Introducción a la Programación Funcional
Nota de clase 04: Estructuras Definidas por el Programador*

Karla Ramírez Pulido Manuel Soto Romero

3 de septiembre de 2020
Facultad de Ciencias UNAM

Uno de los aspectos más importantes de Racket es la definición de tipos de datos mediante *estructuras*. Las estructuras son importantes pues son la base para escribir programas con interfaces gráficas. Una *estructura* agrupa un conjunto de campos que conforman al tipo de dato (valores).

4.1. Definición de Estructuras

Para definir una estructura se usa la siguiente sintaxis:

```
(struct <nombre> (<campo>*))
```

Adicionalmente, al crear una estructura se crean automáticamente una serie de funciones que permiten trabajar con la misma.

- Una *función constructora* que permite construir estructuras del tipo definido a través de sus campos. La función constructora tiene el mismo nombre que el de la estructura.

```
(<nombre> (<campo>*))
```

```
> (struct punto (x y))
> (punto 1 2)
#<punto>
```

- Un *predicado* que indica si el argumento que recibe es del tipo de la estructura definida. El predicado se nombra agregando el símbolo `?` al nombre de la estructura.

```
(<nombre>? <argumento>)
```

```
> (define p (punto 1 2))
> (punto? 'punto)
#f
> (punto? p)
#t
```

*Adaptación de *Manual de Prácticas para la Asignatura de Programación Declarativa*.

- Por cada campo de la estructura, una *función de acceso* que permite obtener el valor de cada campo de la estructura.

```
;; Se crea una función por campo>
(<nombre>-<campo> <argumento>)
```

```
> (punto-x (punto 1 2))
1
> (punto-y (punto 1 2))
2
```

4.2. Copia de estructuras

Es posible clonar una estructura para generar nuevas instancias, con la posibilidad de cambiar el valor de sus campos. Para hacer esto se usa la función `struct-copy`.

```
(struct-copy <nombre-estructura> <instancia> [<campo> <valor>*])
```

La función recibe (1) el nombre de la estructura a copiar, (2) la instancia que se necesita copiar y (3) la actualización de los campos en caso de requerirlo.

```
> (struct punto (x y))
> (define p1 (punto 1 2))
> (define p2 (struct-copy punto p1 [x 3]))
> (list (punto-x p1) (punto-y p1))
'(1 2)
> (list (punto-x p2) (punto-y p2))
'(3 2)
```

4.3. Estructuras heredadas

Una estructura puede ser heredada de otra mediante un mecanismo similar a la herencia de los lenguajes orientados a objetos. Para heredar una estructura se usa la siguiente sintaxis:

```
(struct <nombre-hija> <nombre-madre> (<campo>*))
```

La estructura que hereda se conoce como *estructura hija* mientras que la heredada se conoce como *estructura madre*. Al heredar una estructura se heredan todos los campos de la estructura madre, sin embargo, no se generan nuevas funciones de acceso, por lo que deben usarse las de la estructura heredada.

```
> (struct punto (x y))
> (struct punto-3d punto (z))
> (define p (punto-3d 1 2 3))
> p
#<punto-3d>
> (punto? p)
#t
> (punto-3d-z p)
3
```

```
> (punto-3d-x p)
punto-3d-x: undefined;
cannot reference an identifier before its definition
in module: top-level
internal name: punto-3d-x
> (punto-x p)
1
```

4.4. Estructuras opacas y transparentes

Al imprimir una estructura, RACKET tiene dos formas de mostrar los datos*:*

1. De forma *opaca:*

```
> (define p (punto 1 2))
> p
#<punto>
```

2. De forma *transparente:*

```
> (define p (punto 1 2))
> p
(punto 1 2)
```

Para hacer que una estructura sea transparente, es decir, que al imprimirla muestre el contenido de sus atributos, se debe proporcionar la palabra reservada[1] `#:transparent` dentro de la creación de la estructura. Esto es:

```
> (struct posn (x y) #:transparent)
> (posn 1 2)
(posn 1 2)
```

[1]Las palabras reservadas en Racket se usan como indicadores especiales en los argumentos de las funciones que permiten definir algunas opciones sobre su funcionamiento y no debe confundirse con los nombres de variables reservados para uso exclusivo del lenguaje.

4.4.1. Más palabras reservadas sobre estructuras

`#:mutable` Ocasiona que los atributos de la estructura se vuelvan mutables, esto es, que sus valores puedan cambiar sin necesidad de crear una nueva estructura. Al incluir esta palabra reservada se crean *funciones modificadoras* para cada atributo.

```
> (struct punto (x y) #:mutable)
> (define p (punto 1 2))
> (set-punto-x! p 10)
> (punto-x p)
10
```

`#:auto-value` Permite establecer un valor automático para los atributos de la estructura que incluyan la palabra reservada `#:auto`.

```
> (struct punto-3d (x y [z #:auto]) #:transparent #:auto-value 0)
> (punto-3d 1 2)
(punto 1 2 0)
```

`#:guard` Permite modificar o verificar los atributos de una estructura. Esto es útil para definir de qué tipo son los atributos o bien modificar la estructura cuando alguna *función constructora es llamada.*

```
> (struct thing (name) #:transparent #:guard
      (lambda (name type-name)
            (cond
                  [(string? name) name]
                  [(symbol? name) (symbol->string name)]
                  [else (error type-name "bad name: ~e" name)])))
> (thing "apple")
(thing "apple")
> (thing 'apple)
(thing "apple")
> (thing 1/2)
thing: bad name: 1/2
```

Referencias

[1] Manuel Soto, *Manual de Prácticas para la Asignatura de Programación Declarativa*, Facultad de Ciencias, Reporte de Actividad Docente, 2019.

[2] Matthew Flatt, Robert B. Findler, et. al., *The Racket Guide*, Versión 7.5