Practica 0 - IntroRacket

Programa: conjunto de expresiones válidas escritas en un lenguaje de programación, las cuales podrán ser computadas de acuerdo a la semántica definida por el lenguaje. Los lenguajes de programación establecen una cierta sintaxis que permite definir dichas funciones.

Expresiones: Una operación tiene la forma (Racket prefija)
(<operador> <operando 1> <operando n>) Ejemplo: (+ 1 2) (* 1 2 3 4).

Función: es una operación que asocia a cada valor en un conjunto un único valor en otro conjunto. Las funciones toman valores o elementos de entrada y producen valores o elementos de salida. f(x) = x + 1.

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

Constante: establece que un cierto identificador o nombre será siempre y únicamente sustituido por un determinado valor.
(define <identificador> <expresión>).
(define A 3).

Vocabulario: palabras o símbolos del lenguaje. +, -, *, /, define, (,), etc.

Sintaxis o gramática: reglas que indican como se combinan estas palabras para armar frases validas. (+ 1 2) es una expresión valida en Racket.

Semántica: significado de las frases gramaticalmente correctas.
(+ 1 2) representa al nro. 3.

Vocabulario básico:

- Valores:
    - Numéricos.
    - Booleanos.
    - Cadenas.
    - Imágenes.
- Primitivas:
    - Palabras claves: define, if, cond, etc.
    - Operadores matemáticos: +, /, *, sqrt, sqr, <=, >=, =.
    - Operadores sobre cadenas: String-append, String-length.
    - Operadores booleanos: and, or, not, etc.

Tipos de datos: hacen referencia a la clase de información que se va a manejar.

Tipo	Predicado	Comparación
Number	number?	=
String	string?	string=?
Boolean	boolean?	boolean=?
Image	image?	image=?

Predicado: un predicado es una función booleana.
Sean $T_1 \cdot \cdot \cdot T_n$ tipos de datos, si f es una función con tipo:
; f: $T_1 \cdot \cdot \cdot T_n$ -> Boolean
entonces f es un predicado.

Las proposiciones nos permitirán tomar decisiones en un programa.
(if ⟨condición⟩ ;es una proposición
⟨exp1 ⟩ ;si la condición es V
⟨exp2 ⟩) ;si la condición es F

Practica 1 - Condicionales

Leyes de reducción:

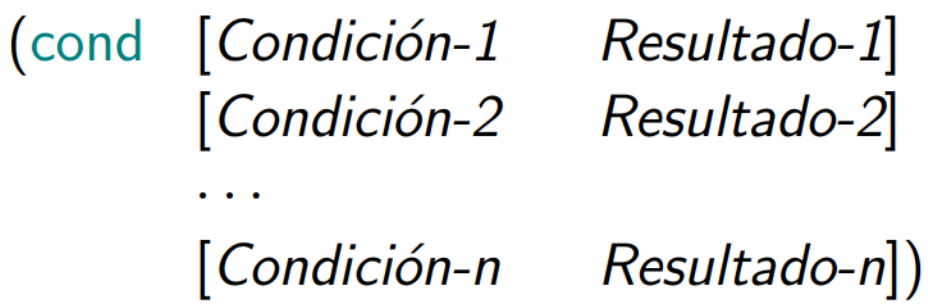
Ley 1:

```
(if #t a b)
==  definición de if (ley 1)
a
```

Ley 2:

```
(if #f a b)
==  definición de if (ley 2)
b
```

Si un programa tiene que tomar más de una decisión usaremos la sentencia cond en lugar de varios if anidados.



La expresión condicional está formada por una lista de parejas, la primera parte de cada pareja (Condición-X) es una expresión que debe evaluar a un valor de tipo booleano. La segunda parte de la pareja se conoce como el resultado de la condición correspondiente, y puede evaluar un valor de cualquier tipo de dato.

En lugar de una sola condición se tienen n condiciones booleanas.

Evaluación:

- Si Condición-1 evalúa a #t, toda la expresión evalúa a Resultado-1.
- Si Condición-1 evalúa a #f y Condición-2 evalúa a #t, toda la expresión evalúa a Resultado-2.
- Se evalúan las condiciones en orden hasta que una evalúe a #t.
- Si todas evalúan a #f se produce un error.

Practica 2

Un programa bien diseñado es aquel que viene acompañado de una explicación de lo que hace, aquel que explica qué tipo de entradas espera y qué resultados produce.

La receta:

- **Diseño de Datos**: En este paso, definimos la forma de representar la información como datos.

- **Signatura y declaración de propósito**: la signatura de una función indica qué datos consume (cuántos y de qué clase), y qué datos produce.

  Ejemplos:

  ; Number -> Number

Signatura para una función que consume un número y produce un número.

; Number String String -> Image

Signatura para una función que consume un número y dos String, y devuelve una imagen.

La **declaración de propósito** consiste en una breve descripción del comportamiento de la función.

- **Ejemplos**: partir de los dos primeros pasos, debemos estar en condiciones de predecir el comportamiento de la función para algunas entradas.

- **Definición de la función**: en este paso ya escribimos el código que creemos resuelve el problema planteado.

- **Evaluar el código en los ejemplos**: una vez que contamos con el código de la función, la ejecutamos en los ejemplos propuestos para verificar que (al menos) funciona para esas entradas cuyas respuestas esperadas conocemos.

- **Realizar modificaciones en caso de errores**: si en el paso anterior tenemos diferencias entre las respuestas esperadas y las respuestas obtenidas al ejecutar la función, debemos buscar la fuente del error.

Algunos lineamientos generales

Principios de la Ingeniería de Software importantes a tener en cuenta al momento de diseñar programas.

▶ Modularidad

Pensar un programa como un conjunto de partes que se combinan para constituir un todo.

- descomponer un programa complejo en piezas más simples.
- componer un programa a partir de módulos existentes.
- modificar el programa, modificando sólo un número pequeño de piezas.
- comprender el programa en términos de sus partes.
- testear por separado las diferentes porciones del programa.

▶ Anticipación al cambio

Mientras diseñamos, nos adelantamos a los cambios que el programa pueda sufrir, aislando los cambios en porciones específicas del programa.

- Si surgen los cambios, llevarlos a cabo implica poco esfuerzo.

Practica 3 – Programas Interactivos

Los programas llamados programas por lotes, que una vez lanzado el proceso no necesitan ningún tipo de interacción con el usuario; y los programas interactivos, que una vez iniciados esperan la intervención del usuario para llevar a cabo sus funciones.

Evento: los programas reciben información de su entorno a través de eventos, en programación llamamos evento para describir una ocurrencia de alguna situación para la que un programa está preparado y debe tomar una determinada acción.

La idea fundamental de un programa interactivo es que ante cierto evento el programa llevará a cabo cierta acción que está establecida a partir de la definición de una función llamada manejador de eventos; será una función que, dado un argumento, devuelve la acción a efectuarse.

Estado: durante la ejecución de un programa, ante ciertos eventos, los correspondientes manejadores de eventos llevan a cabo acciones que cambian determinadas propiedades o valores dentro del programa, el estado. Durante la ejecución de un programa interactivo, el programa estará en un estado particular, el cual cambiará ante la aparición de un evento.

El mecanismo para establecer la relación evento - manejador de evento e indicarle a DrRacket cómo tratar un evento, es a través de la función 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.

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 asociado a la cláusula stop-when al nuevo estado. Si devuelve #true, el programa termina, si no, 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).

Practica 4 – Estructuras

Estructura posn:
(define-struct posn [x y])
Un elemento posn representa una posición de coordenadas cartesianas.

- Constructor `make-posn`

```
; make-posn : Number Number -> posn
(make-posn 3 5)      ⟶      x   y
                             3 | 5
```

- Selector `posn-x`

```
; posn-x : posn -> Number
(posn-x (make-posn 3 5))    ⟶    3
```

- Selector `posn-y`

```
; posn-y : posn -> Number
(posn-y (make-posn 3 5))    ⟶    5
```

- Predicado `posn?`

```
; posn? : Any -> Boolean
(posn? (make-posn 3 5))    ⟶    #true
(posn? 3)    ⟶    #false
```

Estructuras:

```
(define-struct <nombre> [ <campo1> ... <campoN> ])
```

siendo

- **<nombre>** el identificador o nombre de la estructura.
- **<campoi>**, con **i:1..N**, un identificador o nombre de un campo de la estructura.

Esta definición **genera automáticamente** la definición de **otras funciones** asociadas.

- constructor: `make-<nombre>`
- selectores: `<nombre>-<campoi>` con **i:1..N**
- predicado: `<nombre>?`

Practica 5 p1 - Listas

Datos de tamaño fijo:

- Atómicos: números, booleanos, cadenas, imágenes.
- Estructuras: nos permiten unir datos.

Una lista es un tipo de datos autorreferencial.
Una Lista-de-números es o bien:
- '() ,
- (cons Number Lista-de-números)
Las definiciones autorreferenciales nos permiten construir datos de tamaño arbitrario.

Operaciones con listas:

- Constructores:
  - ('() constante que representa la lista vacía.
  - cons agrega un elemento a la lista.
- Selectores:
  - first devuelve el primer elemento de la lista.
  - rest devuelve la lista sin el primer elemento.
- Predicados:
  - empty? determina si la lista es vacía.
  - cons? determina si la lista es no vacía.

Practica 5 p2 - Patrones

Dado un predicado p y una lista l con objetos en X, queremos devolver una lista con aquellos objetos de l para los cuales p evalúa a #true.

```
; filter : (X -> Boolean) List (X) -> List (X)
; Dado un predicado p y una lista l construye una
; lista con los elementos de l que satisfacen p.

(check-expect (filter even? (list 1 2 3 4)) (list 2 4))
(check-expect (filter odd? '()) '())

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

Dada una función f, la función map transforma una lista [a_0, …, a_n] en [f (a_0), …, f (a_n)].
Dada una función que transforma objetos de X en objetos de Y, y una lista con objetos en X, devuelve una lista con objetos en Y.

```
; map : (X -> Y) List (X) -> List (Y)
; dada una función f que transforma elementos de X
; en elementos de Y y una lista de elementos en X
; devuelve el resultado de aplicar f sobre
; cada elemento de la lista.

(check-expect (map sqr (list 1 2 3 4)) (list 1 4 9 16))
(check-expect (map sqr '()) '())

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

Dado un operador $\oplus$ y un valor c (neutro) la función foldr convierte la lista: [a_0, . . ., a_n] en $a_0 \oplus (a_1 \oplus \cdots \oplus (a_n \oplus c))$.

La función fold recibe tres argumentos:

- La función f con la que se quiere operar los elementos de la lista.
- Un valor c, que es el resultado esperado para la lista vacía.
- La lista l a transformar.

```
;foldr : (X Y -> Y) Y List (X) -> Y

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

Practica 6 - Naturales

Los Números Naturales son aquellos que nos permiten contar.

Los números naturales tienen su propia definición de tipo:
; Un Natural es:
; – 0
; – (add1 Natural)
; interpretación: Natural representa los números naturales

Operador	Tipo de Operador	Función
`0`	Constructor	Constante usada para representar el primer número natural
`add1`	Constructor	Calcula el sucesor de un número natural
`sub1`	Selector	Devuelve el predecesor de un número natural positivo
`zero?`	Predicado	Reconoce al natural 0
`positive?`	Predicado	Reconoce naturales construidos con add1