Examen 1: Lenguajes de Programación

Oscar Andrés Rosas Hernández [SoyOscarRH@gmail.com / 417024956]
Facultad de Ciencias, UNAM, CDMX, México

PRIMER PROBLEMA

(1 pt.) ¿Cuál es la complejidad del algoritmo de sustitución? Elabora tu respuesta usando algún ejemplo.

Pues depende, primero que nada veamos sobre que vamos a analizar la complejidad, dado un programa que tiene n expresiones with / let / bindings en general hay que ejecutar el algoritmo de sustitución para cada uno de ellos, el primero tiene que hacerlo sobre $n-1$ expresiones anidadas, el siguiente sobre $n-2$ y asi sucesivamente, eso es claramente la suma de gauss, y un clasico ejemplo de un algoritmo cuadratico, asi que la complejidad de hacerlo si alguna estructura de datos auxiliar es $O(n^2)$ en terminos de tiempo.

Por ejemplo podemos hacer algo como:

```
{with {a 1}
  {with {b 2}
    {with {c 3} }}}
```

Sin usar estructuras de datos hay que sustituir 3 cosas, la primera vez para hacerlo con el id a hay que visitar dos expresiones, para sustituir b solo hay que sustituir 1 expresion y para c ya nada.

Ahora si se usa alguna estructura de datos (la clasica es una pila / lista) o algo como una tabla hash se puede reducir la complejidad a $O(n)$, esto porque podemos encontrar el valor a remplazar en constante y hasta con moverle con el programa una vez.

SEGUNDO PROBLEMA

(1 pt.) Da una expresion usando la gramatica FWAE tal que una misma variable (supongamos x) aparezca en la misma expresion como instancia de ligado una vez (con el mismo nombre de identificador), aparezca ligada al menos una vez y aparezca exactamente una unica vez como identificador libre.

Creo que la forma mas compacta seria algo como:

```
{with {x x}
  x}
```

Ahi tenemos a x como identificador libree, como instancia de ligado y instancia ligada.

Algo mas explicito seria:

```
{with {y x}
  {with {x 2} x}}
```

La x de $\{with\{yx\}$ es libre, la x de $\{with\{x2\}$ es de ligado y la x es $x\}\}$ es de ligada.

TERCER PROBLEMA

(1 pt.) Convierte el siguiente codigo usando indices de Bruijn

```
  {with {a -1}
    {with {b 1}
      {with {c 2}
        {with {d {+ 2 1}}
          {* d {/ c {+ {* b a} {+ a a}}}}}}}}}
```

Facil seria asi:

```
  {with -1
    {with 1
      {with 2
        {with {+ 2 1}
          {* <:0> {/ <:1> {+ {* <:2> <:3>} {+ <:3> <:3>}}}}}}}}}
```

CUARTO PROBLEMA

(1 pt.) Convierte el siguiente codigo con indices de Bruijn a codigo dentro de la gramatica WAE. Las instancias de ligado se llaman x, y, z, a, b con respecto al orden de aparicion de las mismas.

```
{with 1
  {with 2
    {with 3
      {with {* <:0 0> <: 1 0>}
        {with 5
          {+ <: 0 0> {+ <:1 0> {+ <:2 0> {+ <:3 0> <:4 0>}}}}}}}}}
```

Va, jalo:

```
{with {x 1}
  {with {y 2}
    {with {z 3}
      {with {a {* z y}}
        {with {b 5}
          {+ b {+ a {+ z {+ y x}}}}}}}}}
```

QUINTO PROBLEMA

(1 pt.) A que se le conoce como azucar sintactica en un lenguaje de programacion.

Me gusta mucho como se dije en ingles, syntactic sugar, es un 'atajo (visual o aparentemente logico) que nos da un lenguaje de programacion para recudir la cantidad de codigo que se debe de escribir para una situacion comun.

Por ejemplo en JavaScript (mi lenguaje favorito es) estas dos funciones hacen lo mismo:

```
const double1 = x => 2 * x;

const double2 = (x) => {
  return 2 * x;
}
```

O en Racket:

```
(define (hello-simple name)
  (string-append "Hello " name))

(define hello
  (lambda (name)
    (string-append "Hello " name)))
```

SEXTO PROBLEMA

Ponga el ambiente en forma de pila (stack) para la siguiente expresion, y evalue la siguiente expresion usando alcance estatico y alcance dinamico. es necesario especificar cada una de las expresiones a evaluar con los respectivos valores

Va, esta si me va a tomar un rato, vamos.

Primero con alcance estatico:

```
{with {a 1}
  {with {b 1}
    {with {a 0}
      {with {foo1 {fun {x} {* x {+ b a}}}}
        {with {b 0}
          {with {a 1}
            {foo1 2}}}}}}}
```

Alcance Dinámico

Podemos ver a la pila como:

```
a     | 1
b     | 0
foo1  | fun {x} {* x {+ b a}}
a     | 0
b     | 1
a     | 1
```

Llegado a esto evaluamos, con pila. Recuerda que como es alcance dinamico buscamos a a, b desde el principio de la pila, en este caso las primeras dos entradas de la pila nos dan los valores y estamos listos para evaluar, nos queda lo siguiente.

```
{fun {x} {* x {+ b a}} 2}
{fun {x} {* x {+ 0 1}} 2}
{* 2 {+ 0 1}}
{* 2 1}
{* 2}
```

Pero por otro lado...

Alcance Estatico

Podemos ver a la pila como:

```
a     | 1
b     | 0
foo1  | fun {x} {* x {+ b a}} <- empieza de aqui pa' abajo
a     | 0
b     | 1
a     | 1
```

Ahora lo intesante con estatico es que al momento de hacer la sustitución de a, b vamos a empezar desde donde fue definida la funcion, las clasicas closures.

Con esto al momento de sustituir tenemos que:

```
{fun {x} {* x {+ b a}} 2}
{fun {x} {* x {+ 1 0}} 2}
{* 2 {+ 1 0}}
{* 2 1}
{* 2}
```

SEPTIMO PROBLEMA

(1 pt.) Escribe la definicion de la funcion recursiva agregaN en Racket, que reciba tres parametros, un elemento e un numero n y una lista l. La funcion debe regresar la lista resultante de agregar a la lista l, el elemento e en la posicion n. No puedes utilizar ninguna funcion nativa del lenguaje, exceptuando los operadores aritmeticos, car, cons, append e equal?

Listo :)

```
(define (agregaN element position l)
  (if (equal? position 0)
    (cons element l)
    (cons (car l) (agregaN element (- position 1) (cdr l)))))
```