Tarea 2

Herrera Brito Juan José

Bravo García Marco Antonio

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.

Como FAE esta implementado sobre un Paradigma Funcional, podemos decir que es Turing-Completo si podemos hacer recursion y tener condicionales.

Para la recursion basta con el concepto del Combinador Y, podemos definir una funcion recursiva, con una funcion normal de tal manera que

Y = \f. f (Y f).

Por ejemplo:

(Y (\fact.\x. if (< x 1) 1 (* x (fact (- x 1))))) 7

Esta expresión, calcula el factorial de 7.

Para las condicionales, como estamos sobre el nucleo de Scheme, existen.

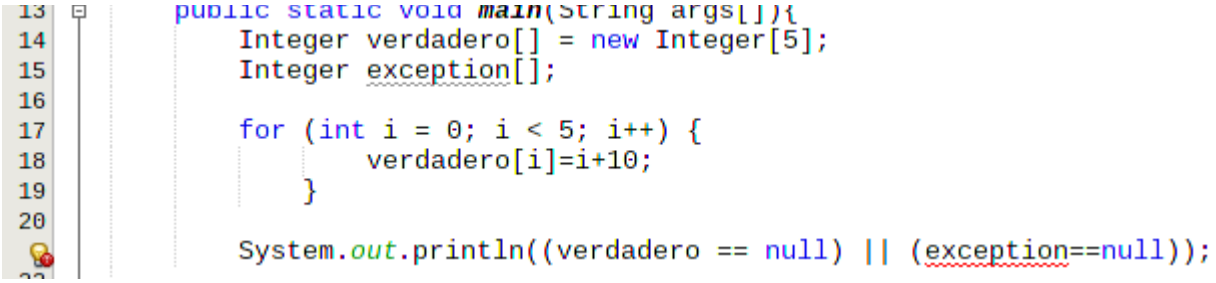
Por tanto FAE es Turing-Completo

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 lineas).

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 Glotón.



```
13      public static void main(String args[]){
14          Integer verdadero[] = new Integer[5];
15          Integer exception[];
16
17          for (int i = 0; i < 5; i++) {
18                  verdadero[i]=i+10;
19              }
20
21          System.out.println((verdadero == null) || (exception==null));
```

Si Java fuera Perezoso:
En la linea 21, imprime True, ya que como sabemos basta con que un lado de un OR sea verdadero para que todo lo sea, como suponemos que es perezoso, solo evualuaria primero el lado izquierdo, el cual regresa un True y con eso basta para que todo lo sea

Pero es Glotón:
En la linea 21, al momento de querer comparar si el arreglo "exception" es vacio, lanza una excepcion ya que no esta inicializado, pero el lado izquierdo regresa un True, esto deberia bastar segun la logica basica, al ser Glotón debe evaluar antes de mandarlo a pantalla, y lanza la excepcion.

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 capitulo 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.

En el interprete del libro en el capitulo 8 al tener un evaluación de funciones se tiene un exprV ya que estas son parte de los "resultados" de nuestro lenguaje perezoso.
Por ejemplo ejecutando la siguiente linea:
(rinterp (cparse '{{fun {x} {{fun {y} y} x}} 5}))

El resultado es
(exprV (id 'x) (aSub 'x (exprV (num 5) (mtSub)) (mtSub)))

Y en el lenguaje de nuestro amigo Doug al ejecutar el mismo código obtenemos:
(numV 5)

Se puede notar que el lenguaje de Shriram trata de hacer evidente lo perezoso del lenguaje, en cambio, el de Doug, interpreta y se obtiene un resultado inmediato.

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 sera muy pequeña.

Al tener operaciones de estado y modificando una variable que ya esta siendo usada con un valor , y darle otro, se podrían crear graves problemas de inconsistencia pues en la misma ejecución cambiaría su valor creando una cantidad inmensa de errores de inconsistencia en el programa, ya que este seria muy frecuente, un ejemplo para ilustrar esto, se muestra en el siguiente código, que para fines del ejemplo se manejara una sintaxis parecida a la del curso, pero no es ejecutable en ninguna de las gramáticas vistas hasta el momento:

```
{+ {with (x 10) x}
   (newState x 5)
   {+ 5 x}}
```

El código normalmente devolvería 20 pues el with evaluaría su x = 10, la segunda x por el (newState x 5) tendría un 5 y sumarían (+ 10 (+5 5)).

Ahora, suponiendo que se tiene un código "valido" como el siguiente, primero el with no evaluaría x como 10 al ser un lenguaje perezoso, quedaría a la espera de ser necesitada la suma

```
{with (x 10) x}
```

Suponiendo que fuera valida la sintaxis queremos darle el valor de 5 a la segunda x, pero al cambiar estados y sin lograr ser evaluada con 10 la primera x, se cambiaría a 5, al final se tendría

(+ 5 (+ 5 5)) = 15

Este resultado claramente no es el esperado. Si bien la sintaxis usada para el ejemplo es un poco absurda, el ejemplo ilustra claramente la cantidad de errores que se tendrían en un lenguaje perezoso con operaciones de estado.