1

1. (1 pt.) Explica qué es un *idiom*. Da un ejemplo en el ámbito de la programación.

Un idiom es una convención o modismo que utilizan los programadores para solucionar determinadas tareas (a menudo cotidianas) haciendo uso de un lenguaje de programación, dichas convenciones generalmente no se encuentran especificadas en la sintaxis sino que son ampliamente adoptadas por la comunidad.

(2)

2. (2 pts.) Explica los 4 paradigmas de lenguajes de programación y da al menos 1 ejemplo de algún lenguaje de programación que pertenezca a cada uno de los paradigmas.

Lógico: Paradigma del tipo declarativo, se enfoca más en el "¿Qué?" del problema que con el "¿Cómo?"
Ejemplo: Prolog.

Orientado a Objetos: Propone modelar entidades del mundo real con un estado interno y capaces de interactuar entre ellas.
Ejemplo: C++

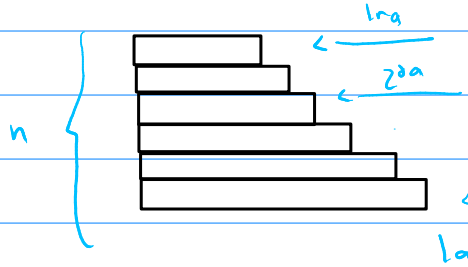
Funcional: Se centra como su nombre lo indica en tratar lo más posible como funciones puras y sin estado interno, se enfoca en la inmutabilidad de los datos
Ejemplo: Haskell

Estructurado: Funcionan principalmente por medio de de bloques de codigo o estructuras de control
Ejemplo: Fortran

D

3. (1 pt.) ¿Cuál es la complejidad del algoritmo de sustitución? Justifica tu respuesta usando algún ejemplo y explicándolo lo más detalladamente posible.

La complejidad es $O(n^2)$ pues podemos pensar que si tenemos n declaraciones, cada una seguida de la otra como en una pirámide



la primera se va a buscar sustituir en a lo más $n-1$ lugares (todos los de abajo) + 1 la inmediata definición siguiente

la segunda en a lo más $n-2$ +1 y si continuamos de esa forma veremos que el número total de sustitución es sería

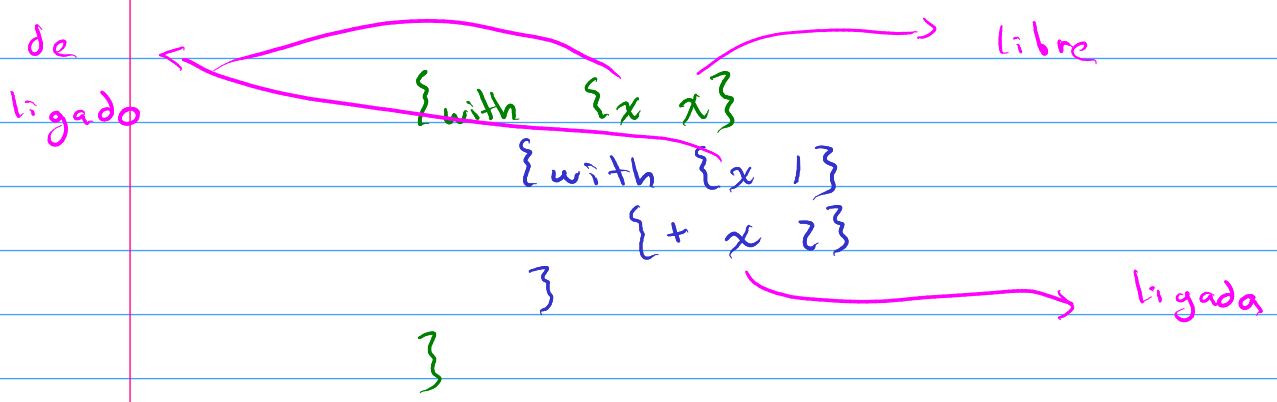
$n \to (n-1)+1$
$n-1 \to (n-2)+1$
$n-2 \to (n-3)+1$
$\vdots$
$1 \to 0+1$

$$k = 1 + \dots + (n-3) + (n-2) + (n-1) + n = \frac{n(n+1)}{2} \approx O(n^2)$$

□

3

4. (1 pt.) Da una expresión usando la gramática FWAE tal que un mismo nombre de identificador (supongamos x) aparezca en la misma expresión como identificador de ligado dos veces (con el mismo nombre de identificador), aparezca ligado al menos una vez y aparezca exactamente una única vez como identificador libre. Especifica a qué tipo de identificadores te refieres (de ligado, ligadas y libre) en tu expresión.



D

5. (1 pt.) Convierte el siguiente código usando índices de Bruijn o direcciones léxicas.

```
{with {x  -1}
  {with {y  1}
    {with {z  0}
      {with {w  {+  1  2}}
            {+  x   {+ y  {+  {+ z  w} {+  w  z}}}}}}}}
```

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

4

6. (1 pt.) Convierte el siguiente código con índices de Bruijn a código dentro de la gramática WAE. Las instancias de ligado se deben de nombrar como “x”, “y”, “z”, “w”, “v”, con respecto al orden de aparición de las mismas.

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

```
{with {x 1}
   {with {y 2}
      {with {z 3}
         {with {w {+ y z}}
            {with {v 4}
               {+ v {+ w {+ z {+ y x}}}}}}}}}
```

□

7. (1 pt.) A qué se le conoce como azúcar sintáctica en un lenguaje de programación.

Se le conoce así al proceso de simplicar la escritura de algunas instrucciones en nuestro lenguaje de modo que resulte más fácil para los desarrolladores escribirlo, aumentando en muchos casos la productividad, esta sintaxis es luego transformada o "desucarada" de vuelta a la sintaxis original del lenguaje.

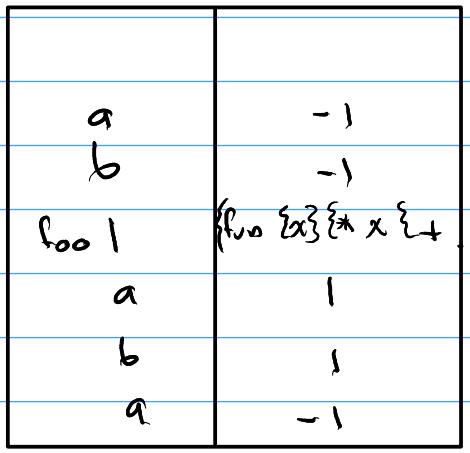
D

5

8. (4 pts.) Ponga el ambiente en forma de pila (stack) para la siguiente expresión, y evalúe la siguiente expresión usando

a) Alcance estático.

```
{with {a  -1}
 {with {b  1}
   {with {a  1}
     {with {foo1 {fun {x} {* x  {+  a  b}}}}
       {with {b  -1}
         {with {a  -1}
            {foo1  7}}}}}}}
```



> {foo1 7}
> {{ fun {x} {* x {+ a b}}} 7}

> {* 7 {+ a b}}
> {* 7 {+ 1 b}}

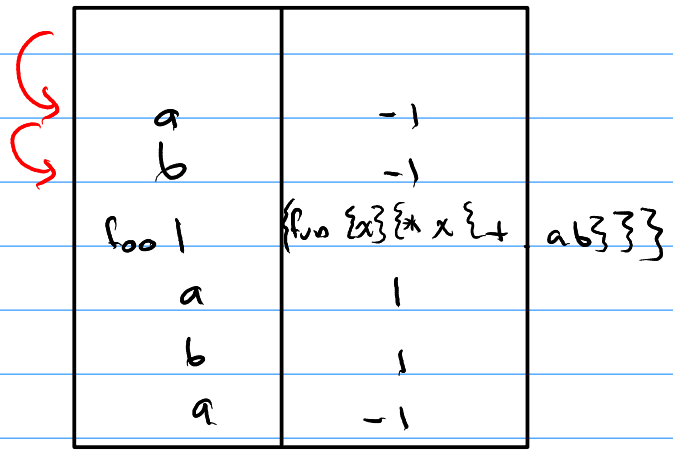
> {* 7 {+ 1 1}}
> {*7 2} = 14 ✗

□

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  1}
     {with {foo1 {fun {x} {* x  {+  a  b}}}}
       {with {b  -1}
         {with {a  -1}
            {foo1  7}}}}}}}
```



> {foo1 7}

> {
  {fun {x} {* x {+ a b}}}
  7
}
> {*7 {+ a b}}
> {*7 {+ -1 b}}
> {*7 {+ -1 -1}}
> {*7 -2}
> -14 ✗

□

6

9. (1 pt.) (a) Explica en un renglón, qué hace la siguiente función recursiva (no tienes que explicar línea por línea sino en general qué hace, (b) usa al menos 3 ejemplos para que primero verifiques qué hace la función (haz ejecuciones, al menos 3), y (c) dale un nombre mnemotécnico a ésta:

```
(define (secreto l n)
   (cond
       [(empty? l) empty]
       [(zero? n) l]
       [else (secreto (cdr l) (sub1 n))]))
```

a) Quita de la lista l los primeros n elementos, si es que los hay.

b) > (secreto '() 0)
> '()

> (secreto '(1) 0)
> '(1)

> (secreto '(1 2 3) 2)
(secreto '(2 3) 1)
(secreto '(2) 0)
'(2)

c) un nombre que me resultaría fácil recordar sería

"saca-elementos"

□