EVALUACIÓN EN ORDEN NORMAL

La evaluación puede definirse utilizando semántica natural o big-step. En este tipo de semántica, uno no describe un paso de ejecución, sino directamente una relación entre los términos y sus valores (que también son términos, son formas canónicas). Llamaremos => a esta relación.

Una regla para formas canónicas:

```
------------
λv.e => λv.e
```

Una regla para la aplicación:

```
e => λv.e"    (e"/v->e') => z
-----------------------------
         e e' => z
```

Cuando afirmamos que e => z se cumple, estamos diciendo que existe un árbol de derivación que demuestra e => z. Estos árboles usualmente son difíciles de graficar, por el tamaño de las expresiones intervinientes. Se suele usar indentación para expresar la estructura. Por ejemplo, la evaluación normal de (λx.λy.xx) Δ se escribe:

```
(λx.λy.xx) Δ           (es una aplicación, por lo tanto usamos la 2da regla, a continuación siguen los 2 hijos)
  λx.λy.xx => λx.λy.xx  (primer hijo, es una abstracción, por lo tanto usamos la 1er regla, no hay hijos)
  λy.Δ Δ => λy.Δ Δ      (segundo hijo, también abstracción, no hay hijos)
=> λy.Δ Δ               (terminó la prueba)
```

Otro ejemplo es la evaluación normal de (λx.x (λy.x y y) x) (λz.λw.z):

```
(λx.x (λy.x y y) x) (λz.λw.z)                    (aplicación, siguen 2 hijos: a y b)
  λx.x (λy.x y y) x => λx.x (λy.x y y) x         (a: abstracción, no hay hijos)
  (λz.λw.z) (λy.(λz.λw.z) y y) (λz.λw.z)         (b: aplicación, siguen 2 hijos: ba y bb)
    (λz.λw.z) (λy.(λz.λw.z) y y)                 (ba: aplicación, siguen 2 hijos: baa y bab)
      λz.λw.z => λz.λw.z                         (baa: abstracción, no hay hijos)
      λw.λy.(λz.λw.z) y y => λw.λy.(λz.λw.z) y y  (bab: abstracción, no hay hijos)
    => λw.λy.(λz.λw.z) y y                       (terminó ba)
    λy.(λz.λw.z) y y => λy.(λz.λw.z) y y         (bb: abstracción, no hay hijos)
  => λy.(λz.λw.z) y y                            (terminó b)
=> λy.(λz.λw.z) y y                              (terminó la prueba)
```

La notación con indentación sola es difícil de leer, se mejora si se escriben corchetes "[" a la izquierda encerrando los subárboles. En el caso de arriba habría un corchete que abarque las 10 líneas, otro que abarque sólo la línea 2, otro de la 3 a la 9, otro de la 4 a la 7, otro para la 8 sola, otro para la 5 sola y otro para la 6 sola. Los corchetes más grandes van más a la izquierda. Es difícil de hacer en un archivo de texto como éste.

Hay expresiones, como Δ Δ cuya evaluación no termina:

```
(λx.xx) (λx.xx)                                  (aplicación, siguen 2 hijos)
  (λx.xx) => (λx.xx)                             (abstracción, no hay hijos)
  (λx.xx) (λx.xx)                                (aplicación, siguen 2 hijos)
    (λx.xx) => (λx.xx)                           (abstracción, no hay hijos)
    (λx.xx) (λx.xx)                              (aplicación, siguen 2 hijos)
      (λx.xx) => (λx.xx)                         (abstracción, no hay hijos)
      (λx.xx) (λx.xx)                            (aplicación, siguen 2 hijos)
...
```

no podemos construir el árbol porque la evaluación no termina.

En efecto, las reglas que definen => dan lugar al siguiente algoritmo:

```
evalN :: Exp -> Exp
evalN (Lam v e) = (Lam v e)
evalN (App e e') = case evalN e of
                     Lam v e" -> evalN (e"/v->e')
```

que, por supuesto, requiere definir la sustitución para que sea un programa ejecutable. Los árboles mostrados más arriba pueden interpretarse como que la indentación corresponde a las llamadas recursivas.

La aplicación de evalN a Δ Δ no termina.

EVALUACION EAGER

La evaluación en orden normal a partir de Δ ((λx.x) (λy.y)) contrae dos veces el redex (λx.x) (λy.y).

```
(λx.xx) ((λx.x) (λy.y))                 (aplicación, siguen 2 hijos: a y b)
  (λx.xx) => (λx.xx)                    (a: abstracción, no hay hijos)
  ((λx.x) (λy.y)) ((λx.x) (λy.y))       (b: aplicación, siguen 2 hijos: ba y bb)
```

```
    (λx.x) (λy.y)                       (ba: aplicación, siguen 2 hijos: baa y bab)
      (λx.x) => (λx.x)                  (baa: abstracción, no hay hijos)
      (λy.y) => (λy.y)                  (bab: abstracción, no hay hijos)
    => (λy.y)                           (terminó ba)
    (λx.x) (λy.y)                       (bb: aplicación, siguen 2 hijos: bba y bbb)
      (λx.x) => (λx.x)                  (bba: abstracción, no hay hijos)
      (λy.y) => (λy.y)                  (bbb: abstracción, no hay hijos)
    => (λy.y)                           (terminó bb)
  => (λy.y)                             (terminó b)
=> (λy.y)                               (terminó la prueba)
```

la repetición de la contracción se observa al ver que las líneas 4 a 7 y 8 a 11 son idénticas. Esto se podría evitar si se cambia el orden de evaluación, reduciendo el operando antes de sustituir en la regla de evaluación. La definición de evaluación eager, entonces, está dada por las reglas:

Una regla para formas canónicas:

```
------------
λv.e => λv.e
```

Una regla para la aplicación:

```
e => λv.e"    e' => z'    (e"/v->z') => z
-----------------------------------------------
              e e' => z
```

Retomando el ejemplo anterior, con la evaluación eager tenemos:

```
(λx.xx) ((λx.x) (λy.y))                 (aplicación, siguen 3 hijos: a, b y c)
  (λx.xx) => (λx.xx)                    (a: abstracción, no hay hijos)
  (λx.x) (λy.y)                         (b: aplicación, siguen 3 hijos: ba, bb y bc)
    (λx.x) => (λx.x)                    (ba: abstracción, no hay hijos)
    (λy.y) => (λy.y)                    (bb: abstracción, no hay hijos)
    (λy.y) => (λy.y)                    (bc: abstracción, no hay hijos)
  => (λy.y)                             (terminó b)
  (λy.y) (λy.y)                         (c: aplicación, siguen 3 hijos: ca, cb y cc)
    (λy.y) => (λy.y)                    (ca: abstracción, no hay hijos)
    (λy.y) => (λy.y)                    (cb: abstracción, no hay hijos)
    (λy.y) => (λy.y)                    (cc: abstracción, no hay hijos)
  => (λy.y)                             (terminó c)
=> (λy.y)                               (terminó la prueba)
```

También da lugar a un algoritmo de evaluación:

```
evalE :: Exp -> Exp
evalE (Lam v e) = (Lam v e)
evalE (App e e') = case evalE e of
                     Lam v e" -> case evalE e' of
                                   Lam w e0 -> evalE (e"/v->Lam w e0)
```

Así como hemos llamado de manera diferente a los programas (evalN y evalE) correspondientes a la evaluación normal e eager, escribiremos =N=> o =E=> para explicitar que nos referimos a la evaluación normal o eager.

Usualmente evaluación eager es más rápida que la evaluación normal porque es común que en la normal se repitan contracciones que en la eager se hacen una sola vez. Puede ocurrir lo contrario: la eager puede evaluar un argumento que no se utiliza. En ese caso, la eager es menos eficiente. Puede incluso ocurrir que la eager no termine en casos en los que la normal sí lo haga:

```
(λx.λy.y) (Δ Δ) =N=> λy.y
```

mientras que en la evaluación eager, (λx.λy.y) (Δ Δ) diverge.

Regla η

Un η-redex es una expresión de la forma λv.ev donde v $\notin$ FV(e). Gracias a la regla β , uno obtiene que (λ v.ev) e' contrae a e e' para toda expresión e'. Si uno asume que toda expresión lambda denota una función, λ v.ev y e parecen comportarse extensionalmente igual: cuando se las aplica a e', ambas dan e e'. Esto motiva la

regla η :

```
------------- si v ∉ FV(e)
(λ v.e v) -> e
```

Esta regla no va a ser tan estudiada como la β ya que no preserva significado en algunos lenguajes que veremos más adelante, en los que hay expresiones que no denotan funciones.