

PARADIGMAS DE PROGRAMACIÓN
3er Parcial: Programación Funcional

INSTRUCCIONES:

- ✓ Las soluciones de los Ejercicios 1, 2a y 2b deben entregarse en estas hojas.
- ✓ Resolver los Ejercicios 2c, 3 y 4 cada uno en una hoja separada.
- ✓ Poner nombre en TODAS las hojas que entregue.
- ✓ Numerar TODAS las hojas indicando en cada una el total (nro. hoja/total).

Ejercicio 1 (25 puntos)**a) En el Paradigma Funcional:**

| | |
|--|---|
| | Se utiliza la operación de asignación de un valor a una variable. |
| | Una función puede no retornar ningún resultado. |
| | Una función puede ser argumento de otra función. |
| | Una función siempre debe tener asociado un nombre. |
| | Una función no puede retornar como resultado una función. |
| | Ninguna de las afirmaciones anteriores es correcta. |

b) En el Cálculo Lambda:

| | |
|--|--|
| | La cantidad de formas normales que tiene un término depende del número de variables ligadas de dicho término. |
| | La aplicación reiterada de la Regla Beta siempre garantiza encontrar la forma normal de un término independientemente de la estrategia de reducción utilizada. |
| | Si un término tiene más de un redex tiene más de una forma normal. |
| | Aplicar la Regla Beta a un término siempre permite encontrar un término equivalente. |
| | Ninguna de las anteriores. |

c) Dado el siguiente término del Cálculo Lambda:

$$(\lambda x. \lambda x. (\lambda y. (f\ y) \ (x\ y))\ w)$$

| | |
|--|--|
| | Tiene un único redex. |
| | Con una única aplicación de la regla Beta (usando estrategia de reducción por Orden Normal) permite obtener su formal normal. |
| | Con una única aplicación de la regla Beta (usando estrategia de reducción por Orden Normal) permite obtener el término: $\lambda x. (\lambda y. (f\ y) \ (x\ y))$ |
| | Con una única aplicación de la regla Beta (usando estrategia de reducción por Orden Aplicativo) permite obtener el término: $\lambda x. (\lambda y. (f\ y) \ (x\ y))$ |
| | Ninguna de las anteriores. |

d) El lenguaje de programación Scheme:

| | |
|--|---|
| | Siempre utiliza orden normal para evaluar las expresiones. |
| | A veces utiliza orden normal para evaluar las expresiones, dependiendo del tipo de expresión que se evalúa. |
| | No admite funciones como argumentos de funciones. |

| | |
|--|---|
| | Sólo admite funciones de 1 argumento como argumento de otras funciones. |
| | Al evaluar la función (lambda (z) (if (pair? z) z (lambda (x) (z x)))), puede retornar una lista. |
| | Ninguna de las afirmaciones anteriores es correcta. |

e) Si se evalúa la siguiente expresión en Scheme: (list 1 + 3 * 4)

| | |
|--|--|
| | Devuelve una lista que tiene como único elemento el número 13. |
| | Devuelve una lista de 5 elementos. |
| | Produce un error. |
| | Devuelve una lista que tiene como único elemento el número 16. |
| | Ninguna de las afirmaciones anteriores es correcta. |

Ejercicio 2 (15 puntos)

Dado el siguiente término del Cálculo Lambda:

$$((\lambda x. (x \ \lambda z. z) \ \lambda y. (y \ (y \ z))) \ \lambda y. (x \ y))$$

- a) Marque en la expresión anterior, cuántos y cuáles (si hubiera) son los redex que contiene.
- b) Para cada variable, indicar el número de ocurrencias libres y ligadas de la misma, completando la siguiente tabla:

| Variable | Ocurrencias Libres | Ocurrencias Ligadas |
|----------|--------------------|---------------------|
| | | |
| | | |
| | | |

- c) Hallar la forma normal del término indicando claramente las reglas utilizadas en cada paso.

Ejercicio 3 (25 puntos)

Definir la función (**los-que-cumplen f**) de orden superior de 1 argumento, donde **f** es una función de 1 argumento que retorna #t o #f.

El resultado de evaluar (**los-que-cumplen f**) es una función de 1 argumento, el cual será un árbol binario cuyos nodos almacenan números (ver en el siguiente párrafo la representación sugerida para los árboles).

Al aplicar la función retornada por (**los-que-cumplen f**) a un árbol binario, retornará una lista con los valores en los nodos del árbol que evalúen #t si se les aplica la función **f**.

Se propone representar un árbol binario como una lista de 3 elementos: la raíz, el subárbol izquierdo y el subárbol derecho. Los subárboles son también árboles con esta representación. El árbol vacío se representa con 'nil'. En la tabla que sigue se muestran algunos ejemplos:

| |
|--|
| <pre>(define arbol '(3 (5 nil nil) (7 nil nil)))</pre> |
|--|

| | |
|--|--|
| <pre>(define arbol2 '(12 (23 (4 nil nil) (5 nil nil)) (14 (24 nil nil) nil)))</pre> | |
| <pre>(define arbol3 '(12 (36 (44 nil nil) (82 nil nil)) (14 (24 nil nil) (66 nil nil))))</pre> | |

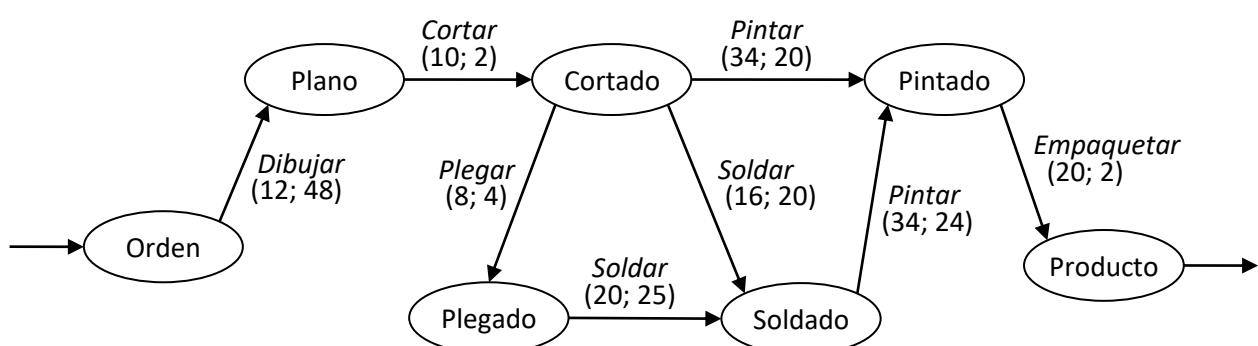
Teniendo en cuenta estos árboles binarios definidos en Scheme, a continuación se muestran ejemplos de la aplicación de la función **los-que-cumplen**.

```
(define menor10 (lambda (nro) (< nro 10)))
; even? Retorna #t si el argumento es un número par, #f en caso contrario
; odd? Retorna #t si el argumento es un número impar, #f en caso contrario

> (los-que-cumplen menor10)
#<procedure>
> ((los-que-cumplen odd?) arbol)
(3 5 7)
> ((los-que-cumplen odd?) arbol2)
(23 5)
> ((los-que-cumplen even?) arbol2)
(12 4 14 24)
> ((los-que-cumplen menor10) arbol2)
(4 5)
> ((los-que-cumplen menor10) arbol3)
()
```

Ejercicio 4 (35 puntos)

Un proceso de producción tiene un punto de entrada y un estado final, pero en el medio existen diferentes alternativas válidas en cuanto a la selección de los pasos intermedios. El proceso puede ser representado por un grafo en el cual, cada nodo se encuentra asociado a un estado de la orden y los arcos representan una actividad productiva que lleva la orden de un estado a otro. Cada actividad tiene asociado un costo y un tiempo. Un ejemplo puede verse en la figura siguiente:



Para el proceso anterior se propone la siguiente representación en el lenguaje Scheme:

```
(define grafo-proceso
  '((orden (dibujar 12 48) plano) (plano (cortar 10 2) cortado)
    (cortado (plegar 8 4) plegado) (cortado (soldar 16 20) soldado)
    (cortado (pintar 34 20) pintado) (plegado (soldar 20 25) soldado)
    (soldado (pintar 34 24) pintado) (pintado (empaquetar 20 2) producto)))
)
```

En esta representación sólo se indican las transiciones de un estado a otro mediante la realización de una actividad. Se asume que no existen estados sin al menos una transición.

Se solicita que defina en Scheme las siguientes funciones:

- ✓ a) (**transicion GP nodo-desde actividad**) que devuelve todos los datos de la transición al realizar la actividad indicada por el tercer argumento desde el nodo indicado por el segundo. El argumento *GP* es el grafo del proceso de acuerdo a la representación presentada anteriormente.

Note en los ejemplos que esta función devuelve alguna de las listas contenidas en la lista *GP*. Si la transición no existe, debe retornar falso.

```
> (transicion grafo-proceso 'plano 'cortar)
(plano (cortar 10 2) cortado)

> (transicion grafo-proceso 'cortado 'plegar)
(cortado (plegar 8 4) plegado)

> (transicion grafo-proceso 'soldado 'cortar)
#f
```

- ✓ b) (**nodo-hasta transicion**) que devuelve el nodo destino de ejecutar la transición recibida como argumento. Ver los ejemplos que siguen:

```
> (nodo-hasta '(plano (cortar 10 2) cortado))
cortado

> (nodo-hasta (transicion grafo-proceso 'cortado 'plegar))
plegado
```

- ✓ c) (**lista-actividades GP**) que devuelve una lista con los nombres de las actividades del proceso representado por *GP*, sin elementos repetidos.

```
> (lista-actividades grafo-proceso)
(dibujar cortar plegar soldar pintar empaquetar)
```

- ✓ d) (**proceso-valido? GP info-proceso**) que devuelve verdadero si *info-proceso* representa un proceso válido según lo indicado en el grafo *GP*.

El argumento *info-proceso* es una lista de 3 elementos, donde los dos primeros son el nodo inicial (*N_I*) y el nodo final (*N_F*) del proceso y el tercer elemento es una lista con las actividades que llevarían desde *N_I* a *N_F*. En caso de que *info-proceso* no corresponda con un camino válido en el grafo, la función deberá retornar falso.

```
> (proceso-valido? grafo-proceso
  '(orden producto (dibujar cortar plegar soldar pintar empaquetar)))
#t

> (proceso-valido? grafo-proceso
  '(orden producto (dibujar plegar soldar pintar cortar)))
#f
```