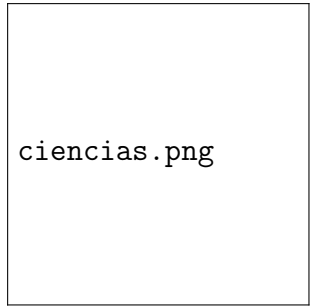


unam.jpg



ciencias.png

Universidad Nacional Autónoma de  
México

Facultad de Ciencias

Lenguajes de programación

Alumnos:

Guerrero Chávez Diana Lucía

Lázaro Arias Jorge Alberto

Sánchez Alcántara Jesús Esteban

# Tarea 2

11 de octubre de 2015

## Problema I

En teoría y laboratorio hemos visto el lenguaje FAE, que es un lenguaje con expresiones aritméticas, funciones y aplicaciones de funciones. ¿Es FAE un lenguaje Turing-Completo?. Debes proveer una respuesta breve e inambigua, seguida de una justificación más extensa de tu respuesta. **Hint:** Investiguen sobre el combinador Y

FAE es Turing-Completo, debido a que se tienen condicionales y recursión, viendo que de acuerdo a como fue definido pertenece al paradigma funcional, aparte de que implementamos puntos estrictos (Combinador Y) y además nuestro lenguaje está dividido en tres primitivas:

- Procedimientos de definición
- Procedimientos de aplicación
- Variables

Podemos calcular el punto estricto del argumento de un procedimiento, lo cual es la definición del calculo lambda.

## Problema II

¿Java es glotón o perezoso? Escribe un programa para determinar la respuesta a esta pregunta. El mismo programa, ejecutado en cada uno de los dos regímenes, debe producir resultados distintos. Puedes usar todas las características de Java que gustes, pero debes mantener el programa relativamente corto; penalizaremos cualquier programa que consideremos excesivamente largo o confuso (Hint: es posible resolver este problema con un programa de unas cuantas docenas de líneas).

Debes anexar tanto el código fuente de tu programa (en un archivo aparte al PDF de la tarea) así como una respuesta a la pregunta de si Java es glotón o perezoso, y una explicación de porque su programa determina esto. Es decir, deben proveer una respuesta breve e inambigua (p.ej. "Java es perezoso") seguida de una descripción del resultado que obtendrías bajo cada régimen, junto con una breve

explicación de por que ese régimen generaría tal resultado.

Java es un lenguaje glotón. En el programa adjunto se realiza una división entre cero. En el programa, podemos ver que cuando Java realiza evaluación glotona, hace una división entre el factorial de 5 y cero (120/0). El programa devuelve un error, ya que primero resuelve la llamada a la función factorial(5) y después resuelve la división.

Si Java realizara evaluación perezosa, primero, intentaría realizar la división, y lanzaría un error, ya que es una división entre cero, es decir, primero resolvería la división, y luego la llamada a factorial(5).

## Problema III

En nuestro intérprete perezoso, identificamos 3 puntos en el lenguaje donde necesitamos forzar la evaluación de las expresiones closures (invocando a la función `strict`): la posición de la función de una aplicación, la expresión de prueba de una condicional, y las primitivas aritméticas. Doug Oord, un estudiante algo sedentario, sugiere que podemos reducir la cantidad de código reemplazando todas las invocaciones de `strict` por una sola. En el interprete visto en el capítulo 8 del libro de Shriram elimino todas las instancias de `strict` y reemplazo

```
[id (v) (lookup v env)]  
por  
[id (v) (strict (lookup v env))]
```

El razonamiento de Doug es que el único momento en que el interprete regresa una expresión closure es cuando busca un identificador en el ambiente. Si forzamos esta evaluación, podemos estar seguros de que en ninguna otra parte del interprete tendremos un closure de expresiones, y eliminando las otras invocaciones de `strict` no causaremos ningún daño. Doug evita razonar en la otra dirección, es decir si esto resultara o no en un interprete mas glotón de lo necesario.

Escribe un programa que produzca diferentes resultados sobre el interprete original y el de Doug. Escribe el resultado bajo cada interprete e identifica claramente cual interprete producirá cada resultado. Asume un lenguaje interpretado con características aritméticas, funciones de primera clase, `with`, `if0` y `rec` (aunque algunas no se encuentren en nuestro interprete perezoso). Hint: Compara este comportamiento contra el interprete perezoso que vimos en clase y no contra el comportamiento de Haskell.

Si no puedes encontrar un programa como el que se pide, defiende tus razones de por que no puede existir, luego considera el mismo lenguaje con `cons`, `first` y `rest` añadidos.

Hay diferencia entre el interprete original y el interprete propuesto por Doug, solo hay que fijarse en los test que regresan los closures, ya que estos son los puntos alterados que hizo Doug, haciendo que se obtengan resultados disntintos al interprete original.

## Problema IV

Ningún lenguaje perezoso en la historia ha tenido operaciones de estado (tales como la mutación de valores en cajas o asignación de valores a variables) ¿Por que no?

La mejor respuesta a esta pregunta incluiría dos cosas: un pequeño programa (que asume la evaluación perezosa) el cual usara estado y una breve explicación de cual es el problema que ilustra la ejecución de dicho programa. Por favor usa la noción original (sin cache) de perezosos sin cambio alguno. Si presentas un ejemplo lo suficientemente ilustrativo (el cual no necesita ser muy largo), tu explicación será muy pequeña.

Tenemos por ejemplo, que la introducción de la mutación nos obliga a diferenciar entre igualdad de valor e igualdad de contenido:

- La igualdad de referencia se comprobaría con una función `eq?`: (`eq? x y`) devuelve `#t` cuando `x` e `y` apuntan al mismo dato
- La igualdad de contenido se comprobaría con la función `equal?`: (`equal? x y`) devuelve `#t` cuando `x` e `y` contienen el mismo valor

Así, si dos variables son `eq?` también son `equal?`, teniendo:

```
(define a (cons 1 2))
(define b (cons 1 2))
(define c a)
(equal? a b)
(equal? a c)
(eq? a b)
(eq? a c)
```