Lenguajes de Programación, 2019-2
Nota de laboratorio 3: Definición de tipos de datos

Manuel Soto Romero

22 de febrero de 2019

Facultad de Ciencias, UNAM

Adicional a los tipos de datos primitivos de Racket, es posible que el programador defina sus propios tipos de datos. La variante `plai` provee algunas primitivas que permiten al programador definir datos y manipularlos a través de funciones generadas al crear el dato o mediante la técnica de reconocimiento de patrones que se estudió en la Nota de laboratorio 2. En esta nota se revisan estas primitivas y se presentan algunos ejemplos.

1. Definición de tipos

Para definir tipos de datos, plai provee la primitiva `define-type`, su sintaxis es la siguiente:

```
(define-type <NombreDelTipo>
   [<nombreConstructor> (<param1> <tipo1>?)*]+)
```

La definición de un tipo de dato se compone de:

- Un nombre para el tipo de dato. El nombrado de estos tipos usa notación Pascal, es decir, cada palabra del identificador inicia con una letra mayúscula.
- Un listado de constructores que a su vez se componen de:
  - Un nombre para el constructor. El identificador separa cada palabra por un símbolo de guión (-) y se escribe en minúsculas.
  - Un listado de parámetros que puede ser vacío. Cada parámetro tiene un identificador y un tipo de dato. El tipo de dato se especifica mediante predicados, por ejemplo: number?, boolean?, etcétera.

`(parametro tipo?)`

Ejemplo 1. Definir un tipo de dato `Arbol` que representa árboles binarios.

Solución. Tipo de dato `Arbol`:

```
1 ;; Función que permite definir estructuras genéricas.
2 (define (any? a) #t)
3
4 (define-type Arbol
5    [hoja (elem any?)]
6    [nodo (elem any?) (izq Arbol?) (der Arbol?)])
```

Listado de código 1: Definición del tipo ArbolBinario

Se aprecia en el Listado de código 1 que el tipo `Arbol` es recursivo, pues los parámetros de los constructores `izq` y `der` deben de ser del mismo tipo. Una vez definido el tipo de dato, pueden usarse los constructores `hoja` y `nodo` para construir árboles.

```
> (hoja 1)
(hoja 1)
> (nodo 1 (nodo 2 (hoja) (hoja)) (nodo (hoja) (hoja))))
(nodo 1 (nodo 2 (hoja) (hoja)) (nodo (hoja) (hoja))))
```

Observación 1. Para poder usar los constructores de un tipo de dato, deben aplicarse como cualquier otra función, esto es, delimitarse por paréntesis.

Observación 2. El predicado `any?` permite definir estructuras genéricas, heterogéneas al regresar `#t` con cualquier entrada.

□

2. Funciones predefinidas

Una vez creado un nuevo tipo de dato, `plai` provee funciones predefinidas para el tipo de dato que pueden usarse para definir funciones sobre el mismo. A continuación se listan estas funciones:

- Un predicado para el tipo de dato definido.

  Arbol?: a -> boolean

- Un predicado por cada uno de los constructores del tipo definido.

  hoja?: a -> boolean
  nodo?: a -> boolean

- Una función de acceso para cada uno de los parámetros de cada constructor del tipo definido.

  hoja-elem: Arbol -> a
  nodo-elem: Arbol -> a
  nodo-izq: Arbol -> Arbol
  nodo-der: Arbol -> Arbol

- Una función modificadora para cambiar el valor de cada uno de los parámetros de cada constructor del tipo definido.

  set-hoja-elem!:Arbol a-> void
  set-nodo-elem!: Arbol a -> void
  set-nodo-izq!: Arbol Arbol -> void
  set-nodo-der!: Arbol Arbol -> void

Observación 3. Por convención, las funciones modificadoras inician con la palabra `set` e incluyen un signo de admiración al final del identificador (`!`).

Observación 4. A lo largo de estas notas se evita el uso de funciones de acceso y modificadoras, pues traen consigo efectos secundarios que violan el principio de transparencia referencial del estilo de programación funcional. Se usan las mismas para casos muy particulares.

Ejemplo 2. Definir la función `numero-hojas` tal que `(numero-hojas a)` es el número de hojas del árbol `a`. Por ejemplo,

```
(numero-hojas (hoja 1))                    = 1
(numero-hojas (nodo 1 (hoja 2) (hoja 3))) = 2
```

Solución. Función `numero-hojas`:

```
1 ;; Función que obtiene el número de hojas de un árbol binario.
2 ;; numero-hojas: Arbol -> number
3 (define (numero-hojas a)
4    (if (hoja? a)
5        1
6        (+ 1 (numero-hojas (nodo-izq a)) (numero-hojas (nodo-der a)))))
```

Listado de código 2: Número de hojas de un árbol

□

Ejemplo 3. Definir la función `contiene?` tal que `(contiene? e a)` indica si el elemento `e` se encuentra en el árbol `a`.

```
(contiene? 1 (hoja 1))                    = #t
(contiene? 1 (hoja 2))                    = #f
(contiene? 3 (nodo 2 (hoja 1) (hoja 3)) = #t
```

Solución. Predicado `contiene?`:

```
1 ;; Predicado que indica si un elemento está contenido en un árbol.
2 ;; contiene?: a Arbol -> boolean
3 (define (contiene? e a)
4    (cond
5       [(hoja? a) (equal? e (hoja-elem a))]
6       [(nodo? a)
7          (or (equal? e (nodo-elem a))
8              (contiene? e (nodo-izq a))
9              (contiene? e (nodo-der a)))]))
```

Listado de código 3: Indica si un elemento está contenido en un árbol

□

3. Reconocimiento de patrones

Al igual que con otros tipos de datos, puede usarse la técnica de reconocimiento de patrones sobre los tipos definidos a través de la primitiva `define-type` ya sea a través de la primitiva `type-case` como de la, ya estudiada, primitiva `match`. La sintaxis de ambas primitivas se presenta a continuación:

`type-case`:

```
(type-case <tipo> <identificador>
   [<nombre-constructor> (<parámetro>*) <expresión-regreso>]+
   [else <expresión-regreso>]?)
```

`match`:

```
(match <identificador>
   [(<nombre-constructor> <parámetro>*) <expresión-regreso>]+
   [else <expresión-regreso>]?)
```

Observación 5. Algunas observaciones sobre estas primitivas:

1. El caso `else` de `type-case` es opcional, a menos que no se realice el reconocimiento de todos los patrones del tipo de dato, en cuyo caso es obligatorio. Esta primitiva debe cubrir todos los casos posibles o lanzará un error.
2. El caso `else` de `match` es completamente opcional y no es necesario reconocer todos los patrones del tipo de dato.

Ejemplo 4. Modificar la función `numero-hojas` mediante las primitivas `type-case` y `match`.

Solución. Funciones `numero-hojas1` y `numero-hojas2`:

```
1  ;; Función que obtiene el número de hojas de un árbol binario.
2  ;; numero-hojas1: Arbol -> number
3  (define (numero-hojas1 a)
4     (type-case Arbol a
5        [hoja (x) 1]
6        [nodo (x i d) (+ 1 (numero-hojas1 i) (numero-hojas1 d))]))
7
8  ;; Función que obtiene el número de hojas de un árbol binario.
9  ;; numero-hojas2: Arbol -> number
10 (define (numero-hojas2 a)
11    (match a
12       [(hoja _) 1]
13       [(nodo _ i d) (+ 1 (numero-hojas2 i) (numero-hojas2 d))]))
```

Listado de código 4: Número de hojas de un árbol

□

Ejemplo 5. Modificar el predicado `contiene?` mediante las primitivas `type-case` y `match`.

Solución. Predicados `contiene1?` y `contiene2?`:

```
;; Predicado que indica si un elemento está contenido en un árbol.
;; contiene1?: a Arbol -> boolean
(define (contiene1? e a)
   (type-case Arbol a
      [hoja (x) (equal? e x)]
      [nodo (x i d)
         (or (equal? e x) (contiene1? e i) (contiene1? e d))]))

;; Predicado que indica si un elemento está contenido en un árbol.
;; contiene2?: a Arbol -> boolean
(define (contiene2? e a)
   (match a
      [(hoja x) (equal? e x)]
      [(nodo x i d)
         (or (equal? e x) (contiene1? e i) (contiene1? e d))]))
```

Listado de código 5: Indica si un elemento está contenido en el árbol

□

Ejemplo 6. Definir la función `aplana` mediante las primitivas `type-case` y `match` tal que `(aplana a)` es la lista de elementos del árbol a en *inorden*.

Solución. Funciones `aplana1` y `aplana2`:

```
;; Función que aplana un árbol binario.
;; aplana1: Arbol -> (listof any?)
(define (aplana1 a)
   (type-case Arbol a
      [hoja (x) (list x)]
      [nodo (x i d) (append (list x) (aplana1 i) (aplana1 d))]))

;; Función que aplana un árbol binario.
;; aplana2: Arbol -> (listof any?)
(define (aplana2 a)
   (match a
      [(hoja x) (list x)]
      [(nodo x i d) (append (list x) (aplana2 i) (aplana2 d))]))
```

Listado de código 6: Lista de elementos de un árbol

□

Ejemplo 7. Definir la función `map-arbol` mediante las primitivas `type-case` y `match` tal que `(map-arbol f a)` es el árbol resultante de aplicar la función `f` a cada elemento del árbol `a`.

Solución. Funciones `map-arbol1` y `map-arbol2`:

```
1  ;; Aplica una función a cada elemento de un árbol.
2  ;; map-arbol1: procedure Arbol -> Arbol
3  (define (map-arbol1 f a)
4     (type-case Arbol a
5        [hoja (x) (hoja (f x))]
6        [nodo (x i d) (nodo (f x) (map-arbol1 f i) (map-arbol1 f d))]))
7
8  ;; Aplica una función a cada elemento de un árbol.
9  ;; map-arbol2: procedure Arbol -> Arbol
10 (define (map-arbol2 f a)
11    (match a
12       [(hoja x) (hoja (f x))]
13       [(nodo x i d) (nodo (f x) (map-arbol1 f i) (map-arbol1 f d))]))
```

Listado de código 7: Aplica una función a cada elemento del árbol

□