Lenguajes de programación 2016-2
Ejercicio Semanal 3

Noé Salomón Hernández Sánchez
Albert M. Orozco Camacho
C. Moisés Vázquez Reyes

Facultad de Ciencias UNAM
Se entrega el 8 de abril del 2016

Los términos del cálculo-lambda puro se generan con la siguiente gramática:

$$e \ := \ x \mid \lambda x.e \mid e \ e$$

su respectiva definición en Haskell es la siguiente:

```
data LamU = Var String | Lam String LamU | App LamU LamU
```

El objetivo de esta práctica es abarcar conceptos del cálculo-lambda puro como la β-reducción, la forma normal de un término y la recursión utilizando operadores de punto fijo. La estrategia de evaluación que se utilizará en la implementación de este ejercicio es reducción β-completa.

1. Ejercicios:

Utilizaremos el tipo:

```
type Sust = (String,LamU)
```

para representar una sustitución del estilo $[x := t]$.

- `fv::LamU->[String]`

  Calcula las variables libres de un término del cálculo-lambda puro.

  ```
  >fv $ Lam ‘‘x’’ $ Lam ‘‘y’’ $ App (App (Var ‘‘x’’) (Var ‘‘w’’)) (Var ‘‘z’’)
  [ ‘‘w’’,‘‘z’’ ]
  ```

- `sust::LamU->Sust->LamU`

  Realiza una sustitución en un término del cálculo-lambda puro.

  ```
  >sust (Lam ‘‘x’’ $ Lam ‘‘z’’ $ App (App (Var ‘‘x’’)
                        (Var ‘‘y’’)) (Var ‘‘z’’)) (‘‘y’’, Var ‘‘r’’)
  /x./z.(xr)z
  ```

- `hayRedex::LamU->Bool`

  Nos dice si una expresión contiene un redex o no. Recordemos que un redex es una aplicación de la forma $(\lambda x.e)\ t$.

  ```
  >hayRedex $ App (Var ‘‘x’’) (Var ‘‘z’’)
  False
  >hayRedex $ App (Var ‘‘x’’) (App (Lam ‘‘z’’ $ Var ‘‘z’’) (Var ‘‘y’’))
  True
  ```

- `betaR::LamU->LamU`

  Realiza una β-reducción en un término y sólo una. La estrategia de evaluación es la siguiente:

  $$(\lambda x.e)\ t \rightarrow_\beta e[x := t]$$

  ```
  >betaR $ App (Var ‘‘x’’) (App (Lam ‘‘z’’ $ Var ‘‘z’’) (Var ‘‘y’’))
  xy
  ```

- `fn::LamU->LamU`

  Reduce un término hasta su forma normal.

  ```
  >fn $ App (Lam ‘‘x’’ $ Var ‘‘z’’) (App (Lam ‘‘z’’ $ Var ‘‘z’’)
  (Var ‘‘y’’))
  z
  ```

- `church::Int->LamU`

  Dado un número entero positivo nos devuelve su representación como numeral de Church.

  ```
  >church 7
  /s./z.s(s(s(s(s(s(sz))))))
  ```

Lo siguiente es declarar constantes que representen algunos operadores del cálculo-lambda puro. Por ejemplo, la constante *true* puede ser representada de la siguiente manera:

```
true = Lam ‘‘x’’ $ Lam ‘‘y’’ $ Var ‘‘x’’
```

deben dar representaciones para *false, if-then-elese, iszero, pair, fst, snd, suc, pred, suma, prod.*

Utilizaremos el operador de punto fijo de Curry-Roser para hacer recursión, su representación en Haskell es la siguiente:

```
pF = Lam ‘‘f’’ $ App (Lam ‘‘x’’ $ App (Var ‘‘f’’) (App (Var ‘‘x’’) (Var ‘‘x’’)))
                     (Lam ‘‘x’’ $ App (Var ‘‘f’’) (App (Var ‘‘x’’) (Var ‘‘x’’)))
```

Para finalizar, da una definición de la función *factorial* para numerales de Church. Dentro del archivo ya se encuentra lo siguiente:

```
fac = App pF g where g = error ‘‘Te toca’’
```

Sólo deben de dar la definición de g.

EXTRA (+4s pts): Traduce el operador de punto fijo de Klop a su respectiva definición en Haskell y calcula $fac\ \overline{3}$.