Facultad de Ciencias, UNAM
Lenguajes de Programación
Examen Parcial 3

Rubí Rojas Tania Michelle

04 de febrero de 2021

1. Evalúa las siguientes expresiones usando cada uno de los pasos de parámetros que se solicitan, debes de poner la última expresión a evaluar antes de dar el resultado final de la misma.

```
{with* {{i -1} {j -1}
          {swap {fun {x y}
                     {seqn {set tmp x}
                           {set x y}
                           {set y tmp}}}}
        {seqn {swap i j}
              {- j i}}}}
```

- Usando paso de parámetros por valor.
  Solución: La expresión que debemos evaluar es

```
{seqn {swap i j}
      {- j i}}
```

  Como se trata de una expresión `seqn`, entonces primero debemos evaluar la función `swap`. De esta forma,

```
{swap i j} = {swap {fun {x y}
                    {seqn {set tmp x}
                          {set x y}
                          {set y tmp}}}}
```

  Esta expresión es una función que recibe los parámetros x, y; y en la aplicación le estamos pasando i, j. Por lo que, los parámetros formales serán x, y y los reales serán i, j. Como vamos a usar paso por valor, entonces

```
{set tmp x}    ;; tmp=x=-1
```

  es una copia del valor $i = -1$. De igual forma, cuando hacemos

```
{set x y}    ;; x=y=-1
```

  pasamos una copia del valor de $y = -1$. Posteriormente, hacemos

```
{set y tmp}    ;; y=tmp=-1
```

Sin embargo, el intercambio de valores se realizó dentro de la función, por lo que al ejecutar la última expresión del `seqn` obtenemos

```
{- j i} = {- (-1) (-1)}
        = 0
```

pues los valores $i = -1$, $j = -1$ siguen conservando sus valores. Así, el resultado final de evaluar esta expresión es 0.
Por otro lado, el ambiente y la memoria quedan de la siguiente manera:

swap	2
j	1
i	0

2	{fun {x y} {seqn {set tmp x} {set x y} {set y tmp}}}
1	-1
0	-1

- Usando paso de parámetros por referencia.
  Solución: La expresión que debemos evaluar es

```
{seqn {swap i j}
      {- j i}}
```

Como se trata de una expresión `seqn`, entonces primero debemos evaluar la función `swap`. De esta forma,

```
{swap i j} = {swap {fun {x y}
                    {seqn {set tmp x}
                          {set x y}
                          {set y tmp}}}}
```

Esta expresión es una función que recibe los parámetros x, y; y en la aplicación le estamos pasando i, j. Por lo que, los parámetros formales serán x, y y los reales serán i, j. Como vamos a usar paso por referencia, entonces

```
{set tmp x}    ;; tmp=x=-1
```

pasa la dirección en memoria de i, que es 0. En esta dirección encontramos que el valor de i es -1. De igual forma, cuando hacemos

```
{set x y}    ;; x=y=-1
```

pasamos la dirección en memoria de j, que es 1. En esta dirección encontramos que el valor de j es -1. Posteriormente, hacemos

```
{set y tmp}    ;; y=tmp=-1
```

Sin embargo, el intercambio de valores se realizó en el ambiente global, por lo que al ejecutar la última expresión del `seqn` obtenemos

```
{- j i} = {- (-1) (-1)}
        = 0
```

pues los valores $i = -1$, $j = -1$ esta vez sí fueron modificados (pero como tienen el mismo valor, no se puede apreciar la diferencia). Así, el resultado final de evaluar esta expresión es 0.

Por otro lado, el ambiente y la memoria quedan de la siguiente manera:

swap	2
j	1
i	0

2	{fun {x y} {seqn {set tmp x} {set x y} {set y tmp}}}
1	-1
0	-1

2. ¿Cuáles son las diferencias principales entre el paso de parámetros por necesidad y por nombre?

   Solución: El paso de parámetros por necesidad sigue la siguiente filosofía: *"Si ya se evaluó una función y posteriormente se vuelve a llamar, entonces nos quedamos con el valor obtenido en esa primera evaluación y ya no la volvemos a evaluar"*; mientras que el paso de parámetros por nombre realiza las llamadas a función por su nombre (es decir, evalúa una función las veces que sea necesaria, sin importar cuántas veces aparezca). Por ejemplo, si la función foo dentro de un programa es llamada n veces, entonces

   - Usando paso de parámetros por necesidad, la función foo sólo se evalúa una vez y para las siguietes $n-1$ llamadas regresa el valor obtenido de esa primera llamada a función.
   - Usando paso de parámetros nombre, la función foo se evalúa las n veces que es llamada.

   Otra diferencia es que el paso por necesidad no preserva siempre el concepto de estado (pues no va mutando los valores correspondientes en cada una de las llamadas), mientras que el paso por nombre sí lo hace (en cada una de sus llamadas muta los valores correspondientes).

3. Del siguiente código en Racket:

```
(define (filter-neg l)
  (cond
    [(empty? l) empty]
    [else
      (if (< (first l) 0)
          (cons (first l) (filter-neg (rest l)))
          (filter-neg (rest l)))]))
```

   a) Convierte el código anterior a CPS.
   Solución:

```
(define (filter-neg l)
  (filter-neg/k l (lambda (x) x)))

(define (filter-neg/k l k)
  (cond
    [(empty? l) (k '())]
    [else
      (if (< (first l) 0)
          (filter-neg/k (rest l)
                        (lambda (v) (k (cons (first l) v))))
          (filter-neg/k (rest l) k))]))
```

b) ¿Qué regresa la función que convertiste a CPS cuando recibe la lista `'(0 1 -1 0 -4 1 -2)`?
SOLUCIÓN:

```
(filter-neg '(0 1 -1 0 -4 1 -2)) = (filter-neg/k '(0 1 -1 0 -4 1 -2) (λ(x) x))
                                 = (filter-neg/k '(1 -1 0 -4 1 -2) (λ(x) x))
                                 = (filter-neg/k '(-1 0 -4 1 -2) (λ(x) x))
                                 = (filter-neg/k '(0 -4 1 -2)
                                                 (λ(v) ((λ(x) x) (cons -1 v))))
                                 = (filter-neg/k '(-4 1 -2)
                                                 (λ(v) ((λ(x) x) (cons -1 v))))
                                 = (filter-neg/k '(1 -2)
                                                 (λ(v) ((λ(v) ((λ(x) x) (cons -1 v)))
                                                        (cons -4 v))))
                                 = (filter-neg/k '(-2)
                                                 (λ(v) ((λ(v) ((λ(x) x) (cons -1 v)))
                                                        (cons -4 v))))
                                 = (filter-neg/k '()
                                                 (λ(v) ((λ(v) ((λ(v) ((λ(x) x)
                                                 (cons -1 v))) (cons -4 v))) (cons -2 v))))
                                 = ((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (cons -1 v)))
                                    (cons -4 v))) (cons -2 v))) '())
                                 = ((λ(v) ((λ(v) ((λ(x) x) (cons -1 v))) (cons -4 v)))
                                    '(-2))
                                 = ((λ(v) ((λ(x) x) (cons -1 v))) '(-4 -2))
                                 = ((λ(x) x) '(-1 -4 -2))
                                 = '(-1 -4 -2)
```

Por lo tanto, la función `filter-neg` regresa la lista `'(-1 -4 -2)`.

4. Da la expresión asociada a la continuación y el resultado de dicha expresión, para cada uno de los siguientes códigos:

```
> (define c #f)
> (+ 1 (+ 2 (+ 3 (+ (let/cc here
                       (set! c here)
                       4) 5))))
```

SOLUCIÓN:

```
> (define c empty)
> (+ 1 (+ 2 (+ 3 (+ (let/cc here (set! c here) 4) 5))))
> (c 20)
```

5. Da el juicio de tipo para la siguiente expresión, la cual está implementada en el lenguaje RACKET.

```
{let {x {+ 1 1}}
  {{fun {y} {+ 0 x}} {+ 2 2}}}
```

Solución:

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash \texttt{1:number}} \quad \dfrac{}{\varnothing \vdash \texttt{1:number}}}{\varnothing \vdash \texttt{\{+ 1 1\}:number}} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{0:number}} \quad \dfrac{}{\Gamma \vdash \texttt{x:number}}}{\Gamma \vdash \texttt{\{+ 0 x\}:number}} \qquad \dfrac{\dfrac{}{\Gamma' \vdash \texttt{2:number}} \quad \dfrac{}{\Gamma' \vdash \texttt{2:number}}}{\Gamma' \vdash \texttt{\{+ 2 2\}:number}}}{\Gamma \vdash \texttt{\{\{fun \{y\} \{+ 0 x\}\} \{+ 2 2\}\}:number}}}{\varnothing \vdash \texttt{\{let \{x \{+ 1 1\}\} \{\{fun \{y\} \{+ 0 x\}\} \{+ 2 2\}\}\}:number}}$$

donde

- Γ = [x ←number]
- Γ' = [x ←number, y ←number]

6. Realiza la inferencia de tipos de la siguiente expresión, mencionándo al término de la inferencia, los tipos de cada una de las variables de la función.

```
(define foo
  (lambda (lst item)
    (cond
      [(nempty? nlist) nempty]
      [(nequal? item (nfirst lst)) (nrest lst)]
      [else (ncons (nfirst lst) (foo (nrest lst) item))])))
```

Solución: Primero, identificamos cada una de nuestras sub-expresiones y las enumeramos.

- 1 `(lambda (lst item) (cond [(nempty? nlist) nempty] [(nequal? item (nfirst lst)) (nrest lst)] [else (ncons (nfirst lst) (foo (nrest lst) item))]))`
- 2 `(cond [(nempty? nlist) nempty] [(nequal? item (nfirst lst)) (nrest lst)] [else (ncons (nfirst lst) (foo (nrest lst) item))])`
- 3 `(nempty? nlist)`
- 4 `nempty`
- 5 `(nequal? item (nfirst lst))`
- 6 `(nfirst lst)`
- 7 `(nrest lst)`
- 8 `else`
- 9 `(ncons (nfirst lst) (foo (nrest lst) item))`
- 10 `(nfirst lst)`
- 11 `(foo (nrest lst) item)`
- 12 `(nrest lst)`

Luego, vamos a analizar el tipo de expresiones que encontramos.

- Para la cajita uno,

$$\begin{aligned}[[\boxed{1}]] &= [[\texttt{(lambda (lst item) (cond [(nempty? nlist) nempty]}\\ &\qquad \texttt{[(nequal? item (nfirst lst)) (nrest lst)]}\\ &\qquad \texttt{[else (ncons (nfirst lst) (foo (nrest lst) item))]))}]]\\ &= [[\texttt{lst}]] \times [[\texttt{item}]] \to [[\texttt{(cond [(nempty? nlist) nempty]}\\ &\qquad \texttt{[(nequal? item (nfirst lst)) (nrest lst)]}\\ &\qquad \texttt{[else (ncons (nfirst lst) (foo (nrest lst) item))])}]]\\ &= [[\texttt{lst}]] \times [[\texttt{item}]] \to [[\boxed{2}]]\end{aligned}$$

- Para la cajita dos,

$$\begin{aligned}[[\boxed{2}]] &= [[\texttt{(cond [(nempty? nlist) nempty] [(nequal? item (nfirst lst)) (nrest lst)]}\\ &\qquad \texttt{[else (ncons (nfirst lst) (foo (nrest lst) item))])}]]\\ &= [[\texttt{(cond [}\boxed{3}\ \boxed{4}\texttt{] [}\boxed{5}\ \boxed{7}\texttt{] [}\boxed{8}\ \boxed{9}\texttt{])}]]\\ &= [[\boxed{3} \to \boxed{4}]]\texttt{or}[[\boxed{5} \to \boxed{7}]]\texttt{or}[[\boxed{8} \to \boxed{9}]]\end{aligned}$$

  de donde
  - $[[\boxed{3}]] = \texttt{boolean}$
  - $[[\boxed{5}]] = \texttt{boolean}$
  - $[[\boxed{4}]] = [[\boxed{7}]] = [[\boxed{9}]]$
- Para la cajita tres,

$$[[\boxed{3}]] = [[\texttt{(nempty? nlist)}]]$$

  donde
  - $[[\texttt{(nempty? list)}]] = \texttt{boolean}$
  - $[[\texttt{list}]] = \texttt{nlist}$
- Para la cajita cuatro,

$$[[\boxed{4}]] = [[\texttt{nempty}]] = \texttt{nlist}$$

- Para la cajita cinco,

$$\begin{aligned}[[\boxed{5}]] &= [[\texttt{(nequal? item (nfirst lst))}]]\\ &= [[\texttt{(nequal? item }\boxed{6}\texttt{)}]]\end{aligned}$$

  de donde
  - $[[\texttt{(nequal? item }\boxed{6}\texttt{)}]] = \texttt{boolean}$
  - $[[\texttt{item}]] = \texttt{number}$
  - $[[\boxed{6}]] = [[\texttt{(nfirst lst)}]]$
    - $[[\texttt{(nfirst lst)}]] = \texttt{number}$
    - $[[\texttt{lst}]] = \texttt{nlist}$
- Para la cajita siete,

$$[[\boxed{7}]] = [[\texttt{(nrest lst)}]]$$

  donde
  - $[[\texttt{(nrest lst)}]] = \texttt{nlist}$
  - $[[\texttt{lst}]] = \texttt{nlist}$
- Para la cajita ocho,

$$[[8]] = [[\texttt{else}]] = [[\texttt{true}]] = \texttt{boolean}$$

- Para la cajita nueve,

$$[[\boxed{9}]] = [[\texttt{(ncons (nfirst lst) (foo (nrest lst) item))}]]$$
$$= [[\texttt{(ncons } \boxed{10} \ \boxed{11}\texttt{)}]]$$

de donde

- $[[\boxed{9}]] = \texttt{nlist}$
- $[[\boxed{10}]] = [[\boxed{6}]] = \texttt{nlist}$
- $[[\boxed{11}]] = [[\texttt{(foo (nrest lst) item)}]] = [[\texttt{(foo } \boxed{12} \texttt{ item)}]] =$ $[[\boxed{12}]] \times [[\texttt{item}]] \to [[\texttt{(foo (nrest lst) item)}]]$ con
  - $[[\boxed{12}]] = [[\boxed{7}]] = \texttt{nlist}$

Por lo tanto, los tipos de las variables de la función son `lst = nlist` e `item = number`; por lo que el tipo de la función `foo` es

```
foo: (nlist x number) -> nlist
```

7. Utiliza el algoritmo de unificación visto en clase en la expresión

```
((lambda (y) (* y (+ 0 0))) 1)
```

Solución: Primero, identificamos cada una de nuestras sub-expresiones y las enumeramos.

- $\boxed{1}$ `((lambda (y) (* y (+ 0 0))) 1)`
- $\boxed{2}$ `(lambda (y) (* y (+ 0 0)))`
- $\boxed{3}$ `(* y (+ 0 0))`
- $\boxed{4}$ `(+ 0 0)`
- $\boxed{5}$ `0`
- $\boxed{6}$ `0`
- $\boxed{7}$ `1`

begincenter

8. Da las sentencias de variables de tipo para las siguientes funciones de `Racket`:

- `list-length`
  Solución:
  $$\texttt{list-length: } \forall\alpha.\texttt{list}(\alpha) \to \texttt{number}$$
- `fibonacci`
  Solución:
  $$\texttt{fibonacci: } \forall n.\texttt{number}(n) \to \texttt{number}$$

9. Selecciona dos características de la siguiente lista del Paradigma Orientado a Objetos:

- Herencia
- Encapsulamiento de información
- Abstracción
- Modularidad

Ahora define y explícalas, usando ejemplos.

Solución:

- Herencia

  La herencia es el procedimiento por el cual una clase hereda los atributos y métodos de otra clase. La clase cuyas propiedades y métodos se heredan se conoce como clase Padre. Y la clase que hereda las propiedades de la clase padre es la clase hija.

  Ejemplo: Supongamos que tenemos alumnos universitarios. Algunos son alumnos normales, otros son de intercambio y otros son becarios. Probablemente tendremos una clase `Alumno` con una serie de métodos como `asistir_a_clase()`, `hacer_examen()`, etc., y una serie de atributos como `nombre`, `número de cuenta`, etc.; que son comúnes a todos los alumnos, pero hay operaciones que son diferentes en cada tipo de alumno como `pagar_mensualidad()` (los becarios no pagan) o `matricularse()` (los de intercambio se matriculan en su universidad de origen). En este caso, la clase Padre sería `Alumno` y las clases hijas serán `AlumnoNormal`, `AlumnoIntercambio` y `AlumnoBecado`.

- Abstracción

  Una clase abstracta es aquella en la que no podemos instanciar objetos. Expresa las características específicas de un objeto, aquellas que lo distinguen de los demás tipos y que logran definir límites conceptuales respecto a quién está haciéndo dicha abstracción.

  Ejemplo: Supongamos que queremos aplicar la Abstracción a las Aves. Un `Ave` es sólo un concepto abstracto que no puede instanciarse. Existen muchas aves que heredan sus características (como el pico, las plumas, las alas, las patas, volar, etc) y ellas sí pueden existir por sí mismos. Un pájaro o un pato serían ejemplos de aves que sí pueden existir de la clase abstracta `Ave`.

10. Se tiene el siguiente código que intenta dar una versión *memoizada* de la función que calcula el $n-$ésimo número de la sucesión de `Tribonacci`:

```
(define (tribonacci n)
  (if (< n 3)
      1
      (+ (tribonacci (- n 1)) (tribonacci (- n 2)) (tribonacci (- n 3)))))


(define (tribonacci-memo n tabla)
  (let ([res (hash-ref tabla n 'ninguno)])
    (cond
      [(equal? res 'ninguno)
       (hash-set! tabla n (tribonacci n))
       (hash-ref tabla n)]
      [else res])))
```

   a) ¿Por qué la función `tribonacci-memo` del código anterior no hace uso correcto de la técnica de memoización?

   Solución: Esta función no hace uso correcto de la memoización porque gracias a la línea

   ```
   (hash-set! tabla n (tribonacci n))
   ```

   efectivamente vamos agregando nuevos registros a la tabla, pero como el valor lo obtenemos llamando a la función `(tribonacci n)`, entonces se están generando sus respectivas llamadas a función repetidas, es decir, al utilizar la función `(tribonacci n)` estamos volviendo a calcular valores que ya habíamos obtenido antes (lo cual queremos evitar usando memoización).

   b) Describir cómo se corregiría este código para que emplee bien la técnica de memoización.

   Solución: Primero, debemos cambiar la línea

```
(hash-set! tabla n (tribonacci n))
```

por la definición de una variable `nuevo`, la cual debe calcular el valor para el parámetro realizando la operación

```
(+ (tribonacci (- n 1)) (tribonacci (- n 2)) (tribonacci (- n 3)))
```

y después debemos modificar la línea

```
(hash-ref tabla n)
```

para poder asignar este valor a la tabla y posteriormente regresarlo.
Por lo tanto, la función modificada queda como:

```
(define (tribonacci-memo n tabla)
  (let ([res (hash-ref tabla n 'ninguno)])
    (cond
      [(equal? res 'ninguno)
       (define nuevo
         (+ (tribonacci (- n 1))
            (tribonacci (- n 2))
            (tribonacci (- n 3))))
       (hash-set! tabla n nuevo)
       nuevo]
      [else res])))
```