Lenguajes de Programación
Tarea I

Andrea Itzel González Vargas
Karla Esquivel Guzmán
Carlos Gerardo Acosta Hernández

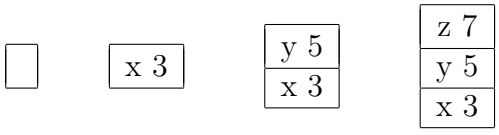
Entrega: 21 de Septiembre de 2015
Facultad de Ciencias UNAM

1. Problema I

a) Para ilustrar la no-linealidad de ambientes podemos utilizar el siguiente esquema de programa:

```
{with {x 3}
  {with {y 5}
    {with {z 7}
      {+ x {+ y z}}}}
```

A pesar de ser muy sencillo nos proporciona una idea de lo que ocurre en tiempo de ejeución. A partir de los siguientes diagramas representativos del stack:



Notemos que en la evaluación de la suma final del esquema, la ejecución no es lineal respecto al número de elementos (tres) de la entrada. Veamos por qué: Para obtener el valor del primer elemento de la suma, es decir x,

es necesario bajar en el stack hasta el primer ambiente, pasando entonces por tres ambientes, para obtener los valores de la segunda suma que corresponde al segundo sumando de la primera, para y es necesario pasar por dos ambientes y, finalmente para obtener el valor de z sólo pasamos por un ambiente que es el último agregado al stack.

De manera general, podemos describir este peor caso en la ejecución señalando que por cada uno de los n elementos, es necesario recorrer un número lineal de ambientes. Es decir, para el primero, se recorrerían n ambientes, para el segundo $n-1$, para el tercero $n-2$, y así sucesivamente hasta llegar al n-ésimo que recorrerá 1. Esta serie de pasos también podemos verla como la suma de los primeros naturales hasta n que pertenece al orden cuadrático, es decir como:

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} \in O(n^2) \tag{1}$$

De manera que la implementación de ambientes con un stack no es lineal sobre el número de elementos de la entrada.

b) Para mejorar la complejidad en tiempo de la implementación de ambientes, podemos utilizar en lugar de un stack, una *hash table*. Es importante definir las características de esta *hash table* especializada en ser el "repositorio de substituciones diferidas" (deferred substitutions repository, como se le llama en el Shriram).

En primer lugar, el id o nombre de la variable sería en este caso la llave, mientras que el valor asignado a ésta se agregaría en la "cubeta" o "casilla" que le corresponde a dicha llave.

En segundo lugar, dicha cubeta debe tener como estructura de datos afín, una pila, pues nos interesa sacar los elementos de las colisiones (que esperamos tener) en la operación de búsqueda para el momento de una substitución que debamos hacer en la evaluación. De tal suerte que dada la profundidad en el stack particular de la casilla de una llave, refiere al nivel de anidamiento en las substituciones subsecuentes, es decir, el ambiente al que pertenece. Entre más arriba se encuentre un valor en la pila de una casilla, pertenece a un ambiente más nuevo, una substitución de la variable más reciente.

c) Inicialmente, si no tenemos subsituciones que realizar, la tabla no tendrá elementos en su interior. Cada vez que en el esquema del programa

se encuentre una subsitución, en nuestro caso con la forma de un *with* o aplicación de función, se tomará el *bound-id* como llave y la *named-expr* como valor para realizar la operación de inserción en la *hash table* y continuamos la interpretación hasta que se exija alguno de los valores de las variables para alguna operación en la ejecución.

Finalmente, en la operación de substitución que implica una búsqueda en la tabla, para obtener el valor para una variable en la llamada a una aplicación de función o una operación que exija la substitución en tiempo de ejecución, es preciso tomar el elemento disponible de la pila de la casilla que le corresponde a la llave (variable) que se está solicitando, si hemos terminado de sustituir las variables que demanden los valores de ese ambiente particular, podemos eliminar dicho valor para preparar la estructura para otra posible operación que exija la substitución de la misma variable pero que ocurra en un ambiente superior (más viejo).

d) Gracias a esta estructura, podemos mejorar la operación de obtener el valor que le corresponde a una variable para la evaluación de n elementos, de tiempo lineal por cada elemento, a constante. Por tanto tendríamos como complejidad resultante un tiempo de orden $O(n)$. Veamos que en el peor caso del inciso a), se sigue conservando la complejidad, pues no es necesario pasar por un número lineal de substituciones para encontrar la solicitada, gracias al *hash table* es posible realizar esta operación en tiempo constante e igualmente en la obtención del valor no es necesario recorrer la pila, tan sólo con tomar el primero ($O(1)$), pues estamos en el ambiente más reciente y por tanto no es necesario bajar. De la misma manera, en cuanto hemos terminado con la sustitución para un ambiente particular es necesario borrar el valor del nivel correspondiente en el stack de una casilla, que siempre será el primero, lo cual no afecta la eficiencia de la substitución ($O(1)$). Tenemos entonces que para una entrada de n elementos, la substitución de cada uno de ellos puede realizarse en tiempo constante, por lo que gracias a la nueva estructura de substituciones diferidas, la complejidad es mejorada.

2. Problema II

a) No. A continuación el contraejemplo y el porqué del fallo en la estrategia de evaluación de Ben para algunos casos.

b) Es cierto que para este caso particular,

```
{with {x 4}
  {with {f {fun {y} {+ x y}}}
    {with {x 5}
      {f 10}}}}
```

la evaluación con alcance dinámico que propone Ben tiene el mismo resultado que con la evaluación de alcance estático, esto es porque la substitución más vieja en el stack que se considera en esta propuesta coincide con la substitución que considera el alcance estático. ¿Qué es lo que sucede si tenemos una substitución en un nivel más abajo que el primero en el stack original? Ben dice que tomemos la substitución más profunda de una variable en el stack, por lo que consideraríamos este, pero en el alcance estático no necesariamente coincide el valor que se considera para una variable con el valor que se le da en el ambiente más profundo. Haciendo una ligera modificación al ejemplo al caso particular en que la adecuación propuesta de Ben funciona, podemos proveer el contraejemplo del caso general que queremos contradecir.

```
{with {x 7}
   {with {x 4}
       {with {f {fun {y} {+ x y}}}
           {with {x 5}
              {f 10}}}}}
```

Con alcance estático, la evaluación resultante es (*num* 14) pues se toma el primer valor que se encuentre a partir de la función, es decir 4. En la propuesta de Ben, se tomará el último valor disponible para esa variable en el stack, en este ejemplo 7, por lo que resultará en algo distinto a la evaluación resultante con alcance estático, (*num* 17).

3. Problema III

a)
```
{with {5 {fun {x} {fun {y} {+ x y}}} 3}
   {with {10 {<:0 1> <:0 0>}}
      {<:0 1> {with {{+ 10 <:1 2>} {<:0 1> 0}}
                 {+ {+ <:0 1> <:0 0>} <:2 2>}}}}}
```

b)
```
interp({with {{x 5} {adder {fun {x} {fun {y} {+ x y}}}} {z 3}}
    {with {{y 10} {add5 {adder x}}}
        {add5 {with {{x {+ 10 z}} {y {add5 0}}}
            {+ {+ y x} z}}}}})

interp(subst(({with {{adder {fun {x} {fun {y} {+ x y}}}} {z 3}}
    {with {{y 10} {add5 {adder x}}}
        {add5 {with {{x {+ 10 z}} {y {add5 0}}}
            {+ {+ y x} z}}}}}) x (num (interp 5))))

interp(subst(({with {{adder {fun {x} {fun {y} {+ x y}}}} {z 3}}
    {with {{y 10} {add5 {adder x}}}
        {add5 {with {{x {+ 10 z}} {y {add5 0}}}
            {+ {+ y x} z}}}}}) x (num 5)))

interp({with {{adder {fun{x} {fun {y} {+ x y}}}} {z 3}}
    (subst({with {{y 10} {add5 {adder x}}}
        {add5 {with {{x {+ 10 z}} {y {add5 0}}}
            {+ {+ y x} z}}}} x (num 5))))})

interp({with {{adder {fun{x} {fun {y} {+ x y}}}} {z 3}}
    {with {{y 10} {add5 {adder x}}}
        (subst {add5 {with {{x {+ 10 z}} {y {add5 0}}}
            {+ {+ y x} z}}} x (num 5))}})

interp({with {{adder {fun{x} {fun {y} {+ x y}}}} {z 3}}
    {with {{y 10} {add5 {adder x}}}
        {add5 (subst {with {{x {+ 10 z}} {y {add5 0}}}
            {+ {+ y x} z}} x (num 5))}}})

interp({with {{adder {fun{x} {fun {y} {+ x y}}}} {z 3}}
    {with {{y 10} {add5 {adder x}}}
        {add5 (subst {with {{x {+ 10 z}} {y {add5 0}}}
            {+ {+ y x} z}} x (num 5))}}})

interp({with {{adder {fun{x} {fun {y} {+ x y}}}} {z 3}}
    {with {{y 10} {add5 {adder x}}}
        {add5 {with {{x (subst({+ 10 z} x (num 5)))} {y {add5 0}}}
```

```
            {+ {+ y x} z}}}}})

interp({with {{adder {fun{x} {fun {y} {+ x y}}}} {z 3}}
    {with {{y 10} {add5 {adder x}}}
        {add5 {with {{x {+ 10 z}} {y {add5 0}}}
            {+ {+ y x} z}}}}})

interp(subst({with {z 3}
    {with {{y 10} {add5 {adder x}}}
        {add5 {with {{x {+ 10 z}} {y {add5 0}}}
            {+ {+ y x} z}}}}} adder (num(interp{fun{x} {fun {y} {+ x y}}}))))
```