Lenguajes de Programación

Tipos de datos abstractos

Introducción

Manuel Soto Romero

Universidad Nacional Autónoma de México
Facultad de Ciencias

23 de agosto de 2017

Tipos de datos

Tipos de datos

Un tipo de dato es una abstracción de un conjunto de valores que tiene asociadas un conjunto de operaciones incluyendo algunas restricciones o propiedades.

Tipos de datos

Un tipo de dato es una abstracción de un conjunto de valores que tiene asociadas un conjunto de operaciones incluyendo algunas restricciones o propiedades.

Ejemplo

- Tipo de dato: `boolean`.
- Conjunto de valores: {`#t`, `#f`}
- Operaciones asociadas: `and`, `or`, `not`.
- Restricción: Las operaciones asociadas son *cerradas*.

Tipos de datos

Un tipo de dato es una abstracción de un conjunto de valores que tiene asociadas un conjunto de operaciones incluyendo algunas restricciones o propiedades.

Ejemplo

- Tipo de dato: `boolean`.
- Conjunto de valores: {`#t`, `#f`}
- Operaciones asociadas: `and`, `or`, `not`.
- Restricción: Las operaciones asociadas son *cerradas*.

Racket proporciona tipos de datos *primitivos*. Pueden ser usados por el programador en la solución de problemas.

Tipos de datos abstractos

Tipos de datos abstractos

Hay ocasiones en las que el programador requiere trabajar con tipos de datos específicos que no están definidos en el núcleo del lenguaje.

Tipos de datos abstractos

Hay ocasiones en las que el programador requiere trabajar con tipos de datos específicos que no están definidos en el núcleo del lenguaje.

Un ***Tipo de Dato Abstracto*** (TDA) es una generalización que permite al programador definir sus propios tipos de datos a partir de los tipos primitivos. Se dice que son abstractos porque nada en su definición especifica cómo deben implementarse.

Tipos de datos abstractos en Racket

Tipos de datos abstractos en Racket

Para definir tipos de datos abstractos en Racket, la variante `plai` proporciona la primitiva `define-type`, cuya sintaxis es:

```
(define-type <nombre>
    [<constructor> (<parámetro> <tipo>?)*]+)
```

Tipos de datos abstractos en Racket

Para definir tipos de datos abstractos en Racket, la variante `plai` proporciona la primitiva `define-type`, cuya sintaxis es:

```
(define-type <nombre>
    [<constructor> (<parámetro> <tipo>?)*]+)
```

Ejemplo

```
(define-type natural
    [cero]
    [suc (nat natural?)])
```

Tipos de datos abstractos en Racket

Para definir tipos de datos abstractos en Racket, la variante `plai` proporciona la primitiva `define-type`, cuya sintaxis es:

```
(define-type <nombre>
    [<constructor> (<parámetro> <tipo>?)*]+)
```

Ejemplo

```
(define-type natural
    [cero]
    [suc (nat natural?)])

> (cero)
```

Tipos de datos abstractos en Racket

Para definir tipos de datos abstractos en Racket, la variante `plai` proporciona la primitiva `define-type`, cuya sintaxis es:

```
(define-type <nombre>
    [<constructor> (<parámetro> <tipo>?)*]+)
```

Ejemplo

```
(define-type natural
    [cero]
    [suc (nat natural?)])

> (cero)
(cero)
```

Tipos de datos abstractos en Racket

Para definir tipos de datos abstractos en Racket, la variante `plai` proporciona la primitiva `define-type`, cuya sintaxis es:

```
(define-type <nombre>
    [<constructor> (<parámetro> <tipo>?)*]+)
```

Ejemplo

```
(define-type natural
    [cero]
    [suc (nat natural?)])
```

```
> (cero)
(cero)
> (suc (cero))
```

Tipos de datos abstractos en Racket

Para definir tipos de datos abstractos en Racket, la variante `plai` proporciona la primitiva `define-type`, cuya sintaxis es:

```
(define-type <nombre>
    [<constructor> (<parámetro> <tipo>?)*]+)
```

Ejemplo

```
(define-type natural
    [cero]
    [suc (nat natural?)])
```

```
> (cero)
(cero)
> (suc (cero))
(suc (cero))
```

Regalos de `plai`

Regalos de `plai`

Al momento de definir un tipo mediante la primitiva `define-type`, `plai` genera cuatro funciones adicionales:

1. Un predicado `type?` que recibe un parámetro cualquiera y regresa `#t` cuando dicho parámetro se evalúa a algo de este tipo y `#f` en cualquier otro caso.
2. Para cada constructor, se genera un predicado `constructor?`, que recibe un parámetro cualquier y regresa `#t` cuando dicho parámetro se evalúa a algo de ese constructor y `#f` en otro caso.
3. Para cada parámetro de cada constructor, se crea una función para acceder al valor de dicho parámetro que tendrá el nombre `constructor-parametro`.
4. Para cada parámetro de cada constructor, se crea una función para modificar el valor de dicho parámetro que tendrá el nombre `set-constructor-parametro!`.

Ejemplos

Ejemplos

```
> (define n1 (suc (cero)))
```

Ejemplos

```
> (define n1 (suc (cero)))
> (cero? n1)
```

Ejemplos

```
> (define n1 (suc (cero)))
> (cero? n1)
#f
```

Ejemplos

```
> (define n1 (suc (cero)))
> (cero? n1)
#f
> (suc? n1)
```

Ejemplos

```
> (define n1 (suc (cero)))
> (cero? n1)
#f
> (suc? n1)
#t
```

Ejemplos

```
> (define n1 (suc (cero)))
> (cero? n1)
#f
> (suc? n1)
#t
> (suc-nat n1)
```

Ejemplos

```
> (define n1 (suc (cero)))
> (cero? n1)
#f
> (suc? n1)
#t
> (suc-nat n1)
(cero)
```

Ejemplos

```
> (define n1 (suc (cero)))
> (cero? n1)
#f
> (suc? n1)
#t
> (suc-nat n1)
(cero)
> (set-suc-nat! n1 (suc (cero)))
```

Ejemplos

```
> (define n1 (suc (cero)))
> (cero? n1)
#f
> (suc? n1)
#t
> (suc-nat n1)
(cero)
> (set-suc-nat! n1 (suc (cero)))
> n1
```

Ejemplos

```
> (define n1 (suc (cero)))
> (cero? n1)
#f
> (suc? n1)
#t
> (suc-nat n1)
(cero)
> (set-suc-nat! n1 (suc (cero)))
> n1
(suc (suc (cero)))
```

Operaciones asociadas a un TDA

Operaciones asociadas a un TDA

Para definir operaciones asociadas a un TDA, simplemente hay que escribir una función que use el tipo de dato previamente definido.

Operaciones asociadas a un TDA

Para definir operaciones asociadas a un TDA, simplemente hay que escribir una función que use el tipo de dato previamente definido.

```
;; Función que suma dos números naturales.
;; suma: natural natural -> natural
(define (suma n1 n2)
    (if (cero? n1)
        n2
        (suc (suma (suc-nat n1) n2))))
```

Operaciones asociadas a un TDA

Para definir operaciones asociadas a un TDA, simplemente hay que escribir una función que use el tipo de dato previamente definido.

```
;; Función que suma dos números naturales.
;; suma: natural natural -> natural
(define (suma n1 n2)
    (if (cero? n1)
        n2
        (suc (suma (suc-nat n1) n2))))

;; Función que multiplica dos números naturales.
;; multiplica: natural natural -> natural
(define (multiplica n1 n2)
    (if (cero? n1)
        (cero)
        (suma n2 (multiplica (suc-nat n1) n2))))
```