Lenguajes de Programación (2020/2)
Profesor: Federico Olmedo
Auxiliares: Bryan Ortiz Tomás Vallejos
Ayudantes: G. Iturra V. Illanes M. Rojas

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Tarea 3

Estrategias de evaluación y recursión

Para la resolución de la tarea recuerde que

- Toda función debe estar acompañada de su firma, una breve descripción coloquial (en `P1.rkt`, `P2.rkt`) y un conjunto significativo de tests (en `P1tests.rkt`, `P2tests.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 — 20 Pt

En este ejercicio vamos a partir del lenguaje con evaluación perezosa y scope estático definido el archivo `base.rkt`, y vamos a extenderlo para incorporar listas. Para operar expresiones aritméticas, además de la suma y la resta, vamos a incluir también el cociente. Para ello asuma que las constantes numéricas usadas en los programas van a ser siempre enteros, y para el cociente considere la división entera, como se muestra los siguientes dos ejemplos:

```
> (run '(with (halve (fun (n) (/ n 2)))
              (halve 7)))
(numV 3)

> (run '(with (halve (fun (n) (/ n 0)))
              (halve 8)))
quotient: undefined for 0
```

Para construir (introducir) listas, el lenguaje debe proveer los constructores `cons` y `nil`, y para destruir (eliminar) listas, un operador de *pattern matching* sobre listas:

```
> (run '(with (l (cons 1 (cons 2 (cons 3 nil))))
              (match l as
                (nil  => 0)
                (cons x xs => (+ x 5)))))
(numV 6)
```

Lo fundamental de esta extensión del lenguaje es que las listas son *perezosas*. Con esto queremos decir que al construir una lista, sus elementos no se evalúan, sino que se "almacenan" tal cual le son dados. Los elementos de las listas se evaluarán sólo al eliminar la lista (i.e. "inspeccionarla" a través de una primitiva de *pattern matching*), y sólo si la forma en que se eliminen así lo requiera. Por ejemplo, considere el siguiente programa:

```
> (run '(with (l (cons 1 (cons (/ 2 0) nil)))
         (match l as
           (nil => 0)
           (cons y ys => (+ 10 y)))))
(numV 11)
```

El programa no lanza un error (de dividir 2 por 0), ya que para calcular el resultado del *pattern matching* sólo se necesita evaluar el primer elemento de la lista.
De la misma manera, el programa de abajo tampoco lanza un error ya que para calcular el resultado del *pattern matching* sólo se debe determinar si la lista es vacía o no, y *no* cuál es el valor de sus elementos:

```
> (run '(with (l (cons (/ 1 0) (cons (/ 2 0) nil)))
         (match l as
           (nil => 0)
           (cons z zs => 1))))
(numV 1)
```

El siguiente ejemplo también ilustra que al crear una lista, la evaluación de sus elementos queda en suspenso:

```
> (run '(with (l (cons (+ 1 2) (cons 3 nil)))
              (match l as
                (nil => 0)
                (cons y ys => y))))
(exprV (add (num 1) (num 2)) (mtEnv) '#&#f)
```

Recuerde que la suma fuerza la reducción de los sumandos (en un lenguaje perezoso estos puntos donde se fuerza la reducción de subexpresiones eran llamados los *strictness points* del lenguaje). Por eso en el segundo ejemplo que vimos se reduce la expresión `(+ x 5)`, mientras que en el ejemplo de arriba (como no hay ninguna operación que fuerce ninguna reducción), se devuelve simplemente la clausura de `(+ 1 2)`.

Otro ejemplo que le puede resultar relevante para comprender, en particular, el manejo de entornos es el siguiente:

```
> (run '(with (x 1)
         (with (l (cons (+ 5 x) (cons 200 nil)))
         (match l as
           (nil => 0)
           (cons x xs => (+ 10 x))))))
(numV 16)
```

Observe que al reducir `(+ 10 x)`, `x` está asociado a la cabeza de la lista `l` (y no a `1` como establece el *binding* en la definición local *top-level*).
Finalmente, el siguiente ejemplo ilustra el anidamiento de listas:

```
> (run '(with (l (cons (cons 1 (cons (/ 2 0) nil))
                       (cons (cons 3 nil) nil)))
              (match l as
```

```
                (nil  => 0)
                (cons x xs => (match x as
                                (nil  => 1)
                                (cons y ys => y))))))

(exprV (num 1) (mtEnv) '#&#f)
```

Aquí `l` representa una lista, cuyos dos elementos son listas de números, concretamente `(cons 1 (cons (/ 2 0) nil))` y `(cons 3 nil)`.

> Extienda el lenguaje en `base.rkt` para que soporte listas perezosas como se describió recién. El resultado final de esta parte debe ser la función `run`.

Hint: va a tener que extender el conjunto de valores para incorporar listas. Dado que al construir (introducir) una lista, la evaluación de sus elementos queda en suspenso, piense cuál va a ser *el valor* de una lista. Le puede servir de ayuda recordar cuál es *el valor* de una función y porqué se define de esa manera.

Ejercicio 2 — 20 Pt

(a) [10 Pt] En este ejercicio se pide que extienda el lenguaje del Ejercicio 1 con el operador `rec` que permite introducir *bindings* recursivos. Puede usar el operador, por ejemplo, para definir funciones recursivas. El siguiente programa define la función `len` que calcula el largo de una lista, y se la aplica a una función de largo 3:

```
> (run '(rec (len (fun (l) (match l as
                         (nil  => 0)
                         (cons x xs => (+ 1 (len xs))))))
    (len (cons 1 (cons 2 (cons (/ 2 0) nil))))))
(numV 3)
```

Asimismo, podemos usar el operador `rec` para definir listas (cíclicas) infinitas. Por ejemplo el siguiente programa define una lista infinita de `1`'s y calcula la suma de sus dos primeros elementos:

```
> (run '(rec (ones (cons 1 ones))
         (match ones as
             (nil  => 0)
             (cons x xs => (match xs as
                             (nil  => 0)
                             (cons y ys => (+ x y)))))))
(numV 2)
```

Finalmente el siguiente programa ilustra el uso de la función `taken` que devuelve los primeros `n` elementos de una lista:

```
'(with (l (cons (+ 1 1) (cons (+ 2 2) (cons (+ 3 3) nil))))
         (rec (taken (fun (n)
                (fun (l) (if0 n
                             nil
                             (match l as
                               (nil  => nil)
                               (cons x xs => (cons x ((taken (- n 1))
                                   xs))))))))
           ((taken 2) l))))
```

Si corremos este programa, obtendremos un valor (bien difícil de entender, pero) que básicamente representa una lista de dos elementos, cuya evaluación está en suspenso.

(b) [10 Pt] Defina una función `deeprun` tal que al correr un programa, el resultado que devuelva no contenga ningún cómputo en suspenso. Para testear la función, corra el programa de arriba y los 4 programas de abajo, y verifique que obtenga el resultado esperado (parte de la consigna es que Ud. mismo determine cuál es dicho valor):

```
'(with (l (cons (cons (+ 1 2) nil)
                (cons (cons (+ 3 4) nil) nil)))
         l))
```

```
'(with (l (cons (+ 1 2) (cons 3 nil)))
   (match l as
     (nil  => 0)
     (cons y ys => y))))
```

```
'(with (l (cons (+ 1 1) (cons (+ 2 2) (cons (/ 3 0) nil))))
         (rec (taken (fun (n)
                (fun (l) (if0 n
                             nil
                             (match l as
                               (nil  => nil)
                               (cons x xs => (cons x ((taken (- n 1))
                                   xs))))))))
           ((taken 2) l))))
```

```
'(rec (ones (cons 1 ones))
  (rec (taken (fun (n)
         (fun (l) (if0 n
                      nil
                      (match l as
                        (nil  => nil)
                        (cons x xs => (cons x ((taken (- n 1)) xs))))))))
    ((taken 4) ones))))
```

Ejercicio 3 **20 Pt**

(a) [3 Pt] Considere la implementación naive en C++ de la función $f\colon \mathbb{N}_0 \to \mathbb{Z}$

$$\begin{aligned} f(0) &= 0 \\ f(n) &= \begin{cases} f(n-1)+n & \text{si } n \text{ es par} \\ f(n-1)-n & \text{si } n \text{ es impar} \end{cases} \end{aligned}$$

dada en el archivo `P3.cpp`. Compile el archivo fuente, y experimente con distintos *inputs* a f (llamada `naive_f` en la implementación) hasta determinar un valor que "agote" el espacio reservado para la pila (produciendo un *segmentation fault*). ¿Qué valor observó que "agota" la pila?

(b) [17 Pt] Dé una implementación recursiva de cola de f (llámela `smart_f`). Compile el archivo fuente activando la optimización TCO (*tail call optimization*) y verifique que `smart_f` funcione como esperado para el input que respondió en el apartado anterior.