

Control 3 – Lenguajes de Programación

Departamento de Ciencias de la Computación

Universidad de Chile

Profesor: Éric Tanter

17 de Noviembre 2008

2 horas / 2 puntos por pregunta

1. Considere el siguiente lenguaje:

```
<expr> ::= <num> | <id>
         | (+ <expr> <expr>)
         | (lambda (<id>) <expr>)
         | (let (<id> <expr>) <expr> )
```

- a) Escriba la regla de juicio para el **let**.
b) En este lenguaje, ¿que hace el siguiente programa?

```
{{fun {x} {x x}}
 {fun {x} {x x}}}
```

¿Es válido en tipos? ¿A su juicio, es posible escribir en este lenguaje un programa válido que no termina?

- c) Agregue una expresión para definición recursiva, **letrec**. Escriba la regla de juicio correspondiente.
d) ¿Es ahora posible escribir un programa válido que no termina? Justifique.

2. Considere el siguiente intérprete de un lenguaje con estructuras de datos mutables (box), escrito usando *store-passing style* :

```
(define (interp expr env store)
  (type-case Expr expr
    ...
    [app (fun-expr arg-expr)
      (type-case Value*Store (interp fun-expr env store)
        [v*s (fun-value fun-store)
          (local ([arg-value (interp arg-expr env fun-store)]
                  [define new-loc (next-location fun-store)])
            (interp (closureV-body fun-value)
                     (aSub (closureV-param fun-value)
                           new-loc
                           (closureV-env fun-value))
                     (aSto new-loc
                           arg-value
                           fun-store))) ..)]
```

- a) Qué problema tiene este intérprete?
 - b) Escriba un programa que ilustre que la semántica del **app** es errónea.
 - c) Modifique el intérprete para que funcione adecuadamente.
3. En clases, vimos que es posible dar una representación procedural de tipos de datos abstractos (como ambientes, colas, etc.). Es decir, es posible implementar esos tipos de datos como *funciones* (o conjuntos de funciones), y así proveer toda la interfaz del tipo de datos considerado sin usar **define-type**.
- También hemos visto como agregar estado a funciones, gracias al scope estático. Además, si usamos mutación, podemos permitir que una función tenga estado mutable. Recuerde que definiciones de funciones pueden estar anidadas.

Consideremos un tipo de dato abstracto *tree*, tal que representa un árbol binario de búsqueda, que solamente permita las operaciones de insertar y buscar en el árbol. Defina, usando una *representación procedural* de un árbol binario de búsqueda, las funciones **create-tree**, **search-tree** y **insert-tree** tal que:

```
(define mytree (create-tree))
..
(insert-tree mytree 5)
(insert-tree mytree 3)
(insert-tree mytree 7)
(insert-tree mytree 4)
(insert-tree mytree 1)

(test (search-tree mytree 4) #t)
(test (search-tree mytree 8) #f)
```