Facultad de Ciencias, UNAM
Lenguajes de Programación
Examen Parcial I

Rubí Rojas Tania Michelle

30 de octubre de 2020

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

   SOLUCIÓN: Sabemos que un programa tiene n variables. El algoritmo hace la sustitución de las variables una por una, y por cada variable **de-ligado** hace el recorrido sobre las m posibles variables más que haya sobre el resto del programa. Notemos también que el algoritmo hace el recorrido por cada variable **de-ligado** (exista o no dentro del programa). Entonces, en el peor caso, si tenemos n variables **de-ligado**, el algoritmo de sustitución se realizará en cada una de las n variables del programa; por lo que la complejidad del algoritmo es de $O(n^2)$ (pues el recorrido que hace una variable **de-ligado** en el programa nos toma $O(n)$, pero esto se realiza n veces).

   Un ejemplo que muestra esto es:

```
{with {x_1 4}
 {with {x_2 4}
   {with {x_3 4}
     ...
     {with {x_n 4}
       {+ x_1 {+ x_2 {+ x_3 ... {+ x_n-1 x_n}}}}}}}}}
```

   donde tenemos n variables **de-ligado** y n variables **ligadas**. En este ejemplo, a cada variable **de-ligado** le corresponde una única variable **ligada**; pero el algoritmo no puede saber ésto, así que por cada variable **de-ligado** realizará el recorrido sobre las n variables **ligadas** que contiene nuestro programa para intentar sustituir el valor. Por lo que la complejidad será de $O(n^2)$.

2. Da una expresión usando la gramática FWAE tal que una misma variable (supongamos x) aparezca en la misma expresión como instancia de ligado una vez (con el mismo nombre de identificador), aparezca ligada al menos una vez y aparezca exactamente una única vez como identificador libre.

   SOLUCIÓN: La expresión propuesta es

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

   Donde tenemos que la variable de color rojo es una variable **de-ligado**, la variable de color azul es una variable **libre** (pues tiene que salir a buscar su valor fuera de esta expresión, y fuera de ésta no está ligada a nadie) y las variables de color verde son variables **ligadas**.

   Otra expresión podría ser

```
{+ x {with {x 2} {x x}}}
```

   donde la variable de color azul es una variable **libre** (pues no está ligada a nadie), la variable de color rojo es una variable **de-ligado** y las variables de color verde son **ligadas**.

3. Convierte el siguiente código usando índices de Bruijn o direcciones léxicas.

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

SOLUCIÓN:

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

4. Convierte el siguiente código con índices de Bruijn a código dentro de la gramática WAE. Las instancias de ligado se llaman x, y, z, a, b con respecto al órden de aparición 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>}}}}}}}}}
```

SOLUCIÓN:

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

5. ¿A qué se le conoce como azúcar sintáctica en un lenguaje de programación?

   SOLUCIÓN: La azúcar sintáctica es una especie de *sintáxis especial* que hace más fácil la escritura de algunas expresiones en un lenguaje de programación, haciéndo que éstas sean más *dulces* para el usuario. Por ejemplo, `with` es azúcar sintáctica de una aplicación de función.

6. Ponga el ambiente en forma de pila (stack) para la siguiente expresión, y evalúe la siguiente expresión usando

   a) Alcance estático

   b) Alcance dinámico

   es necesario especificar cada una de las expresiones a evaluar con los respectivos valores.

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

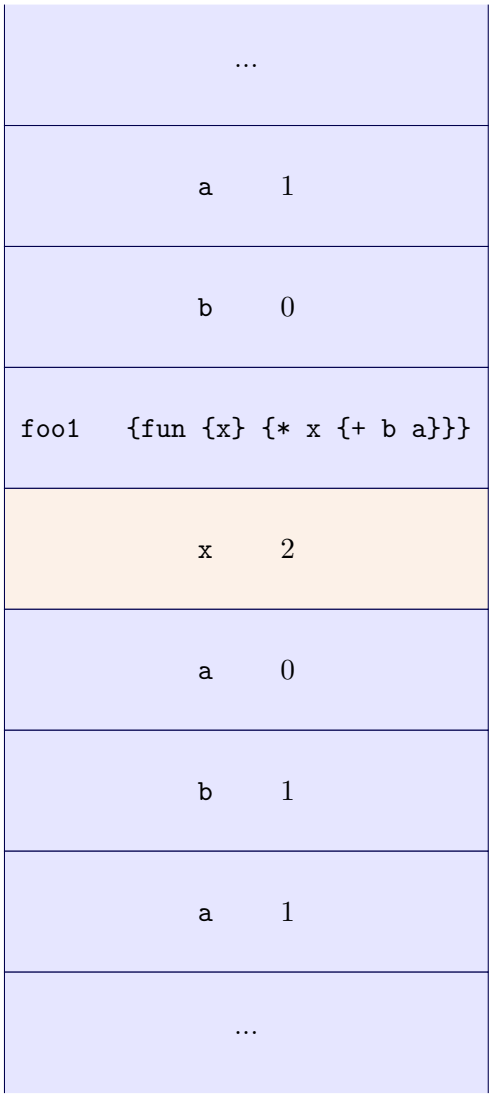
Solución:

a) Alcance estático

La expresión que debemos evaluar es `{foo1 2}`, por lo que

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

Además, el ambiente en forma de pila correspondiente es:



b) Alcance dinámico

La expresión que debemos evaluar es `{foo 2}`, por lo que

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

Además, el ambiente en forma de pila correspondiente es:

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

7. Escribe la definición de la función recursiva `agregaN` en *Racket*, que reciba tres parámetros: un elemento e, un número n y una lista l. La función debe regresar la lista resultante de agregar a la lista l, el elemento e en la posición n. No puedes utilizar ninguna función nativa del lenguaje, exceptuándo los operadores aritméticos `car`, `cons`, `append` e `equal?`.

```
;; agregaN: a exact list -> list

(agregaN 2 7 (list 1 2 3 4 5 6 7 8))
> '(1 2 3 4 5 6 7 2 8)
```

Solución:

```
        ;; agregaN: a exact list -> list
        (define (agregaN e n l)
          (cond
            [(< n 0) error 'agregaN "El indice debe ser un entero positivo."]
            [(= n 0) (cons e l)]
            [else (cons (car l) (agregaN e (- n 1) (cdr l)))]))

```