

# Pauta Control 1 – Lenguajes de Programación

Departamento de Ciencias de la Computación

Universidad de Chile

Profesor: Éric Tanter

1 de Septiembre 2008

2 horas / 2 puntos por pregunta
---------------------------------

1. Definamos el lenguaje **PWAE** como una extensión del lenguaje **WAE** visto en clases con una primitiva **print-val**. Esta primitiva imprime en pantalla el valor pasado en parametro como una string, y retorna ese mismo valor. Por ejemplo:

```
> {+ 3 {print-val 4}}  
"4"  
7
```

- a) (0.5pt) Explique como extender la definición del lenguaje **WAE** para obtener **PWAE**.

Los pasos en orden para obtener **PWAE** en base a **WAE** son:

- 1) Extender la gramática concreta con una regla que soporte **print-val**
  - 2) Agregar al parser la condición adecuada para soportar **print-val** que produzca un  $\hat{\text{PWAE}}$  con la regla anterior mencionada
  - 3) Agregar a la función subst la condición para manejar la regla del **print-val**
  - 4) Por último en la función calc o interp se añade la condición para manejar la regla del **print-val** que toma su parámetro, lo calcula, imprime en pantalla el valor calculado con una primitiva de scheme y retorna el valor calculado
- b) (1.5pt) En el lenguaje **PWAE**, es cierto que los regímenes de substitución temprana (eager) y perezosa (lazy) son equivalentes? Demuestre su respuesta usando una expresión ejemplo, y desarrollando su evaluación paso a paso.

Analizando el siguiente ejemplo:

```
{with {a {print-val {/ 1 0}}}  
      7  
}
```

En este caso vemos que en una evaluación eager el interprete va calcular el parámetro del **print-val** obteniendo un error de división por cero, generando automáticamente una salida del interprete. En el caso de la evaluación lazy el interprete no va calcular el parámetro del **print-val** por que luego en la interpretación de la expresión el id **a** no se usa.

Por tanto concluimos que los regimenes de evaluación no son equivalentes en el PWAE, ya que no generan los mismo resultados esperados.

2. Considere un lenguaje con la siguiente gramática:

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

Defina en Scheme la función **free-vars** que retorna la lista de variables libres de una expresión dada. No se olvide de partir escribiendo al menos un test por cada tipo de expresión.

Implementación en el archivo p2.scm

3. Al definir la operación de substitución, uno de los problemas principales fue especificar bien lo que pasa cuando se usa el mismo identificador varias veces. Resulta que es posible deshacerse de los nombres, simplemente. Nicolas De Bruijn propuso reemplazar nombres por números, más bien, *índices*, que indican la profundidad de una asociación (binding).<sup>1</sup>

Por ejemplo, en vez de escribir: **{with {x 5}{+ x x}}**, podemos escribir: **{with 5 {+ <0> <0>}}**.

Por convención el alcance (scope) corriente es 0, entonces **x** se reemplazó por el índice **<0>**. Con esta representación, basta saber que la presencia de un **with** indica que entramos en un nuevo alcance.

Así mismo, la siguiente expresión:

```
{with {x 5}  
      {with {y 4}  
            {+ x y}}}
```

se convierte en:

---

<sup>1</sup>Esa representación es usada por (casi) todos los compiladores.

```
{with 5
  {with 4
    {+ <1> <0>}}}}
```

- a) (0.5pt) Escriba la conversión de la siguiente expresión usando índices de De Bruijn:

```
{with {x 5}
  {with {y {+ x 2}}
    {+ x
      {with {y {- y 1}}
        {+ y x}}}}}}
```

- b) (0.5pt) Extiende la gramática del lenguaje WAE para acomodar expresiones con Índices de De Bruijn.
- c) (1pt) Implemente la función `toDeBruijn :: WAE ->WAE` que transforma una expresión con `with` normales a una expresión con `with` modificado e índices De Bruijn.

Implementación en el archivo `p3-WAE.scm`