Tarea 2 (Entrega: 22 de junio de 2020)

Esta tarea se distribuye con un archivo zip (baseT2.zip) que contiene 2 archivos: main.rkt y tests.rkt. Los archivos main.rkt y tests.rkt están incompletos, y en ellos tienen que implementar lo que se solicita en las preguntas siguientes.

Deben entregar via U-cursos **un archivo .zip** que contenga los archivos main.rkt y tests.rkt.

Consulte las normas de entrega de tareas en http://pleiad.cl/teaching/cc4101 (http://pleiad.cl/teaching/cc4101)

¡Los tests, los contratos y las descripciones forman parte de su evaluación!

Resumen

El objetivo de esta tarea es estudiar como proveer estructuras infinitas como streams en un lenguaje que combine evaluación temprana y evaluación perozosa (call-by-need).

La tarea está dividida en 5 secciones, las cuales se describen a continuación:

- **MiniScheme+**: En esta sección se describe el lenguaje MiniScheme+, el cual posee estructuras de datos (tipos de datos algebraícos) y pattern matching.

- **Warm-up**: En esta sección, luego de haberse familiarizado con el lenguaje MiniScheme+, se le pide brindar un string de salida más amigable para las estructuras.

- **Listas**: El objetivo de esta sección es extender el lenguaje MiniScheme+ con listas y sus funcionalidades.

- **Evaluación perezosa**: En esta sección vamos a agregar al lenguaje MiniScheme+ evaluación call-by-need para casos específicos. Esta extensión permite la definición de streams (estructuras de datos infinitas, similares a las listas infinitas de Haskell).

- **Trabajando con Streams**: Esta sección está dedicada a definir diferentes streams y funciones sobre estas estructuras.

MiniScheme+

En esta tarea se le provee el lenguaje MiniScheme+, que corresponde a un intérprete extendido con estructuras de primera clase y pattern matching. Además, MiniScheme+ incorpora el uso de primitivas[1)]. 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 (incluso recursivas) 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 {Succ {Zero}}}}}})
"{Succ {Zero}}"
> (run '{local {{datatype Nat
                  {Zero}
                  {Succ n}}
                {define twice {fun {n}
                                  {match n
                                    {case {Zero} => {Zero}}
                                    {case {Succ m} => {Succ {Succ {twice m}}}}}}}}
          {twice {Succ {Succ {Zero}}}}})
"{Succ {Succ {Succ {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`, permitiendo el uso de funciones constructoras y además, predicados para determinar si un valor corresponde a la estructura (para el tipo en general y para cada constructor). Para más detalles, consulte la implementación y tests provistos.

Warm-up (0.5 ptos)

- (0.4 ptos) Si ejecutan el penúltimo ejemplo, verán que el output no es

```
"{Succ {Zero}}"
```

sino `(structV 'Nat 'Succ (list (structV 'Nat 'Zero empty)))`. Defina la función `pretty-printing` que toma una estructura y entrega un string más amigable al usuario (similar a como se indica anteriormente).

```
> (pretty-printing (structV 'Nat 'Succ (list (structV 'Nat 'Zero empty))))
"{Succ {Zero}}"
```

- (0.1 ptos) Utilice la función `pretty-printing` en la función `run` para obtener un output más amigable.

```
(run '{local {{datatype Nat
                  {Zero}
                  {Succ n}}
                {define pred {fun {n}
                                  {match n
                                    {case {Zero} => {Zero}}
                                    {case {Succ m} => m}}}}}
          {pred {Succ {Succ {Succ {Zero}}}}}})
"{Succ {Succ {Zero}}}"
```

Listas (1.5 ptos)

1. (0.3) 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. Con estas definiciones, modifique la función `run` para que evalúe la expresión dada en un contexto donde se tiene la definición de `List` y de `length`. Se espera el siguiente comportamiento:

   ```
   > (run '{Empty? {Empty}})
   #t
   > (run '{List? {Cons 1 2}})
   #t
   >(run '{length {Cons 1 {Cons 2 {Cons 3 {Empty}}}}})
   3
   ```

2. (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

   >(run '{match {list}
             {case {Cons h r} => h}
             {case _ => 0}})
   0
   ```

   No necesita modificar ni el AST ni el intérprete.

3. (0.5) Extienda ahora el pattern matching para que la notación `'{list e1 e2 … en}` se pueda usar también en posición de patrón.

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

4. (0.2) Finalmente, modifique el `pretty-printing` para que en el caso de listas, se use la notación `'{list v1 … vn}`.

   ```
   > (run '{list 1 4 6})
   "{list 1 4 6}"

   >(run '{list})
   "{list}"
   ```

Evaluación Perezosa (3 ptos)

MiniScheme+ usa call-by-value como estrategia de evaluación. Sin embargo, es posible agregar evaluación 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 las funciones constructoras. **¡Asegúrese de obtener semántica call-by-need, y no call-by-name!**

```
> (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"
> (run '{local {{datatype T {C a {lazy b}}}
                {define x {C  0 {+ 1 2}}}}
               x})
"{C 0 3}"
```

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 en MiniScheme+ 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.

     ```
     (def make-stream '{define make-stream  {fun ...}})
     ```

     Con la función `(make-stream hd tl)`, podemos definir un stream infinito de 1s.

     ```
     ; 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*[2] 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}"
```

Para no tener que volver a definir todas las funciones para cada ejercicio, utilice la siguiente definición de `stream-lib`:

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

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

```
(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 la función `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}"
```

Asuma que ambas listas se encuentran ya ordenadas.

---

1)
Para entender el concepto de funciones primitivas, vea http://pleiad.cl/teaching/primitivas (http://pleiad.cl/teaching/primitivas) .
2)
Quasi-quoting es una forma de evaluar partes de una expresión quote-eada.

```
> '(1 (+ 1 2))
'(1 (+ 1 2))

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

> '(1 ,(+ 1 2))
'(1 ,(+ 1 2))
```