           (else (let* ((V1 (car (filter-not
1040                              (λ (K) (vacio? (intersec (car LK) K)))
1041                              res)))
1042                        (V2 (difConjuntos res (list V1))) ;
1043                        (V3 (agregar (union (car LK) V1) V2)))
1044                   (reduceClases (cdr LK) V3))))))
1045
1046 ; Calcula las clases de equivalencia de los estados de un autómata.
1047 ; (qEqv AF) ↦ (listade conjunto?)
1048 ; AF : AFD%
1049 (define Q-Keq
1050   (λ (AF)
1051     (let* ((LPQ (spCart (send AF edos)))
1052            (A   (send AF acep))
1053            (QDi (filter (λ (p)
1054                           (ox (en? (send AF Tr* (car p)  '()) A)
1055                               (en? (send AF Tr* (cadr p) '()) A)))
1056                         LPQ))
1057            (QIn (difConjuntos LPQ QDi))
1058            (Qeq (difConjuntos LPQ (qDist AF QDi QIn)))
1059            (Kunit (map (λ (q) (list q)) (send AF edos))) )
1060       (reduceClases (union Qeq Kunit)))))
1061
1062 ; Calcula la clase de equivalencia de un estado.
1063 ; (Kde q LK) ↦ lista?
1064 ; q : cualquier
1065 ; LK : (listade lista?) -- Lista de clases de equivalencia.
1066 (define Kde
1067   (λ (q LK)
1068     (car (filter (λ (K) (en? q K)) LK))))
1069
```

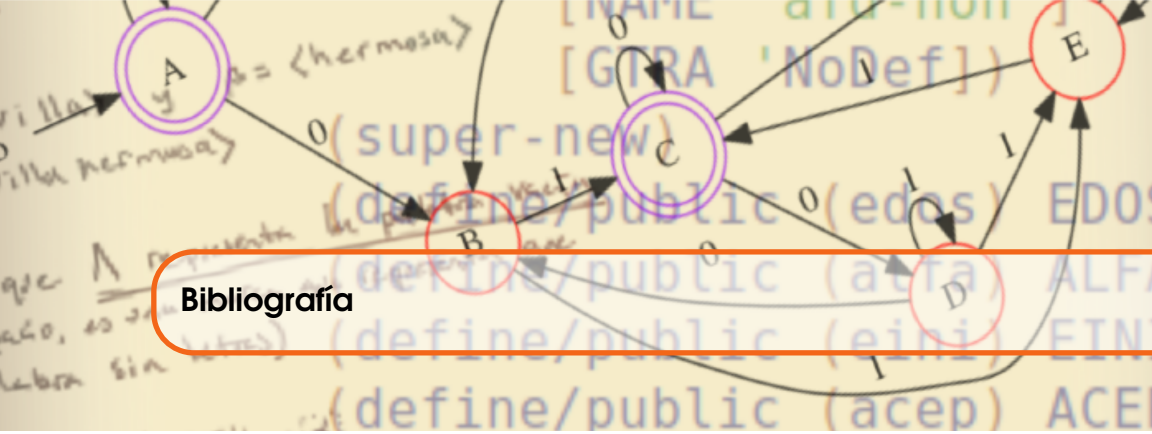
```
; Genera un autómata con clases de estados equivalentes.
; (Q-dis D) ↦ AFD%
; D : AFD%
(define Q-dis
  (λ (D)
    (let* ((NQ (Q-Keq D)) ; Los nuevos estados.
           (NS (send D alfa)) ; El mismo alfabeto.
           (NT (append*
                (map (λ (nq)
                       (map (λ (s)
                              (tupla nq s
                                     (Kde (send D Tr (car nq) s) NQ)))
                            NS))
                     NQ)))
           (NqO (car (filter (λ (k) (en? (send D eini) k)) NQ)))
           (NA (filter-not (λ (k) (vacio?
                                   (intersec k (send D acep))))
                           NQ)))
      (new AFD% [tuplaDef (list NQ NS NT NqO NA)]))))

; Transforma un AFE% en un AFN% equivalente.
; (afe->afd AF #:pref [pref "d"]) ↦ AFD%
; AF : (or AFE% AFN%)
; pref : string = "d- Un prefijo para el nombre de estados.
(define afe->afd
  (λ (AF #:pref [pref "d"])
    (let ((N (if (is-a? AF AFE%) (afe->afn AF) AF)))
     (Q-morf (Q-dis (Q-alc (Q-morf (afn->afd N)))) pref))))
```

Código fuente para el capítulo 10

```
; Crea el autómata vacío. El lenguaje vacío.
(define rx-∅
  (new AFD% [tuplaDef (list '(q1) '() '() 'q1 '())]))

; Crea el autómata palabra vacía.
;El lenguaje que acepta únicamente la palabra vacía.
(define rx-Λ
  (new AFD% [tuplaDef (list '(q1) '() '() 'q1 '(q1))]))

; Verifica que el sea un autómata nulo.
;(AFnulo? A) ↦ booleno?
; A : AFD%
(define AFnulo?
  (λ (A)
    (cond ((vacio? (send A acep)) #t)
          ((vacio? (send A nLeng (card (send A edos)))) #t)
          (else #f))))

; Verifica un autómata vacío. Acepta solo la palabra vacía.
; (AFvacio? A) ↦ booleano?
; A : (or AFE% AFN% AFD%)
(define AFvacio?
  (λ (A)
    (cond ((c=? (send A nLeng (card (send A edos))) '(())) #t)
          (else #f))))

; Calcula la unión de dos autómatas.
; (AF-union B C [pref]) ↦ AFD%
; B : (or AFE% AFN% AFD%)
; C : (or AFE% AFN% AFD%)
; pref ="q": string? -- Un prefijo para renombrar estados.
(define AF-union
  (λ (B C [pref "q"])
    (let* ((B1 (Q-morf B "B"))
           (C1 (Q-morf C "C"))
           (tib (tupla 'ii nulo (send B1 eini)))
           (tic (tupla 'ii nulo (send C1 eini)))
           (NQ (agregar 'ii (union (send B1 edos)
                                   (send C1 edos))))
           (NS (union (send B1 alfa) (send C1 alfa)))
           (NT (union* (send B1 tran) (send C1 tran)
                       (conj tib tic)))
           (NqO 'ii)
           (NA (union (send B1 acep) (send C1 acep)))
           (NE (new AFE% [tuplaDef (list NQ NS NT NqO NA)])))
      (if (equal? pref "")
          NE (afe->afd NE #:pref pref)))))

; Calcula la unión extendida de una lista no determinada de autómatas.
; (AF+ B1 ...+) ↦ AFD%
; B1 ...+ : (listade (or AFE% AFN% AFD%))
(define AF+
  (λ (#:pref [pref ""] . LA)
    (letrec ([AF+aux
              (λ (LAF res)
                (if (vacio? LAF)
                    res
                    (AF+aux (cdr LAF)
                            (AF-union res (car LAF) pref))))])
      (if (vacio? LA) rx-∅ (AF+aux (cdr LA) (car LA))))))

```

```
; Calcula la concatenación de dos autómatas.
; (AF-concat B C [pref]) ↦ AFD%
; B : (or AFE% AFN% AFD%)
; C : (or AFE% AFN% AFD%)
; pref : string? = "q- Un prefijo para renombrar estados.
(define AF-concat
  (λ (B C [pref "q"])
    (let* ((B1 (Q-morf B "B"))
           (C1 (Q-morf C "C"))
           (NQ (union (send B1 edos) (send C1 edos)))
           (NS (union (send B1 alfa) (send C1 alfa)))
           (tfa (map (λ (ef) (tupla ef nulo (send C1 eini)))
                (send B1 acep)))
           (NT (union* (send B1 tran) (send C1 tran) tfa))
           (NqO (send B1 eini))
           (NA (send C1 acep))
           (NE (new AFE% [tuplaDef (list NQ NS NT NqO NA)])))
      (if (equal? pref "")
          NE
          (afe->afd NE #:pref pref)))))

; Calcula la concatenación extendida de una lista no determinada de autómatas.
; (AF* B1 ...+) ↦ (or AFE% AFD%)
; B1 ...+ : (listade (or AFE% AFN% AFD%))
(define AF*
  (λ (#:pref [pref ""] . LA)
    (letrec ([AF*aux
              (λ (LAF res)
                (if (vacio? LAF)
                    res
                    (AF*aux (cdr LAF) (AF-concat res (car LAF) pref))))])
      (if (vacio? LA) rx-Ø (AF*aux (cdr LA) (car LA))))))

; Calcula la n-ésima potencia de un autómata.
; (AF-pot A n [pref] [res]) ↦ AFD%
; A : (or AFE% AFN% AFD%)
; n : numero-entero-no-negativo?
; pref : string? = "q- Un prefijo para renombrar estados.
; res : AFD% -- El autómata resultante.
(define AF-pot
  (λ (A n [pref ""] [res A])
    (cond ((= n 0) rx-Λ)
          ((= n 1) res)
          (else (AF-pot A (- n 1) pref (AF-concat res A pref))))))

; Calcula la cerradura de Kleene de un autómata.
; (AF-Kleene* AF [pref]) ↦ AFD%
; AF : (or AFE% AFN% AFD%)
; pref = "q": string? -- Un prefijo para renombrar estados.
(define AF-Kleene*
  (λ (AF [pref "K"])
    (let* ((NQ (send AF edos))
           (NS (send AF alfa))
           (Tf (map (λ (ea) (tupla ea nulo (send AF eini)))
                    (send AF acep)))
           (NT (union (send AF tran) Tf))
           (NqO (send AF eini))
           (NA (agregar (send AF eini) (send AF acep)))
           (NE (new AFE% [tuplaDef (list NQ NS NT NqO NA)])))
      (if (equal? pref "")
          NE
          (afe->afd NE #:pref pref)))))
```

```
; Calcula la cerradura finita de Kleene de un autómata.
; (AF-nKleene A n [pref] [res]) ↦ AFD%
; A : (or AFE% AFN% AFD%)
; n : numero-entero-no-negativo?
; pref : string = "q- un prefijo para renombrar estados.
; res : AFD% -- el autómata resultante
(define AF-nKleene*
  (λ (A n [pref ""] [res rx-Λ])
    (if (= n 0)
        res
        (AF-nKleene* A (- n 1) pref (AF-union res (AF-pot A n pref) pref)))))

; Calcula la cerradura positiva de Kleene de un autómata.
; (AF-Kleene+ A [pref]) ↦ AFD%
; A : (or AFE% AFN% AFD%)
; pref : string = "q- Un prefijo para renombrar estados.
(define AF-Kleene+
  (λ (A [pref "K"])
    (AF-concat A (AF-Kleene* A pref) pref)))
```



Bibliografía

[Abb03] Janet Abbate. Women and Gender in the History of Computing. *IEEE Annals of the History of Computing*, 25(4):4–8, Oct 2003.

[Amb04] S.W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 2004.

[ASS96] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. Electrical engineering and computer science series. Cambridge, MA, 1996.

[Ass16] Association for Computing Machinery. A. M. Turing Award. Web page at `http://amturing.acm.org/`, 2016 (last visited on jul-2016).

[BM76] J. A. Bondy and U.S.R. Murty. *Graph Theory with Applications*. North Holland, 1976.

[BW57a] Arthur W. Burks and Hao Wang. The Logic of Automata – Part I. *J. ACM*, 4(2):193–218, apr 1957.

[BW57b] Arthur W. Burks and Hao Wang. The Logic of Automata – Part II. *J. ACM*, 4(3):279–297, jul 1957.

[Cer91] Paul E. Ceruzzi. When Computers Were Human. *Annals of the History of Computing*, 13(3):237–244, July 1991.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ, 1941.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.

[FFF06] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with Classes, Mixins, and Traits (invited tutorial). In *Asian Symposium on Programming Languages and Systems*, 2006. `http://www.cs.utah.edu/plt/publications/aplas06-fff.pdf`.

[FP16] Mattew Flatt and PLT. The Racket Reference v.6.6. Creating Objects. On Line Reference at `https://docs.racket-lang.org/reference/objcreation.html?q=super-new`, 2016.

[Gin68] Abraham Ginzburg. *Algebraic Theory of Automata*. ACM Monograph Series. Academic Press, 1968.

[Gri13] David Alan Grier. *When Computers Were Humans*. Princeton University Press, 2013. 424 pp.

[Har78] Michael A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.

[Hop71] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In A. Kohavi, Z. anf Paz, editor, *Theory of machines and computers (Proc. Internat. Sympos., Technion Haifa)*, pages 189 – 196. Academic Press N.Y., 1971.

[HU79] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley Publishing Company, Reading, Massachusets, USA, 1979.

[Jak15] Sebastian Jakobi. *Modern Aspects of Classical Automata Theory: Finite Automata, Biautomata, and Lossy Compression*. Logos Verlag Berlin, 2015.

[KKR05] Kenneth E. Kendall, Julie E. Kendall, and Antonio N. (Tr) Ramos. *Análisis y Diseño de Sistemas*. Pearson Educacón. Pearson Educación, 2005.

[Kle81] Stephen C. Kleene. The Theory of Recursive Functions, Approaching its Centennial. *Bulletin (New Series) of the American Mathematical Society*, 5(1):43 – 61, July 1981.

[Koe01] Teun Koetsier. On the Prehistory of Programmable Machines: Musical Automata, Looms, Calculators. *Mechanism and Machine Theory*, 36(5):589 – 603, 2001.

[Lan65a] Peter John Landin. Correspondence Between ALGOL 60 and Church's Lambda-notation: Part I. *Commun. ACM*, 8(2):89–101, February 1965.

[Lan65b] Peter John Landin. Correspondence Between ALGOL 60 and Church's Lambda-notation: Part II. *Commun. ACM*, 8(3):158–167, March 1965.

[Law03] Mark V. Lawson. *Finite Automata*. Chapman & Hall CRC press, 2003.

[LY08] Li Li and Yun Yang. E-Business Process Modelling with Finite State Machine Based Service Agents. In Weiming Shen, Jianming Yong, Yun Yang, Jean-Paul A. Barthès, and Junzhou Luo, editors, *Computer Supported Cooperative Work in Design IV: 11th International Conference, CSCWD 2007, Melbourne, Australia, April 26-28, 2007. Revised Selected Papers*, volume 5236 of *Lecture Notes in Computer Science*, chapter Computer Supported Cooperative Work in Design IV, pages 261–272. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[McC78] John McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, aug 1978.

[Mea55] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal, The*, 34(5):1045–1079, Sept 1955.

[MH81] A.N. Michel and C.J. Herget. *Applied Algebra and Functional Analysis*. Dover Books on Mathematics. Dover Publications, 1981.

[MM67] Marvin Minsky. *Computation, Finite and Infinite Machines*. Automatic Computation. Prentice-Hall Inc. NJ, 1967.

[Moh96] Mehryar Mohri. On Some Applications of Finite-State Automata Theory to Natural Language Processing. *Journal of Natural Language Engineering*, 2, 1996.

[Moo56] Edward F. Moore. Gedanken-Experiments on Sequential Machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[MP43] Warren S. McCulloch and Walter Harry Pitts. A Logical Calculus of the Idea Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[NS03] Lev Naumov and Anatoly Shalyti. Automata Theory for Multi-Agent Systems Implementation. In *Proceedings of International Conference Integration of Knowledge Intensive Multi-Agent Systems: Modeling, Exploration and Engineering. KIMAS-03*, volume Boston: IEEE Boston Section. 2003, pages 65 – 70, 2003.

[Rea16] Real Academia Española. *Diccionario de la lengua Española. Versión normal (Spanish Edition)*. Espasa, 23 edition, 2016.

[RM04] K. H. Rosen and J. M. P. Morales. *Matemática Discreta y sus Aplicaciones*. McGraw-Hill, 5 edition, 2004.

[Rog87] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA, 1987.

[RS59] Michael O. Rabin and Dana Stewart Scott. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.*, 3(2):114–125, apr 1959.

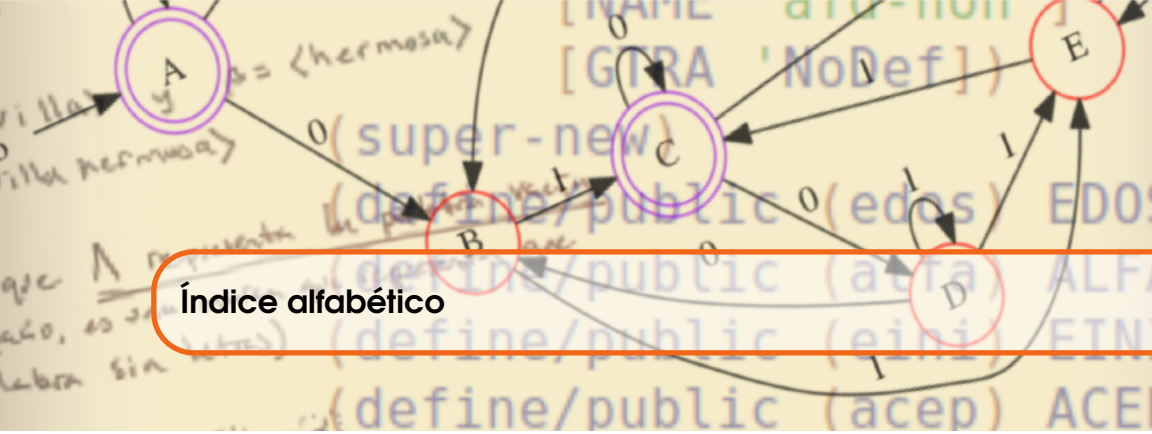
[Sha58] Claude E. Shannon. Von Neumann's Contributions to Automata Theory. *Bull. Amer. Math. Soc.*, 3 part 2(64):123 – 129, 1958.

[Smu14] R.M. Smullyan. *A Beginner's Guide to Mathematical Logic*. Dover Books on Mathematics. Dover Publications, 2014.

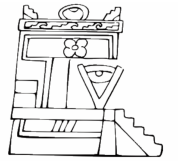
[YKSK05] Boris Yartsev, George Korneev, Anatoly Shalyto, and Vladimir Kotov. Automata-Based Programming of the Reactive Multi-Agent Control Systems. In *2005 International Conference on "Integration of Knowledge Intensive Multiagent Systems. KIMAS '05: Modeling, Exploration, and Engineering"*, pages 449 – 453, USA, MA, 2005. IEEE 2005.

[YYWL11] Jingdong Yang, Jinghui Yang, Weiguang Wang, and Gary (Ed) Lee. *Advances in Automation and Robotics, Vol.2: Selected papers from the 2011 International Conference on Automation and Robotics (ICAR 2011)*, chapter An Efficient Path Planning Method Based on State Automata Model for Mobile Robots, pages 181 – 188. Lecture Notes in Electrical Engineering. Springer Berlin Heidelberg, 2011.

[Zad13] L.A. Zadeh. Stochastic Finite-State Systems in Control Theory. *Information Sciences*, 251:1 – 9, 2013.



Índice alfabético



Difusión y Divulgación
Científica y Tecnológica

José Manuel Piña Gutiérrez
Rector

Raúl Guzmán León
Secretario de Investigación, Posgrado
y Vinculación

Andrés González García
Director de Difusión y Divulgación Científica
y Tecnológica

Calíope Bastar Dorantes
Jefa del Departamento Editorial
de Publicaciones No Periódicas

Esta obra se terminó de editar el 17 de octubre de 2019;
en la División Académica de Ciencias Básicas, ubicada
en la Carretera Cunduacán-Jalpa KM. 1 Col. La
Esmeralda CP.86690. El cuidado estuvo a cargo del autor
y del Departamento Editorial de Publicaciones No
Periódicas de la Dirección de Difusión y Divulgación
Científica y Tecnológica de la UJAT.

9 786076 065075