

Práctica 0

Programación I

1 Expresiones

Por ejemplo, en castellano, las palabras: "*grande*", "*casa*", "*es*", "*la*", son parte del vocabulario; y las reglas gramaticales permiten construir la expresión válida "*la casa es grande*", como así también, establecer que "*casa la es grande*" es una expresión inválida.

Como en castellano, los lenguajes de programación tienen un vocabulario y una gramática que determinan la **sintaxis** del mismo, y cierto significado que establece su **semántica**. Esto es, palabras básicas y reglas para combinarlas a fin de construir **expresiones válidas** en el lenguaje, con un cierto significado.

Un programa es un conjunto de expresiones válidas escritas en un lenguaje de programación, las cuales podrán ser computadas de acuerdo a la semántica definida por el lenguaje.

En este curso utilizaremos *racket* para escribir programas que podremos computar usando el intérprete *DrRacket*.

La notación prefija establece que el operador es escrito delante de sus argumentos.

" $+ 2 3$ " notación **prefija**

" $2 + 3$ " notación **infija**

" $2 3 +$ " notación **posfija**

racket tiene definidos datos primitivos (por ej., números) y operadores sobre dichos datos (por ej.: $+$, $-$, $*$, etc.). La sintaxis de *racket* establece que una operación tiene **notación prefija** y debe ser **encerrada entre paréntesis** para ser una expresión válida. Es decir, una operación tiene la siguiente forma:

$(<operador> <operando 1> \dots <operando n>)$

Por ejemplo; para escribir un programa en *racket* que calcule

$2 + 3$

debemos escribir el operador $+$ seguido de sus dos argumentos $2 3$, todo encerrado entre paréntesis, del siguiente modo.

$(+ 2 3)$

Veamos otro ejemplo en el cual queremos un programa que calcule

$2 + 3 * 4$

Si tuviéramos que llevar a cabo el cálculo a mano, debido a la precedencia de los operadores, primero computaríamos $3 * 4$ y luego le sumaríamos 2 . Con lo cual, si queremos escribir esto en *racket*, debemos usar el mismo razonamiento. Primero escribimos la operación del producto

$(* 3 4)$

y luego le sumamos 2

`(+ 2 (* 3 4))`

Como veremos, además de los números hay otros datos primitivos con sus respectivas operaciones definidos en *racket* . Dichos datos y operaciones nos permitirán, por ejemplo, crear y manipular textos e imágenes.

2 Expresiones aritméticas.

1. Considere las siguientes expresiones aritméticas:

`12 * 5 - 7 * 6`

`3 * 5 - 7 * 4 / 14 + 3 / 1`

`cos(0.8) + 1/5 + sen(0.5) * 3.5`

a. Exprese cada una de ellas con la sintaxis de *racket* .

b. Sin usar *DrRacket* evalúelas. Aquí un ejemplo para la expresión `(* (+ 2 3) 4)`:

`(* (+ 2 3) 4)`

`==`

`(* 5 4)`

`==`

`20`

c. Usando *DrRacket*, verifique los resultados obtenidos.

d. Para algunas de las expresiones, realice la evaluación *paso a paso* en *DrRacket* .

2. Usando *DrRacket* , evalúe las siguientes expresiones:

Quizás algunas de estas expresiones no denoten valores y resulten en error. Es bueno entender qué mensaje recibimos en esos casos!

a. `(/ 1 (sin (sqrt 3)))`

b. `(* (sqrt 2) (sin pi))`

c. `(+ 3 (sqrt (- 4)))`

d. `(* (sqrt 5) (sqrt (/ 3 (cos pi))))`

e. `(/ (sqrt 5) (sin (* 3 0)))`

f. `(/ (+ 3) (* 2 4))`

g. `(* 1 2 3 4 5 6 7 8)`

h. `(/ 120 2 3 2 2 5)`

3. En los ejercicios anteriores aparecen algunas funciones como `cos`, `sin`, `sqrt`.

Si la mayoría de estos nombres no le dicen nada, no se preocupe, no es necesario conocerlas por ahora.

Muchas funciones que quizás le resulten conocidas ya se encuentran predefinidas en *DrRacket*, tales como `log`, `tan`, `expt`, `random`, `max`, `min`, `floor`, `ceiling`, `abs`. Escriba expresiones que combinen algunas de estas funciones. Si no conoce su comportamiento, consulte la documentación o a los y las docentes.

3 Strings

Otro tipo de dato que generalmente encontramos en los lenguajes de programación es el *string* o *cadena de caracteres*.

Un *string* es una secuencia de caracteres encerrada entre comillas. Estos son algunos ejemplos.

Notar que

"3.14" es un *string* y

3.14 es un número

- "yagareté"
- "b"
- "45"
- "Pueblo que se somete, perece."

Así como existe una aritmética para los números, existe también una "*aritmética de strings*". Es decir, *racket* nos provee operaciones que manipulan strings. Una operación muy usada es *string-append*, que concatena strings. Aquí un ejemplo:

```
(string-append "Program" "ar")
```

```
==
```

```
"Programar"
```

Al igual que para la operación suma, la función *string-append* puede recibir más de dos argumentos:

```
(string-append "El " "oso " "sala " "la " "sopa.")
```

```
==
```

```
"El oso sala la sopa."
```

Algunas operaciones sobre strings pueden devolver otra clase de datos. Por ejemplo, la función *string-length* nos dice cuántos símbolos tiene un string:

```
(string-length "Primer año")
```

```
==
```

```
10
```

Incluso existen operaciones sobre números que devuelven strings. Por ejemplo:

```
(number->string 42)
```

```
==
```

```
"42"
```

A medida que avancemos con el curso veremos muchas otras operaciones que manipulan strings que nos serán de utilidad.

1. Evalúe en *DrRacket* las siguientes expresiones.

a. (string-append "Hola" "mundo")

b. (string-append "Pro" "gra" "ma.")

c. (number->string 1357)

d. (string-append "La respuesta es " (+ 21 21))

e. `(string-append "La respuesta es " (number->string (+ 21 21)))`

f. `(* (string-length "Hola") (string-length "Chau"))`

Recuerde que en Ciencias de la Computación contamos desde 0.

Explore la función `string-ith`, que, dados un string `s` y un número natural `n`, devuelve el caracter que ocupa la n -ésima posición en `s`.

Más interesante es la función `substring`. Aquí un ejemplo de cómo funciona:

```
(substring "Programar" 2 5)
```

```
==
```

```
"ogr"
```

Escriba expresiones que utilicen esta función ¿Qué sucede si el último argumento es menor que el penúltimo? ¿Y si son iguales? ¿Qué ocurre si el último argumento es mayor al largo del string? ¿Ocurre algo parecido si el penúltimo es un número menor a 0?

4 Valores de verdad

Los valores de verdad, o *valores booleanos* son fundamentales para la programación, y se encuentran presentes en cualquier lenguaje. En general, cuando un programa debe tomar una decisión, lo hace en función de un valor booleano.

Hay sólo dos valores de verdad, verdadero y falso, que en *DrRacket* escribimos de la siguiente forma, respectivamente: `#true` (o `#t`) y `#false` (o `#f`).

4.1 Algunas operaciones

Así como es posible utilizar operadores aritméticos para crear expresiones complejas, *DrRacket* nos provee *operadores booleanos*, que permiten formar nuevas expresiones a partir de otras más simples.

En esta práctica comenzaremos viendo tres operadores: `and`, `or` y `not`. Aprovechando que sólo hay dos valores booleanos, es fácil definir las operaciones indicando cuál es el resultado de la operación para cada caso posible.

El operador `and` es verdadero exactamente cuando sus argumentos son todos verdaderos. Es decir, la expresión `(and p q)` es verdadera si y sólo si tanto `p` como `q` evalúan a `#true`. Esta operación puede definirse mediante la siguiente tabla:

p	q	(and p q)
<code>#true</code>	<code>#true</code>	<code>#true</code>
<code>#true</code>	<code>#false</code>	<code>#false</code>
<code>#false</code>	<code>#true</code>	<code>#false</code>
<code>#false</code>	<code>#false</code>	<code>#false</code>

Esta tabla se denomina comúnmente *tabla de verdad* y se lee por filas. La primera fila se de la siguiente manera. "Si una proposición `p` evalúa a `#true` y una proposición `q` evalúa a `#true` entonces `(and p q)` evalúa a `#true`.

Del mismo modo, el operador `or` se define a través de la siguiente tabla:

p	q	(or p q)
<code>#true</code>	<code>#true</code>	<code>#true</code>

#true	#false	#true
#false	#true	#true
#false	#false	#false

Observe que $(\text{or } p \ q)$ es verdadero si al menos uno entre p y q es verdadero.

Finalmente, el operador `not` recibe un sólo argumento, y nos devuelve el valor *opuesto*:

p	(not p)
#true	#false
#false	#true

A partir de estas definiciones, ya tenemos todo lo necesario para calcular con valores de verdad. Consideremos la expresión

```
(and #true (or #false (not #false)))
```

Podemos proceder de la siguiente forma:

```
(and #true (or #false (not #false)))
```

== definición de `not`

```
(and #true (or #false #true))
```

== definición de `or`

```
(and #true #true)
```

== definición de `and`

```
#true
```

4.2 Mezclando booleanos y números

Muchas expresiones que involucran números evalúan a números. Por ejemplo, $(+ \ 1 \ 4)$ reduce a 5. Sin embargo, hay otras expresiones que utilizamos en matemática, tales como

$$3 < 2 + 2$$

que no representan un número, sino un valor de verdad (es decir, algo que es verdadero o falso).

En *DrRacket* escribimos $(< \ 3 \ (+ \ 2 \ 2))$, y si evaluamos esta expresión obtenemos `#true`, pues 3 es menor que 4.

Otras relaciones que ya vienen definidas en *DrRacket* son `>` (mayor), `=` (igual), `<=` (menor o igual), `>=` (mayor o igual).

1. Evalúe en *DrRacket* las siguientes expresiones:

a. $(\text{not } \#t)$

b. $(\text{or } \#t \ \#f)$

c. $(\text{and } \#t \ \#f)$

d. $(\text{and } \#t \ (\text{or } \#f \ (\text{not } \#f)) \ (\text{not } \#t))$

e. $(\text{not } (= \ 2 \ (* \ 1 \ 3)))$

f. $(\text{or } (= \ 2 \ (* \ 1 \ 3)) (< \ 4 \ (+ \ 3 \ 2)))$

2. Considere la siguiente frase:

"*pi* es un número mayor que 3, y $2 + 2$ es igual a 5".

Si quisiéramos representarla en *racket*, escribimos:


```
(and (> pi 3) (= (+ 2 2) 5))
```

Evaluando esta expresión podemos concluir que la frase es falsa. Se pide, para cada frase, escribir una expresión que la represente y calcular su valor de verdad:

- a. El coseno de 0 es positivo.
- b. La longitud de la cadena "Hola, mundo" es 14.
- c. *pi* es un número entre 3 y 4.
- d. El área de un cuadrado de lado 5 es igual a la raíz cuadrada de 625.
- e. No es cierto que el tercer caracter de la cadena "Ada Lovelace" es "a".




5 Imágenes

Además de números, strings y booleanos, es común que los programas manipulen imágenes. El lenguaje *racket* nos provee una gran variedad de funciones para manipular esta clase de información. Por ejemplo, si nos interesa saber cuántos píxeles de ancho tiene una imagen:


```
(image-width   
==  
28
```

Se pueden copiar y pegar imágenes del navegador (u otra aplicación) a *DrRacket*

Si queremos saber cuántos píxeles ocupa en total:

```
(* (image-width  (image-height   
==  
(* 28 (image-height   
==  
(* 28 42)  
==  
1176
```

También podemos crear nuestras propias imágenes. Por ejemplo, la operación `circle` produce una imagen a partir de un número y dos strings.

```
(circle 20 "solid" "green")  
==  

```

Se pueden crear toda clase de figuras geométricas, e iremos aprendiendo a lo largo del curso cómo hacerlo. Lo importante para notar ahora es que las imágenes son una clase de datos que, al igual que las vistas hasta ahora, pueden ser manipuladas a través de operaciones que las reciban como argumento (por ejemplo, `image-width`) o las producen como salida (por ejemplo, `circle`).

1. Este ejercicio presenta algunas expresiones con la intención de familiarizarse con imágenes. Modifique estas expresiones para observar el comportamiento de las funciones aquí presentadas.

- a. `(circle 10 "solid" "red")`
- b. `(rectangle 10 20 "solid" "blue")`
- c. `(rectangle 20 12 "outline" "magenta")`
- d. `(overlay (rectangle 20 20 "solid" "blue") (circle 7 "solid" "green"))`
- e. `(empty-scene 100 100)`
- f. `(place-image (circle 10 "solid" "blue") 40 80 (empty-scene 100 100))`
- g. `(+ (image-width (circle 10 "solid" "red")) (image-height (rectangle 10 20 "solid" "blue")))`

6 Funciones y Constantes

Probablemente se hayan familiarizado con el concepto de función en cursos de matemática. En general, una función se presenta como una operación (o regla, o ley) que asocia a cada valor en un conjunto (llamado dominio o conjunto de partida) un único valor en otro conjunto (llamado codominio o conjunto de llegada). Por ejemplo, si consideramos el conjunto de números naturales, la definición

$$f(x) = x + 1$$

es una función que asocia a cada valor de entrada su sucesor.

El concepto de función no es distinto en programación: las funciones toman valores o elementos de entrada y producen valores o elementos de salida. Estos valores o elementos pueden ser de diferentes tipos: números, strings, booleanos, imágenes y combinaciones de todos ellos.

Los programas son, en definitiva, funciones que manipulan diferentes tipos de datos. Los lenguajes de programación establecen una cierta sintaxis que permite definir dichas funciones.

En particular, en *racket* los programas están constituidos por definiciones que son de dos clases: constantes y funciones.

6.1 Constantes

Definir una constante consiste en establecer que un cierto identificador o nombre será siempre y únicamente sustituido por un cierto valor. La definición de una constante tiene la siguiente forma.

Notar que en `(define abc "abc")` `abc` es un nombre y `"abc"` es una cadena de caracteres.

¿Qué pasaría si escribiésemos `(define "abc" abc)` o `(define abc abc)`?

`(define <identificador> <expresión>)`

Por ejemplo,

`(define A 3)`

En este caso, cada vez que el identificador "A" aparezca en una expresión, éste será reemplazado por 3. Esto es,

`(* A 10)` será `(* 3 10)`

Observe los siguientes ejemplos y pruébelos en *DrRacket* .

Aunque es posible usar minúsculas, en general, utilizaremos MAYÚSCULAS para escribir el nombre de las constantes.

- `(define P "Neptuno")`
- `(define ITRES 3)`
- `(define CINCO (+ 1 (* 2 2)))`
- `(define VERDADERO #true)`
- `(define PUNTOROJO (circle 3 "solid" "red"))`

6.2 Definiendo funciones

Para definir una función debemos dar su nombre, explicitar los argumentos que recibirá como entrada, así como también indicar su "ley" mediante una expresión. La forma general de la definición de una función es la siguiente.

`(define (<identificador> <argumento 1> ... <argumento n>) <expresión>)`

Observe cómo se definen las siguientes funciones en *racket* . Pruébelas en *DrRacket* .

- La función sucesor, la definimos como

```
(define (f x) (+ x 1))
```

- La función que calcula el doble de un número:

```
(define (doble x) (* x 2))
```

- "Función que establezca si un número es menor que el doble de otro.":

```
(define (h x y) (< x (* 2 y)))
```

Observar que para definir *h* podemos **componer funciones** y usar la función *doble* ya definida. Esto es,

```
(define (h x y) (< x (doble y)))
```

- "Función que dibuje un cuadrado azul de una dimensión dada."

la definimos como

```
(define (cuad-azul x) (square x "solid" "blue"))
```

Dar a las funciones un nombre representativo de su comportamiento constituye una buena práctica de programación.

Por ejemplo, en nuestra primer función, el nombre *sucesor* es más apropiado que *f*.

En general, utilizaremos minúsculas para escribir el nombre de las funciones.

Otra buena idea puede ser separar con - las palabras de un nombre, siguiendo el estilo de *racket*, como se hizo con *cuad-azul*.

Resuelva los siguientes ejercicios.

1. Utilizando las definiciones de más arriba, evalúe las siguientes expresiones:


```
(cuad-azul (doble 10))
```

```
(and (h 2 3) (h 3 4))
```

```
(= (f 1) (doble 1))
```

2. Defina una función que recibe dos números x e y , y devuelve la distancia al origen del punto (x,y) .
3. Defina una función que recibe cuatro números x_1 , y_1 , x_2 e y_2 , y devuelve la distancia entre los puntos (x_1, y_1) y (x_2, y_2) .
4. Defina la función `vol-cubo` que recibe la longitud de la arista de un cubo y calcula su volumen.
5. Defina la función `area-cubo` que recibe la longitud de la arista de un cubo y calcula su área.
6. Defina una función `metro-pie`, que dada una longitud expresada en metros, devuelva su equivalente en pies.
7. Defina una función `cel-far`, que dada una temperatura expresada en Celsius, devuelve su equivalente en Fahrenheit.
8. Defina una función `posible?` que, dados tres números positivos a , b , c , devuelve `#true` si es posible construir un triángulo de lados a , b , c . Caso contrario, devuelve `#false`. Por ejemplo, `(posible? 1 2 5)` debe evaluar a `#false`, pues no es posible construir un triángulo de lados 1, 2 y 5.

9.

¿No sabe lo que es una terna pitagórica? Wikipedia puede brindarle una ayuda.

Defina una función `pitagórica?` que dados tres números, devuelve `#true` si estos forman una *terna pitagórica*. Caso contrario, devuelve `#false`.

0. Defina una función `suma-long`, que dados dos strings devuelve la suma de sus longitudes.
1. Defina una función `comienzaA?` que dado un string devuelve `#true` si el string comienza con `"A"`.
2. Defina la función `poner-` que recibe un string y un número i e inserta `"-"` en la posición i -ésima del string.