Dr. Racket
Programación I (R-113)
Licenciatura en Ciencias de la Computación

Iker M. Canut

2020

1 Expresiones

Los lenguajes de programación tienen un vocabulario y una gramática que determinan la **sintaxis** del mismo, y cierto significado que establece su **semántica**.

- Notación prefija: (+ 2 3)
- Notación infija: $(2 + 3)$
- Notación posfija: (2 3 +)

La sintaxis de racket establece que una operación tiene notación prefija y debe ser encerrada entre paréntesis para ser una expresión válida: *(<operador> <operando 1> ... <operando n>)*

1.1 Expresiones aritméticas

Claramente se pueden usar operadores como +, **-**, / y ***. Luego, otros más interesantes son: **modulo**, **sqrt**, **sin**, **cos**, **tan**, **log**, **expt**, **random** ,**max**, **min**, **floor**, **ceiling**, **abs**.

1.2 Strings

Un string es una secuencia de caracteres encerrada entre comillas. Algunas operaciones utiles de strings son: **string-append**, **string-length**, **number->string**, **string-ith** (que dados un string y un número n, nos devuelve el caracter que ocupa la n-ésima posición $[0, n)$, o **substring**, que dado un string y dos numeros nos devuelve el intervalo cerrado abierto $[a, b)$ (consejo, restar mayor y menor, te da cuantos caracteres).

1.3 Valores Booleanos

En Racket se escriben como #**t** (o #true) y #**f** (o #false). Se pueden usar los operadores **and**, **or** y **not**. Tambien se pueden usar el $<$, $>$, $<=$ y $>=$.

2 Imagenes

Se pueden copiar y pegar imágenes a DrRacket, o crear las propias. Para obtener sus dimensiones se pueden usar **image-width**, **image-height**. Para crear imagenes, se pueden usar:

- **(circle radius mode color)**
- **(ellipse width height mode color)**
- **(add-line image x1 y1 x2 y2 color)**
- **(text string font-size color)**
- **(triangle side-length mode color)**,
  (right-triangle side-length1 side-length2 mode color),
  (isosceles-triangle side-length angle mode color),
  (triangle/sss side-length-a side-length-b side-length-c mode color),
  (triangle/sas side-length-a angle-b side-length-c mode color)
- **(square side-len mode color)**
- **(rectangle width height mode color)**
- **(rhombus side-length angle mode color)**
- **(star side-length mode color)**
  (star-polygon side-length side-count step-count mode color)

Donde *mode* puede ser "outline" o "solid", y *color* un string con un color en ingles.

Para dibujar una imagen encima de la otra, se usa la función **(overlay i1 i2 is ...)**, o sino **(underlay i1 i2 is ...)**, que funciona igual pero con los parametros invertidos.

Además, es muy util crear una escena para sacarle provecho a las funciones de 2htdp/universe. Para crear una escena se puede usar **(empty-scene width height [color])**. Luego, se pueden insertar imagenes con **(place-image image x y scene)**.

3 Funciones y Constantes

Para crear una constante se usa **(define <identificador> <expresión>)**. Para crear una función se usa **(define (<identificador> <argumento 1> ... <argumento n>) <expresión>)**.

4 Leyes de Reducción

Una definición **(define (f x) e)** agrega la siguiente ley de reducción:

> $(f\ a)$
> == definición de f (ley 1)
> $e[a/x]$

Donde e[a/x] significa "*reemplazar en e todas las ocurrencias de x por la expresión a*".

5 Condicionales Simples: if

Una proposición es una expresión que puede evaluar a verdadero o a falso. En DrRacket, para tomar decisiones usamos los ifs: **(if p a b)**. Donde **p** es un predicado, una proposición. Si **p** es verdadero se evalua **a**, en caso contrario, es decir, si **p** es falso se evaulua **b**.

6 Condicionales Multiples

```
(cond  [ Condición-1 Resultado-1 ]
       [ Condición-2 Resultado-2 ]
       ...
       [ Condición-n Resultado-n ])
```

Primero se evalúa la primer condición. Si esta reduce a #true, entonces el resultado de toda la expresión es el que se obtiene de evaluar Resultado-1. Caso contrario, DrRacket descarta la primer pareja de la expresión condicional, y procede con la segunda pareja del mismo modo que con la primera. Si todas las condiciones evalúan a #false, entonces se produce un error. De esta manera, creamos **n** leyes de reducción.

7 Predicados

Son funciones que devuelven un valor de verdad. DrRacket provee funciones para establecer si un valor pertenece a una determinada clase, como **string?**, **number?**, **boolean?**...

8 Diseñando Programas

1. **Diseño de datos**: Definimos la forma de representar la información como datos.
2. **Signatura**: Indica qué datos consume, y qué datos produce.
3. **Declaración de propósito**: Consiste en una breve descripción del comportamiento de la función. Entender qué calcula la función sin necesidad de inspeccionar el código.
4. **Definición de la función**: Escribimos el código.
5. **Evaluar el código en los ejemplos**: Verificamos que el programa funciona para algunos ejemplos.

8.1 Ejemplo

```
; Representamos temperaturas mediante números
; far->cel : Number -> Number
; Recibe una temperatura en Fahrenheit, devuelve su equivalente en Celsius
(check-expect (far->cel 32) 0)
(check-expect (far->cel 212) 100)
(check-expect (far->cel 122) 50)
(define
        (far->cel t)
        (* 5/9 (- t 32)))
```

9 Programas Interactivos

Programas por lotes: una vez lanzado el proceso no necesitan ningún tipo de interacción.
Programas interactivos: esperan la intervención del usuario (o de eventos producidos por las computadoras) para llevar a cabo sus funciones. Los programas reciben información de su entorno a través de **eventos**. Los eventos describen la ocurrencia de alguna situación contemplada, que requiere información para tomar una determinada acción. E.g, presionar una tecla, mover el mouse, otro programa envía un mensaje, el reloj de la computadora marca el paso del tiempo (ticks)... La acción que se lleva a cabo queda establecida a partir de la definición de una función llamada **manejador de eventos**. Los manejadores de eventos cambian determinadas propiedades o valores dentro del programa, es decir, cambian el **estado** del mismo.

9.1 big-bang

```
(big-bang < estado inicial >
        [to-draw < controlador de pantalla >]
        [on-key < manejador de teclado >]
        [on-mouse < manejador de mouse >]
        [on-tick < manejador de reloj >]
        [stop-when < predicado de fin de ejecución >]
        etc...
        )
```

Cuando se evalúa, el comportamiento de esta expresión es como sigue:

1. La función asociada a **to-draw** es invocada con el **estado inicial** como argumento, y su resultado se muestra por pantalla.
2. El programa queda a la espera de un evento (**on-key**, **on-mouse**, **on-tick**...)
3. Cuando un evento ocurre, el manejador asociado a dicho evento (si existe) es invocado, y devuelve el nuevo estado.
4. En caso de estar presente, se aplica el predicado (la condición) asociado a la clásula **stop-when** al nuevo estado. Si devuelve #true , el programa termina, si no,
5. la función asociada a **to-draw** es nuevamente invocada con el nuevo estado, devolviendo la nueva imagen o escena.
6. El programa queda a la espera de un nuevo evento (volvemos al paso 2).

9.1.1 Ejemplo

```
(define (interpretar s) (place-image (circle 10 "solid" s) 100 100 (empty-scene 200 200)))

(big-bang "blue"
          [to-draw interpretar]
          [on-key manejarTeclado])

(define (manejarTeclado s k) (cond [(key=? k "a") "blue"]
                                   [(key=? k "r") "red"]
                                   [(key=? k "v") "green"]
                                   [else s]))
```

Siempre que asociemos una función al evento on-key, esta tendrá **dos argumentos**: el estado actual y un string que representa la tecla que generó el evento. Y siempre devolverá un nuevo estado.

Generalizando, todos los manejadores de eventos son funciones que toman como argumento el estado actual del sistema y una descripción del evento ocurrido, y devuelven siempre el nuevo estado.

10 Estructuras

```
(define-struct posn [x y]); Crea las siguientes funciones:
; make-posn : Number Number -> posn ; CONSTRUCTOR
; posn-x : posn -> Number ; SELECTOR (ley 1)
; posn-y : posn -> Number ; SELECTOR (ley 2)
; posn? : X -> Boolean ; PREDICADO
```

11 Listas

La lista vacia está dada por la expresión **'()**. Para agregar un elemento a una lista, se usa la función **cons** $<$ **elemento** $>$ $<$ **lista** $>$. Como *cons* devuelve una lista, se puede pasar a otro *cons*. Esto es una definición autorreferenciada, porque se puede definir como una lista vacia o bien una expresión del mismo tipo (es decir, cons x l).

11.1 Operaciones sobre listas

Operador	Tipo de Operador	Función
'()	Constructor	Representa la lista vacía.
cons	Constructor	Agrega un elemento a una lista.
first	Selector	Devuelve el primer elemento de la lista.
rest	Selector	Devuelve la lista sin su primer elemento.
empty?	Predicado	Reconoce únicamente la lista vacía.
cons?	Predicado	Reconoce listas no vacías.

También estan los selectores **second**, **third**, **fourth**, **fifth**, **sixth**, **seventh**, **eighth**.

11.2 Simplificaciones

DrRacket nos permite simplificar listas de la siguiente manera:

- En lugar de escribir **'()** podemos usar la palabra **empty**.
- Para definir **(cons a0 (cons a1 (.... (cons an empty))))**, escribimos **(list a0 a1 ... an)**.

Es importante notar que **list** no es un constructor, sino un *operador* que nos provee el lenguaje para simplificar la escritura de las listas en nuestros programas.

11.3 Operaciones Definidas

- **(member? x l)** dado un valor x y una lista l, chequea si el valor x está en la lista l.
- **(length l)** dada una lista l, devuelve la cantidad de elementos que hay en l.
- **(list-ref l pos)** dada una lista l y un número entero no negativo, devuelve el elemento en la posición pos de la lista l.
- **(list-tail l pos)** dada una lista l y un número entero no negativo, devuelve la lista l despues de la posición pos. (Pensa como se saltea pos elementos).
- **(append l1 l2 .. ln)** dadas n listas, las concatena en orden.
- **(reverse l)** dada una lista l, la devuelve con el orden invertido.

11.4 Operaciones de Alto Orden

filter: Nos quedamos con los elementos de una lista que cumplen determinada condición.

```
(define (filter p l)
        (cond [(empty? l) empty]
              [else (if (p (first l))
                        (cons (first l) (filter p (rest l)))
                        (filter p (rest l)))]))
```

map: aplicamos una transformación a cada elemento de una lista. $[a0, a1, ..., an] \rightarrow [f(a0), f(a1), ..., f(an)]$.

```
(define (map f l)
        (cond  [(empty? l) empty]
               [else (cons (f (first l)) (map f (rest l)))]))
```

foldr: realizar una operación que involucra a todos los elementos de la lista. $[a0, a1, ..., an] \rightarrow (fa0(fa1(...(fan-1an))))$

```
(define (fold f c l)
          (cond  [(empty? l) c]
                 [else (f (first l) (fold f c (rest l)))]))
```

. . . . . .

andmap: mapea un predicado a una lista, y se fija que todos los valores sean #true.

ormap: mapea un predicado a una lista, y se fija que al menos un valor sea #true.

remove: dado un elemento y una lista, devuelve la lista omitiendo la primer aparación del elemento.

remove*: como *remove*, pero remueve todas las apariciones.

(sort l proc): dado una lista y un procedimiento que toma 2 elementos (e.g <, >), ordena la lista l.

(require racket/list)

(take l pos): devuelve una lista cuyos elementos son los primeros pos elementos de l. $[0, n]$

(drop l pos): devuelve una lista cuyos elementos son los elementos de l sin los primeros pos. $[0, n]$

check-duplicates: dada una lista devuelve el primer duplicado que tiene. #f si no encontró ninguno.

remove-duplicates: dada una lista devuelve la misma sin elementos duplicados.

count: dado un predicado y una lista, cuenta los que devuelven #true.

partition: dado un predicado y una lista, devuelve dos listas, primero con los elementos que evaluan el predicado como #true, y la segunda lista que son los que evaluan a #false.

combinations: dada una lista y opcionalmente el tamaño de los sets a formar, devuelve todos las posibles combinaciones de elementos (powerset).

permutations: dada una lista, devuelve todas las posibles permutaciones de sus elementos.