Tarea 4

Mutación y macros

The Last

Para la resolución de la tarea recuerde que

- Desarrolle las preguntas conceptuales en un archivo Markdown `P1.md` o de texto plano `P1.txt`.
- Toda función debe estar acompañada de su firma, una breve descripción coloquial (en `P2.rkt`, `P3.rkt`) y un conjunto significativo de tests (en `P2tests.rkt`, `P3tests.rkt`).
- Todo datatype definido por el usuario (via `deftype`) debe estar acompañado de una breve descripción coloquial y de la gramática BNF que lo genera.

Si la función o el datatype no cumple con estas reglas, será ignorado.

Ejercicio 1 **10 Pt**

Resuelva las siguientes preguntas conceptuales:

(a) [3 Pt] ¿En una invocación a función, qué diferencias existen entre pasar los argumentos por valor y pasarlos por referencia?

(b) [2 Pt] Considere el siguiente intérprete de un lenguaje con variables y mutación, escrito usando store-passing style:

```
(define (interp expr env sto)
  (match expr
    ...
    [(app fun-expr arg-expr)
    (def (closureV id body fenv) (interp fun-expr env sto))
    (def loc (env-lookup (id-x arg-expr) env))
    (interp body (extend-env id loc fenv) sto)]
    ... ))
```

¿Qué problema tiene este intérprete? Escriba un programa que ilustre que la semántica del app es errónea. Arregle el interprete y luego escriba un test que entregue resultados distintos si la estrategia de paso de parámetros es Call by Value o Call by Reference.

(c) [2 Pt] Cite 3 escenarios generales en los cuales es conveniente usar macros; Luego, considere estos escenarios en el caso de Haskell. ¿Tendría sentido desarrollar un sistema de macros para Haskell en dichos casos?

(d) [3 Pt] ¿Qué significa que las macros de Racket sean higiénicas? De un ejemplo donde la falta de higiene en las macros pudiese traer problemas y luego explique qué se podría hacer para implementar higiene en un sistema de macros que no lo es.

Ejercicio 2 **27 Pt**

Comenzaremos con el lenguaje definido `base.rkt`, el cual provee funciones de primera clase con *scope* estático y evaluación temprana. En esta pregunta se espera que modifique todo lo necesario para añadir lo que se solicite.

(a) [2 Pt] Extienda el lenguaje, a modo de permitir secuenciación de instrucciones (`seq`) como un azúcar sintáctico de una aplicación de función. Para ello considere que `(seq (+ 1 2) 5)` equivale a `(with ( _ (+ 1 2)) 5)`. Para ser más específicos, la instrucción `seq` ejecuta su expresión izquierda, descarta el resultado, y luego evalúa su expresión derecha retornando su resultado.

(b) [2 Pt] Ahora, añadiremos mutación al lenguaje. Para ello es necesario extenderlo con la instrucción `(set <id> <val>)`, que permita mutar el valor de una *id* específica, retornando luego el valor de tal *id*. Lo anterior debería permitir el funcionamiento de:

```
> (run '(with (x 9)
        (if0 (seq (set x 2) (- 3 3))
             (+ 1 x)
             (- 14 x))))
(numV 3)
```

No puede usar las *box*, ni ningún tipo de mutación nativa de Racket, en su lugar utilice un *store-passing style* como fue visto en clases.

(c) [8 Pt] Extienda las funciones de manera que se puedan recibir múltiples argumentos, como en el siguiente ejemplo.

```
> (run '(with (f (fun (x y z) (+ (- x y) z)))
        (f (8 3 7)))
```

Es decir, la estructura abstracta de `Expr` quedaría como:

```
#|<expr> ::= ...
         | (fun <sym list> <expr>)
         | (app <expr> <expr list>) |#
```

Hint: Necesitará estandarizar la evaluación y paso de parámetros para que operen con listas de valores y de ids en lugar de solo un valor y una id, puede venir bien crear funciones adicionales que se encarguen de tal labor. Asuma que al inovar una función no se producen *arity-mismatches*, y que sus argumentos formales se evalúan de izquierda a derecha.
¿Se le ocurre otra forma de implementar funciones de múltiples argumentos? Añádalo como comentario bajo la definición de `Expr` (*ie:* no lo implemente).

(d) [15 Pt] En el lenguaje C (y variantes), es posible aplicar algo similar a *call by reference* pasando la dirección de una variable `v` (con `&v`), permitiendo que la mutación ejercida al interior de una función pueda tener efecto fuera de ella. El objetivo es implementar algo similar en nuestro lenguaje, para ello consideremos los siguientes dos ejemplos:

Ejemplo 1: `a` se pasa *por valor* en la invocación a `f`.

```
> (run '(with (f (fun (x y z) (seq (set y 10) (+ x z))))
              (with (a 3)
                    (+ (f (8 a 5)) a))))
(numV 16)
```

Ejemplo 2: `a` se pasa *por referencia* en la invocación a `f`.

```
> (run '(with (f (fun (x y z) (seq (set y 10) (+ x z))))
              (with (a 3)
                    (+ (f (8 &a 5)) a))))
(numV 23)
```

Ayuda: Para este ítem pueden ser de utilidad las siguientes funciones:

```
;; is-ref-expr ?:: sym -> boolean
;; Retorna verdadero si un simbolo corresponde a una expresion de referencia
    ('&var), o falso sino.
(define (is-ref-expr ? expr)
    (def symstr (symbol->string expr))
    (equal? (string-ref symstr 0) #\&))

;; get-id:: sym -> sym
;; Retorna el valor de una id de referencia ('&var) en su id real ('var).
(define (get-id id)
    (string->symbol (substring (symbol->string id) 1)))
```

Ejercicio 3 **23 Pt**

Un ratón de laboratorio puede estar haciendo tres cosas: durmiendo, comiendo, o corriendo en la rueda. Se ha observado empíricamente que, luego de dormir, el ratón pasa a comer con probabilidad 2/3 y a correr con probabilidad 1/3. Luego de comer, pasa con probabilidad 3/4 a dormir y 1/4 a correr. Finalmente, luego de correr pasa a dormir con probabilidad 1/2 y a comer con probabilidad 1/2. La dinámica del ratón puede representarse a través de un sistema de transición probabilístico —conocido como una *cadena de Markov*— que se muestra en la Figura 1.

Si en un instante t_n el ratón está durmiendo con probabilidad p_s, comiendo con probabilidad p_e y corriendo con probabilidad p_r —lo que representaremos con la terna (p_s, p_e, p_r) también llamada *estado probabilístico* o simplemente *estado*—, en el instante t_{n+1} estará en un estado dado por la función:

$$\mathsf{step}(p_s, p_e, p_r) \;=\; \left(\tfrac{1}{2}p_r + \tfrac{3}{4}p_e,\; \tfrac{2}{3}p_s + \tfrac{1}{2}p_r,\; \tfrac{1}{3}p_s + \tfrac{1}{4}p_e\right)$$

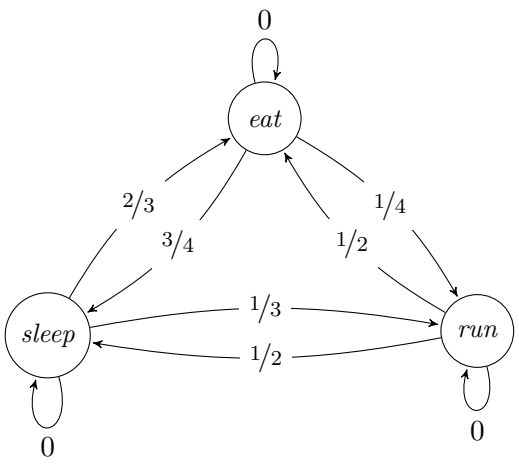

Figura 1: Cadena de Markov representando la dinámica del ratón.

Puede probarse que independientemente del estado inicial del que parta, después de un tiempo lo suficiente largo el ratón se estabilizará en un estado de equilibrio (p_s^*, p_e^*, p_r^*) caracterizado por las ecuaciones:

$$\begin{cases} \mathsf{step}(p_s^*, p_e^*, p_r^*) = (p_s^*, p_e^*, p_r^*) \\ p_s^* + p_e^* + p_r^* = 1 \end{cases} \tag{1}$$

> El objetivo de este ejercicio es definir una macro `equilibrium` que tome como entrada la especificación de una cadena de Markov y al expandirse genere un programa que calcula su estado de equilibrio a largo plazo.

Las cadenas de Markov se especificarán como muestra el siguiente ejemplo (de la Figura 1):

```
(define mouse
  (equilibrium
   [run   : (0   → run)
            (1/2 → eat)
            (1/2 → sleep)]
   [eat   : (1/4 → run)
            (0   → eat)
            (3/4 → sleep)]
   [sleep : (1/3 → run)
            (2/3 → eat)
            (0   → sleep)]))
```

Observe que para cada estado (*run*, *eat* y *sleep*) se provee la probabilidad de transición a todos los estados restantes y a él mismo (independientemente de si la probabilidad es nula o no). Puede suponer además que dichas probabilidades se enumeran siguiendo

el mismo orden en que se listan los estados. Por ejemplo, en la especificación de arriba los estados se listan *run*, *eat*, *sleep*, y para cada uno de éstos, las probabilidades se enumeran siguiendo el mismo orden.

El programa al que se expande la macro debe devolver una lista, describiendo el estado de equilibrio a largo plazo, siguiendo el mismo orden en el que se especificó la cadena de Markov:

```
> mouse
'(12/53 20/53 21/53)
```

Es decir, $12/53$ representa la probabilidad de equilibrarse en el estado *run*, $20/53$ en el estado *eat* y $21/53$ en el estado *sleep*.

Para resolver el sistema de ecuaciones (1), le puede resultar útil la librería de matrices provista por Racket.