Segundo Examen Parcial

Materia: Programación Avanzada
Profesor: Gustavo Márquez Flores
Estudiantes: Rodrigo Sebastián Cortez Madrigal y Rubén Romero Flores

Primera parte: Haskell

I. Considerando las siguientes definiciones de funciones:

Encuentra el valor de las siguientes expresiones

```haskell
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' _ e [] = e
foldr' g e (x:xs) = g x (foldr' g e xs)

foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' _ e [] = e
foldl' g e (x:xs) = foldl' g (g e x) xs
```

```haskell
let lista = [81, 27, 9, 3]

-- a) foldl (-) 1 [81, 27, 9, 3]
print $ foldl' (-) 1 lista

-- b) foldr (-) 1 [81, 27, 9, 3]
print $ foldr' (-) 1 lista

-- c) foldr (/) 1 [81, 27, 9, 3]
print $ foldr' (/) 1 lista

-- d) foldl (/) 81 [9, 3, 1]
print $ foldl' (/) 81 [9, 3, 1]
```

Estas funciones son para “reducir” una lista desde la derecha (foldr) o desde la izquierda (foldl).

- **a) foldl (-) 1** $[81, 27, 9, 3]$
  - Inicialmente e = 1 y aplicamos foldl de izquierda a derecha:
  1. (1 - 81) = -80

2. (-80 - 27) = -107
3. (-107 - 9) = -116
4. (-116 - 3) = -119

- **Resultado:** -119
- **b) foldr (-) 1** $[81, 27, 9, 3]$
  - Inicialmente, e = 1, y aplicamos foldr de derecha a izquierda:
  1. (3 - 1) = 2
  2. (9 - 2) = 7
  3. (27 - 7) = 20
  4. (81 - 20) = 61
- **Resultado:** 61
- **c) foldr (/) 1** $[81, 27, 9, 3]$
  - Inicialmente, e = 1, y aplicamos foldr de derecha a izquierda:
  1. (3 / 1) = 3
  2. (9 / 3) = 3
  3. (27 / 3) = 9
  4. (81 / 9) = 9

  Resultado: 9
- d) foldl (/) 81 $[9, 3, 3, 1]$:
  - Inicial: e = 81 y aplicamos foldr de izquierda a derecha
  1. 81 / 9 = 9
  2. 9 / 3 = 3
  3. 3 / 3 = 1
  4. 1 / 1 = 1

  Resultado: 1.0

II. Definir la función todosImpares :: $[Int]$ -> $Bool$

Que verifica si todos los elementos de una lista xs son impares.

Ejemplo:

- todosImpares [1, 3, 5] == True
- todosImpares [1, 3, 5, 6] == False

```haskell
todosImpares :: [Int] -> Bool
todosImpares xs = all odd xs

todosImpares2 :: [Int] -> Bool
todosImpares2 xs = foldr' (\x y -> odd x && y) True xs
```

III. Explica el significado de las siguientes expresiones

A) Función que implementa la primera expresión: Eq a => a -> a -> Bool

Compara si dos valores son iguales

1. Es el tipo de una función que toma **dos valores** del mismo tipo a y regresa un valor de tipo Bool.
2. La restricción Eq a significa que el tipo a debe ser una instancia de la clase de tipos Eq, es decir, debe soportar operaciones de comparación de igualdad $(==)$ y desigualdad $(/=)$.

Ejemplo

```
esIgual :: Eq a => a -> a -> Bool
esIgual x y = x == y
```

B) Función que implementa la segunda expresión: Eq a => a -> $[a]$ -> Bool

Verifica si un elemento está en una lista

1. Es el tipo de una función que toma **un valor** de tipo a y una **lista de valores** de tipo a y regresa un valor de tipo Bool.
2. La restricción Eq a significa que el tipo a debe ser una instancia de Eq, permitiendo comparar el valor con los elementos de la lista.

Ejemplo

```
esElemento :: Eq a => a -> [a] -> Bool
esElemento x xs = x `elem` xs
```

IV. La función concat tiene la siguiente definición:

Redefine esta función usando plegado por la derecha.

La función concat toma una lista de listas y las combina en una sola lista.

```
concat [] = []
concat (xs:xss) = xs ++ concat xss
```

Podemos reescribir esta función utilizando un **plegado por la derecha** (foldr), que es una forma más idiomática en Haskell para combinar elementos recursivamente.

1. foldr: Recorre la lista de listas desde la derecha hacia la izquierda.
2. Operador (++): Combina cada lista con el acumulador.
3. Caso base ([]): Cuando la lista de listas está vacía, el resultado es una lista vacía ([]).

```
concatFoldr :: [[a]] -> [a]
concatFoldr = foldr (++) []
```

Para $[[1,2,3],[4,5,6]]$, el foldr opera de la siguiente manera:

1. Inicialmente: foldr (++) [] $[[1,2,3],[4,5,6]]$
2. Combina el último elemento $[4,5,6]$ con el acumulador []:

$[4,5,6]$ ++ [] = $[4,5,6]$

3. Combina el penúltimo elemento $[1,2,3]$ con el resultado previo:

$[1,2,3]$ ++ $[4,5,6]$ = $[1,2,3,4,5,6]$

Resultado: $[1,2,3,4,5,6]$

V. Expresa en cálculo lambda la composición de funciones $f(g(x))$.

$\lambda x.\, f(g\, x)$

Explicación:

1. λx: Representa una función anónima que toma un argumento x.
2. $g\, x$: Aplica la función g al argumento x.
3. $f(g\, x)$: Aplica la función f al resultado de $g(x)$.

Generalización:

La composición de funciones, denotada como $(f \circ g)(x) = f(g(x))$, puede expresarse en cálculo lambda como:

$\lambda f g x.\, f(g\, x)$

Este último representa una función que toma f, g, y luego x, y devuelve $f(g(x))$.

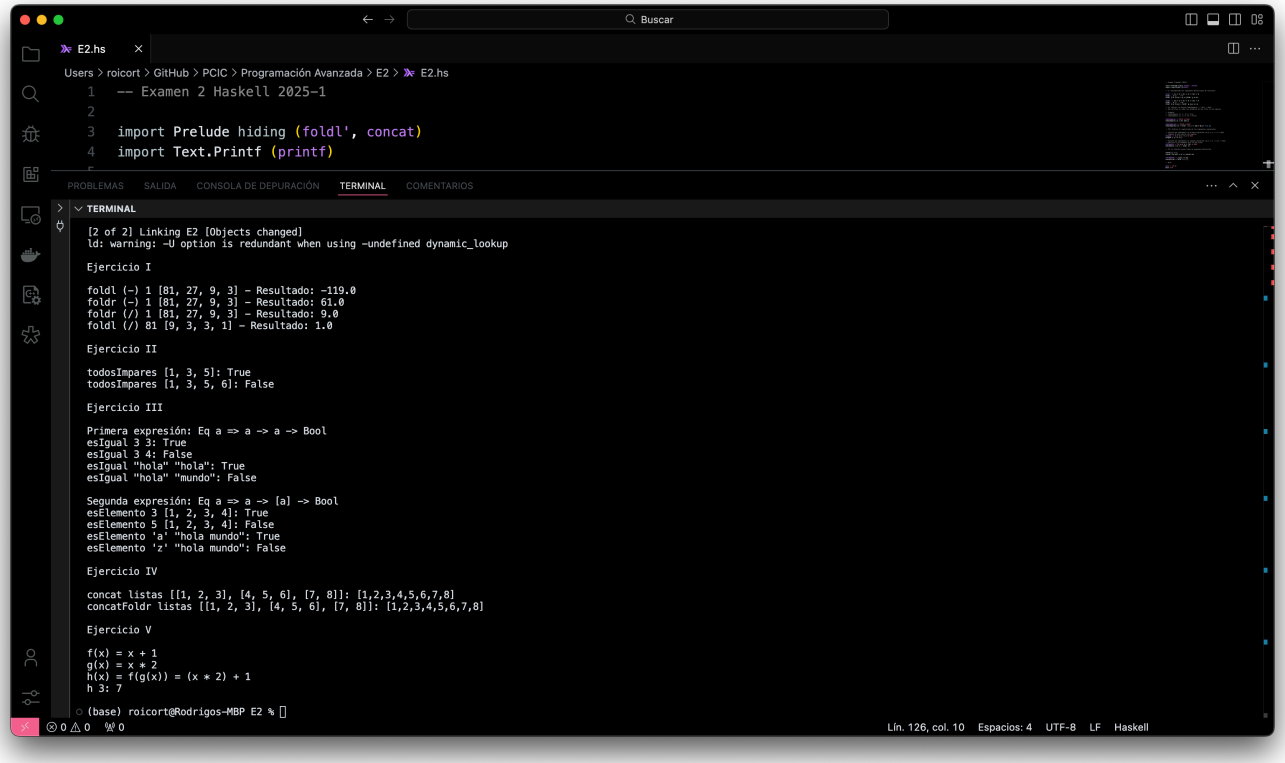
```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Ejemplo

```haskell
f = (+1)
g = (*2)

h = f . g  -- h(x) = f(g(x)) = (x * 2) + 1
```

Ejecución



Segunda parte: Prolog

VI. ¿Qué es una cláusula de *Horn*?

Cláusula que contiene a lo sumo una literal no negada. Su forma general es la siguiente:

$$\forall X1, X2, \ldots, XnL1 \vee \neg L2 \vee \ldots \vee \neg Lm$$

Las cláusulas de Horn se pueden reescribir de la siguiente manera:

$$\forall X1, X2, \ldots, XnL1 \vee \neg(L2 \wedge \ldots \wedge Lm)$$

$$\forall X1, X2, \ldots, XnL1 \leftarrow L2 \wedge \ldots \wedge Lm$$

Son fundamentales en sistemas de inferencia, como Prolog, por su simplicidad y eficiencia computacional. Estas cláusulas permiten representar hechos y reglas de manera compacta, facilitando la resolución de problemas lógicos y la deducción en bases de conocimiento.

VII. Escribe un programa en *Prolog* que muestre sus capacidades de aprendizaje.

¿Cómo agregary eliminar hechos y reglas a su base de Conocimiento?

```
% Declaramos que los predicados pueden ser modificados en tiempo de
ejecución.

:- dynamic hecho/1.
:- dynamic regla/2.

% Predicado para agregar un hecho

agregar_hecho(Hecho) :-
assertz(Hecho),
assertz(hecho(Hecho)).

% Predicado para eliminar un hecho

eliminar_hecho(Hecho) :-
retract(Hecho),
retract(hecho(Hecho)).

% Predicado para agregar una regla

agregar_regla(Cabeza, Cuerpo) :-
assertz((Cabeza :- Cuerpo)),
assertz(regla(Cabeza, Cuerpo)).

% Predicado para eliminar una regla

eliminar_regla(Cabeza, Cuerpo) :-
retract((Cabeza :- Cuerpo)),
retract(regla(Cabeza, Cuerpo)).

% Consultar los hechos y reglas actuales.

listar_hechos :-
findall(H, hecho(H), Hechos),
write('Hechos actuales: '), writeln(Hechos).

listar_reglas :-
findall((C :- B), regla(C, B), Reglas),
write('Reglas actuales: '), writeln(Reglas).
```

```
% Main

main :-

% Socrates es un hombre.
agregar_hecho(hombre(socrates)),
% Todos los hombres son mortales.
agregar_regla(mortal(X), hombre(X)),
% Consultar los hechos y reglas actuales.
listar_hechos,
listar_reglas,

% Consultar si Socrates es mortal.
(mortal(socrates) -> writeln('Socrates es mortal'); writeln('Socrates no es
mortal')),
% Eliminar el hecho de que Socrates es un hombre.
writeln('Eliminando hecho de que Socrates es un hombre...'),
eliminar_hecho(hombre(socrates)),

% Consultar si Socrates sigue siendo mortal.
(mortal(socrates) -> writeln('Socrates es mortal'); writeln('Socrates no es
mortal')),

% Eliminar la regla de mortalidad.
writeln('Eliminando regla de mortalidad...'),
eliminar_regla(mortal(X), hombre(X)),

% Consultar los hechos y reglas actuales.
listar_hechos,
listar_reglas.
```

VIII. ¿Qué es una refutación?

Una **refutación** consiste en una serie finita de metas y submetas:

$$(M, M', M'', \ldots, M^{n}, \square)$$

en la que la última meta corresponde a la cláusula vacía. Cada una de estas metas se obtiene a partir de la anterior utilizando el método de resolución.

IX. Escribe en *Prolog* las reglas y hechos necesarios para definir los siguientes predicados:

prefijo(Xs, Ys) : Cierto si la lista Xs es el prefijo de Ys.

```
% Caso base:Lista vacía es prefijo de cualquier lista.

prefijo([], _).

% Caso recursivo: La cabeza y la cola de la lista deben coincidir.
% CL = Cabezas de las listas
% Tx = Colas de las listas

prefijo([CL|T1], [CL|T2]) :-
prefijo(T1, T2).
```

X. Ilustra con un ejemplo que es el corte en *Prolog*.

Ejemplo 1

```
maximo(X,Y,X) :- X < Y.
maximo(X,Y,X) :- X >= Y.

max(X,Y,X) :- X < Y, !.
max(X,Y,X) :- X >= Y.
```

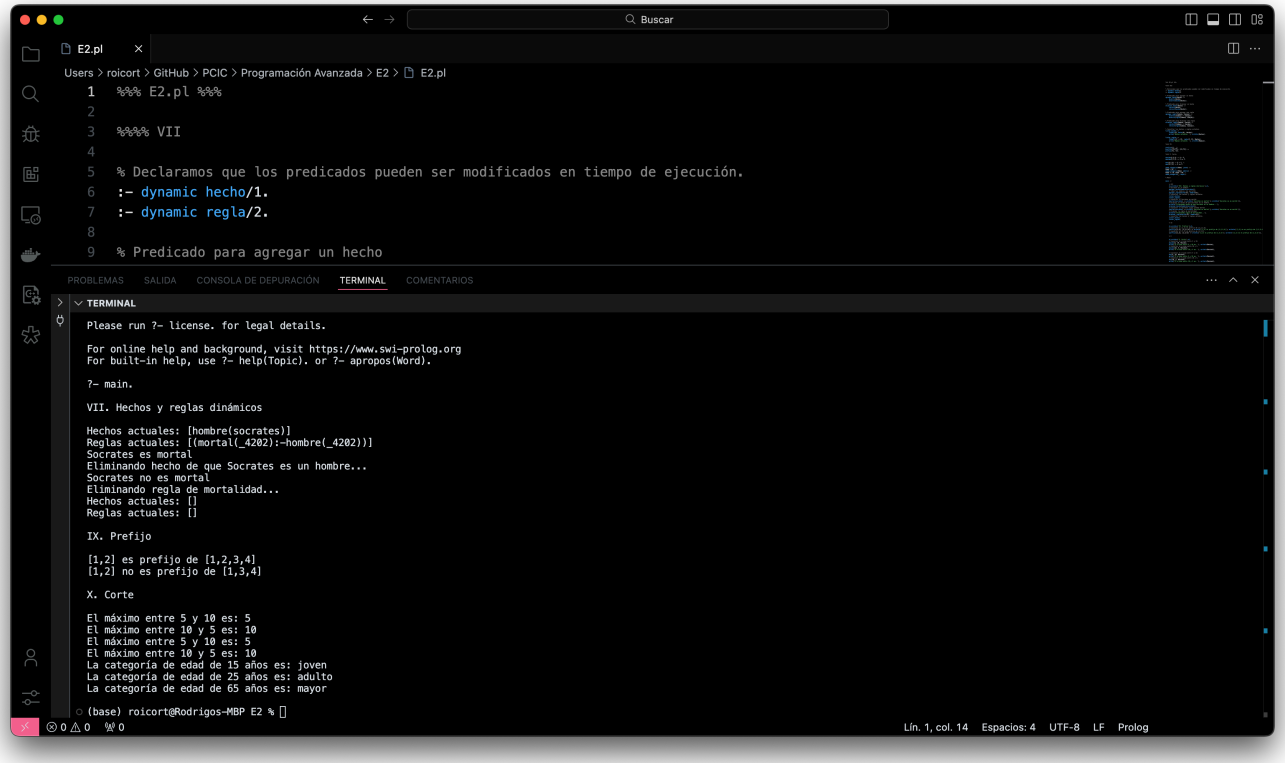
Para este ejemplo se puede observar sin el uso del corte, en Prolog aunque en la primera clausula tenga un resultado válido seguirá con la próxima clausula, a diferencia con la consulta con corte donde Prolog unificara el valor de x como el máximo y se detendrá la evaluación de un clausula posterior.

Ejemplo 2

```
edad_categoria(Edad, joven) :-
Edad < 18, !.
edad_categoria(Edad, adulto) :-
Edad >= 18, Edad < 60, !.
edad_categoria(_, mayor).
```

El corte garantiza que Prolog no evalúe las cláusulas adicionales, cuando encuentra una categoría válida regresa el resultado obtenido optimizando el rendimiento del programa.

Ejecución


Buscar
E2.pl
Users > roicort > GitHub > PCIC > Programación Avanzada > E2 > E2.pl
1 %%% E2.pl %%%
2
3 %%%% VII
4
5 % Declaramos que los predicados pueden ser modificados en tiempo de ejecución.
6 :- dynamic hecho/1.
7 :- dynamic regla/2.
8
9 % Predicado para agregar un hecho
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL COMENTARIOS
TERMINAL
Please run ?- license. for legal details.
For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).
?- main.
VII. Hechos y reglas dinámicos
Hechos actuales: [hombre(socrates)]
Reglas actuales: [(mortal(_4202):-hombre(_4202))]
Socrates es mortal
Eliminando hecho de que Socrates es un hombre...
Socrates no es mortal
Eliminando regla de mortalidad...
Hechos actuales: []
Reglas actuales: []
IX. Prefijo
[1,2] es prefijo de [1,2,3,4]
[1,2] no es prefijo de [1,3,4]
X. Corte
El máximo entre 5 y 10 es: 5
El máximo entre 10 y 5 es: 10
El máximo entre 5 y 10 es: 5
El máximo entre 10 y 5 es: 10
La categoría de edad de 15 años es: joven
La categoría de edad de 25 años es: adulto
La categoría de edad de 65 años es: mayor
(base) roicort@Rodrigos-MBP E2 %
Lín. 1, col. 14 Espacios: 4 UTF-8 LF Prolog