Tarea 2

Esta tarea se distribuye con dos ficheros start.rkt y tests.rkt (base-tarea2). Considere las definiciones del archivo start.rkt y escriba sus funciones en él. Escriba sus tests en el archivo tests.rkt adjunto. Ambos ficheros deben ser entregados vía U-Cursos. Los tests forman parte de su evaluación! Consulte las normas de entrega de tareas en http://pleiad.cl/teaching/cc4101.

En esta tarea se le provee el lenguaje MiniScheme+, que corresponde a un intérprete extendido con estructuras de primera clase y pattern matching[1]. Además, MiniScheme+ incorpora el uso de primitivas[2]. A continuación se presenta un breve tour de las características de MiniScheme+:

1. *Funciones de primera clase con argumentos múltiples*: las funciones y las expresiones `with` pueden tener 0 o más argumentos, por ejemplo:

```
> (run '((fun (x y z) (+ x y z)) 1 2 3))
6
> (run '(with ((x 1) (y 2) (z 3)) (+ x y z)))
6
> (run '(with () ((fun () 42))))
42
```

2. *Definiciones usando `local`, `define` y `datatype`*: la expresión `local` permite realizar definiciones de identificadores y de estructuras de datos, usando `define` y `datatype` respectivamente. Por ejemplo:

```
> (run '(local ((define x 1)
                (define y 2))
          (+ x y)))
3
> (run '(local ((datatype Nat
                  (Zero)
                  (Succ n)))
          (Nat? (Zero))))
#t
> (run '(local ((datatype Nat
                  (Zero)
                  (Succ n))
                (define pred (fun (n)
                               (match n
                                 ((case (Zero) => (Zero))
                                  (case (Succ m) => m))))))
          (pred (Succ (Succ (Zero))))))
(Succ (Zero))
```

Observe que `define` y `datatype` sólo pueden usarse en la zona de declaraciones de una expresión `local`. Al declarar una estructura, la implementación extiende el ambiente usado en el cuerpo de `local` con las funciones constructoras de cada variante; y con predicados para determinar si un valor corresponde a la estructura (en general, y para cada variante). Para más detalles, consulte la implementación y tests provistos.

Warm-up (0.5 ptos) Si ejecutan el último ejemplo, veran que el output no es `(Succ (Zero))` sino `(structV 'Nat 'Succ (list (structV 'Nat 'Zero empty)))`. Luego de estudiar el código entregado para entender como se implementan las estructuras de datos, definan una función de pretty-printing para que las estructuras se representen al usuario (como resultado de `run`) tal como esperado.

Listas (1.5 ptos)

1. (0.2) Defina en MiniScheme+ el tipo de dato inductivo `List`, con dos constructores `Empty` y `Cons`, y la función recursiva `length` que retorna el largo de una lista.
2. (0.2) Con el objetivo de no tener que introducir la estructura de datos `List` en cada ejemplo, modifique la función `run` para que evalué la expresión en un contexto donde se tiene (al menos) la definición de `List` y de `length`.

```
> (run '(List? (Empty)))
#t
```

3. (0.5) Extienda el lenguaje para soportar la notación `{list e1 e2 … en}` como *azúcar sintáctico* para `{Cons e1 {Cons e2 … {Cons en {Empty}}…}}`:

```
> (run '(match (list (+ 1 1) 4 6)
          ((case (Cons h r) => h)
           (case _ => 0)))
2
```

   No necesita modificar ni el AST ni el interprete.

4. (0.4) Extienda ahora el pattern matching para que se pueda usar la notación `{list e1 e2 … en}` también en posición de patron, por ejemplo:

```
> (run '(match (list 2 (list 4 5) 6)
          ((case (list a (list b c) d) => c)))
5
```

5. (0.2) Finalmente, para dar una impresión más cómoda trabajando con listas, modifique el pretty-printer para que en el caso de listas, se use la notación `{list v1 … vn}`:

```
> (run '(list 1 4 6))
(list 1 4 6)
```

Evaluación Perezosa (3 ptos)

MiniScheme+ usa call-by-value como semántica de aplicación de funciones. Sin embargo, es posible agregar evaluación usando call-by-need para casos específicos.

1. (2.0) Agregue el keyword `lazy` para indicar que el argumento de una función debe ser evaluado usando call-by-need. Además, `lazy` puede ser usado en la declaración de estructuras para determinar la semántica de las funciones constructoras. Asegúrese de obtener semántica call-by-need, y no call-by-name! Ejemplos:

```
> (run '((fun (x (lazy y)) x) 1 (/ 1 0)))
1
> (run '((fun (x y) x) 1 (/ 1 0)))
"/: division by zero"
> (run '(local ((datatype T
                  (C (lazy a)))
                (define x (C (/ 1 0))))
          (T? x)))
#t
> (run '(local ((datatype T
                  (C (lazy a)))
                (define x (C (/ 1 0))))
          (match x
            ((case (C a) => a)))))
"/: division by zero"
```

2. En Haskell vimos que gracias a la evaluación perezosa es posible construir listas infinitas. Un stream es una estructura infinita compuesta por una cabeza `hd` y una cola `tl`, al igual que las listas. Un stream puede emular una lista infinita si se evita evaluar la cola del stream hasta que sea estrictamente necesario. Realice lo siguiente en MiniScheme+ extendido con el keyword `lazy`:
   - (0.5) Defina una estructura `Stream` que evite evaluar su cola a menos que sea estrictamente necesario.

     ```
     (def stream-data '(datatype Stream ...))
     ```

   - (0.5) Defina la función `(make-stream hd tl)` en MiniScheme+ que construye un stream basado en la estructura anterior. Ejemplo:

     ```
     (def make-stream '(define make-stream (fun ...)))
     ; Stream infinito de 1s
     (def ones '(define ones (make-stream 1 ones)))
     ```

Trabajando con Streams (1 ptos)

Nota: Todas las definiciones que se le piden a continuación deben realizarse en el lenguaje MiniScheme+ con las extensiones hasta este punto de la tarea.

Observe que para fines de presentación y de corrección, el intérprete define una conversión entre estructuras `List` de MiniScheme+ y listas de Racket.

1. (0.2) Defina las funciones `stream-hd` y `stream-tl` para obtener la cabeza y la cola de un stream. Por ejemplo:

```
(def stream-hd ...)
(def stream-tl ...)
> (run '(local (,stream-data ,make-stream ,stream-hd ,ones)
(stream-hd ones)))
1
> (run '(local (,stream-data ,make-stream
                             ,stream-hd ,stream-tl ,ones)
          (stream-hd (stream-tl ones))))
1
```

   Observe el uso de *quasi-quoting*[3] para definir individualmente las funciones pedidas, así como el stream ones. Sus respuestas deben definirse como fragmentos de programa que luego serán compuestos de la forma que aquí se ilustra.

2. (0.2) Implemente la función `(stream-take n stream)` que retorna una lista con los primeros n elementos de stream. Ejemplo:

```
(def stream-take ...)
> (run '(local ,stream-lib
          (local (,ones ,stream-take)
            (stream-take 10 ones))))
(list 1 1 1 1 1 1 1 1 1 1)
```

   Use la siguiente definición de `stream-lib` en los próximos ejercicios:

```
(def stream-lib (list stream-data
                      make-stream
                      stream-hd
                      stream-tl
                      stream-take))
```

   así no tendrá que volver a definir todas las funciones para cada ejercicio.

3. (0.2) Implemente la función `stream-zipWith` que funciona de manera análoga a `zipWith` para listas. Ejemplo:

```
(def stream-zipWith ...)

> (run '(local ,stream-lib
          (local (,ones ,stream-zipWith)
            (stream-take 10
                         (stream-zipWith
                           (fun (n m)
                             (+ n m))
                           ones
                           ones)))))
(list 2 2 2 2 2 2 2 2 2 2)
```

   .

4. (0.2) Implemente el stream fibs, de todos los números de Fibonacci.

```
(def fibs ...)

> (run '(local ,stream-lib
          (local (,stream-zipWith ,fibs)
            (stream-take 10 fibs))))
(list 1 1 2 3 5 8 13 21 34 55)
```

   .

5. (0.2) Implemente el stream merge-sort, que dados dos Stream ordenados retorna un Stream con la mezcla ordenada

```
(def merge-sort ...)

> (run '(local ,stream-lib
          (local (,stream-take ,merge-sort ,fibs ,stream-zipWith)
            (stream-take 10 (merge-sort fibs fibs)))))
(list 1 1 1 1 2 2 3 3 5 5)
```

[1] MiniScheme+ usa un ambiente mutable que permite definir funciones recursivas con `define`.

[2] Para entender el concepto de funciones primitivas, vea http://pleiad.cl/teaching/primitivas.

[3] Quasi-quoting es una forma de evaluar partes de una expresión quote-eada. Por ejemplo:

```
> '(1 (+ 1 2))
(list 1 (list '+ 1 2))
> `(1 ,(+ 1 2))
(list 1 3)
> `(1 ,(+ 1 2))
(list 1 (list 'unquote (list '+ 1 2)))))
```