PRÁCTICA UNIDAD III

1. La siguiente sentencia de asignación en Java:

```
a = b + c
```

contiene los componentes:

- Identificadores: `a, b, c`
- Operadores: `=, +`

Para cada componente de la sentencia, enumera distintas ligaduras que se requieren para determinar el significado cuando se ejecuta la sentencia. Para cada ligadura, indica el tiempo en el que se produce en este lenguaje.

2. Considera las siguientes sentencias en ADA (a) y Python (b) respectivamente:

   a. `x := y`
   b. `x = y`

Para cada una de las mismas menciona ligadura/s que se produce/n al ejecutarse la misma, e indica qué entidades y atributos están vinculados.

Nota: tener en cuenta el concepto de declaración de variables en cada lenguaje.

3. Para cada código escrito en C que figura en el siguiente cuadro, indica el atributo de la variable `x` que se muestra con su ejecución:

Código A	Código B	Código C

`#include <stdio.h>`
`int main(void) {`
`  int x;`
`  printf("%d\n", x);`
`  return 0;`
`}`
	`#include <stdio.h>`
	`int main(void) {`
	`  int x;`
	`  x = 20;`
	`  printf("%d\n", x);`
	`  return 0;`
	`}`
		`#include <stdio.h>`
		`int main(void) {`
		`  int x;`
		`  x = 20;`
		`  printf("%u\n", &x);`
		`  return 0;`
		`}`

4. Una declaración de constante tiene típicamente la forma "`const I = E;`", asociando el identificador `I` al valor de la expresión `E`; el tipo de la constante está determinado por el tipo de `E`. Una forma alternativa es "`const T I = E;`", que declara explícitamente el tipo de la constante es `T`. En muchos lenguajes, `E` puede ser una expresión.
   a. Analiza comparativamente aspectos vinculados con las constantes nominadas en ADA, C, TypeScript y Python.
   b. ¿Cuáles son las ventajas que presenta el uso de constantes nominadas?

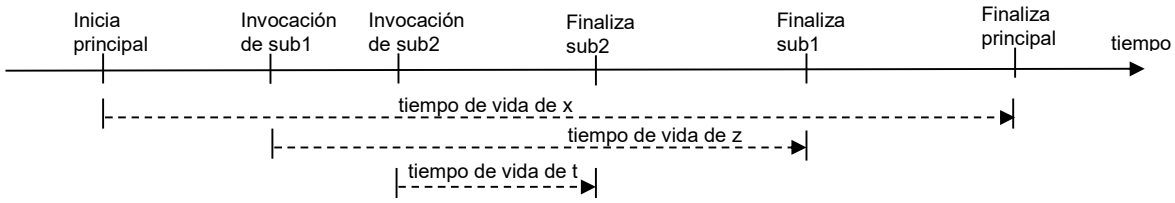
5. Considera el siguiente fragmento de código escrito con sintaxis similar a C:

```
int g;
void Q(){
   int z;
   …
}
void P(){
   float y1; int y2;
   … Q(); …
}
void main(){
   int x1; float x2;
   … P(); … Q(); …
}
```

Realiza un diagrama mostrando el tiempo de vida de las variables presentes en ese fragmento de código.

Nota: Para orientar la resolución de este tipo de ejercicio se brinda el siguiente ejemplo:

```
program principal
   var x: integer;
   procedure sub1
       var z: integer;
       procedure sub2
       var t: integer;
       begin {sub2}
       …
       end; {sub2}
   begin {sub1}
   sub2;
   end; {sub1}
begin {principal}
sub1;
end. {principal}
```



6. Dado el siguiente código escrito con sintaxis similar a C:

```
int fac(int n){
  int p, i;
  p = 1;
  for (i = 2; i <= n; i++)
    p = i;
  return p;
}
void main(){
  int f;
  f = fac(3);
}
```

   a. Realiza un esquema mostrando el tiempo de vida de las variables presentes en este programa.
   b. Sustituye la versión anterior de la función factorial por la siguiente versión recursiva:

```
int fac(int n){
  if (n > 1) return n * fac(n-1);
  else return 1;}
```

      i. Realiza un esquema mostrando el tiempo de vida de las variables presentes en el código modificado.

      ii. Considerando la invocación recursiva de `fac()`, qué relación puedes establecer entre alcance y tiempo de vida de su variable `n`.

7. Dado el siguiente código escrito en Java:

```
public class Example{
    static int n = 3;
    public static void main(String[] args) {
       int k;
       k = n;
       Controller control = new Controller(k);
       control.compute();
    }
}
class Controller{
    int n;
    Controller(int n){
       this.n = n;
```

```
        }
        void compute(){
           int i, m;
           int sum = 0;
           String message;
           Functor func;
           for(i = n; i > 0; i--){
              func = new Functor();
              m = func.calculate(i);
              sum = sum + m;
           }
        message = "The sum of factorials from 1 to " + n + " is " + sum;
        System.out.println(message);
        }
    }
    class Functor {
        int calculate(int n) {
        int temp;
        temp = n - 1;
        if (n > 0){
           return n * calculate(temp);
           } else {
           return 1;
           }

        }
    }
```

a. Enumera las variables estáticas, variables dinámicas de pila, variables dinámicas en heap explícitas y variables dinámicas en heap implícitas.

b. Relaciona el concepto de variable dinámica en heap con los principios de diseño de LP.

8. Responde:

   a. Menciona en qué situación:

      i. una misma variable puede estar asociada con diferentes direcciones de memoria en diferentes tiempos de ejecución de un programa.

      ii. múltiples variables pueden estar asociadas a una misma dirección de memoria durante la ejecución de un programa.

   b. Relaciona la situación planteada en el punto a)ii con los principios de diseño de LP. Fundamenta tu respuesta.

9. Dado el siguiente código escrito en C++:

```
#include <iostream>
using namespace std;

int m;
void sub(void){
  float m;
  m = 5;
  ::m = m + 2;
  cout<< m <<"\n";
  cout<< ::m <<"\n";
}
int main(){
  m = 2;
  cout<< m <<"\n";
  sub();
  cout<< m <<"\n";
  return 0;
}
```

a. Explica la salida que produce la ejecución de ese código (*será necesario que averigües la funcionalidad del operador ::*).
Nota: `cout<<` permite mostrar por pantalla.
b. ¿Cuál es el alcance de la variable `m` declarada de tipo entero?
c. Escribe un código similar en ADA utilizando el cualificador correspondiente para obtener el mismo efecto que el código brindado.

10. Indica cuáles son las palabras reservadas utilizadas en Typescript para declarar variables. Luego, explica cuáles son las reglas de alcance para cada una de ellas, y ejemplifica esas reglas de alcance con códigos breves.

11. Considera los siguientes códigos escritos en Python:

Código A	Código B

`def fun():`
`  c = 3`
`  print(b)`
 	
`a = 1`
`b = 2`
`fun()`
	`def fun():`
	`  c = 3`
	`  print(b)`
	`  b = 5 # (1)`
	
	`a = 1`
	`b = 2`
	`fun()`

a. El código A y el B difieren en la línea comentada con (1). Explica por qué el código B produce error al intentar ejecutarlo.
b. Reescribe el código B a fin de poder modificar desde fun() el valor de la variable `b` no local.

12. Dado el siguiente código escrito con sintaxis similar a C:

```
f0(){
  int a, b, c, d;
  f1(){
      int a, b, c, e;
      f2(){
       int a, b, f;
          ...
      }
      ...
  }
f3(){
  int b, c, d, g;
  ...
  }
...
}
```

a. ¿Cuál es el ambiente de referenciamiento para el cuerpo de código de `f0()`, `f1()`, `f(2)` y `f3()`, respectivamente, utilizando reglas de alcance estático?

b. Supongamos que `f0()` llama a `f1()`, `f1()` llama a `f2()`, y `f2()` llama a `f3()`, ¿cuál es el ambiente de referenciamiento del código de `f2()` después de llamar a `f3()`, utilizando reglas de alcance dinámico?

13. Considera el siguiente programa, escrito en un lenguaje nuevo y desconocido hasta ahora (es decir, creado para este ejercicio). Supongamos que el programa es correcto en el lenguaje dado, es decir, que no contiene errores sintácticos o semánticos, y que su ejecución no da lugar a un error de ejecución.

    Además, supón que la sentencia `print` de la función `g` imprimirá algo en la consola y que la ejecución comienza con la función `main`.

```
var a = "la respuesta a la vida, el universo y todo: ";
var b = 42;
function f(){
      var b = 11;
      g();
}
function g(){
      print(a + b);
}
function main(){
      var a = "esta va para: ";
      f();
}
```

¿Qué se imprimirá cuando se ejecute la función `g` en este programa? ¿Hay más de una interpretación razonable? Explica brevemente tu razonamiento.

14. Dado el siguiente código escrito con sintaxis similar a Java, ¿cuál es la salida del programa cuando se aplica cada una de las siguientes reglas de alcance? Explica tus respuestas.

```
public class A {
      int x = 0;
      void fa(){
            int x = 1;
            fb();
      }
      void fb(){
            println("x = " + x)
      }
      public static void main(String[] args){
            new A().fa();
      }
}
```

a. Alcance estático.
b. Alcance dinámico.

15. Ejecuta el siguiente código escrito en C y responde:

```
#include <stdio.h>
int errorCode ;
char * message;

void log () {
   printf ("Found error %d: %s\n", errorCode , message);
}

void checkPositive (int *array , int length ) {
   int errorCode = 0;
   char * message;
   int i;
   for(i = 0; i < length ; i++) {
         if(array [i] < 0) {
               errorCode = 20;
         }
   }
   if(errorCode != 0) {
         message = "Check positive failed ";
         log ();
   }
}

int main (void) {
   errorCode = 10;
   message = "File not found ";
```

```
    log ();
    int test [5] = {1,2,3,4,-1};
    checkPositive (test , 5);
    return 0;
}
```

a. ¿Qué tipo de alcance maneja el LP en función de los resultados obtenidos?
b. ¿Cuál sería la salida por pantalla si C tuviera el otro tipo de reglas de alcance?

16. Dada la siguiente estructura de un programa escrito en un LP con alcance estático:

```
program MAIN;
      procedure BIGSUB;
         procedure SUB1;
               procedure SUB4;
               ...
               end; { SUB4 }
         ...
         end; { SUB1 }
         procedure SUB2;
               procedure SUB3;
               ...
               end; { SUB3 }
         ...
         end; { SUB2 }
      ...
      end; { BIGSUB }
end. { MAIN }
```

Esquematiza la pila de ejecución y el contenido del display correspondiente:

a. Suponiendo que la secuencia de ejecución es:
   MAIN invoca a **BIGSUB**
   BIGSUB invoca a **SUB2**
   SUB2 invoca a **SUB3**
   SUB3 invoca a **SUB1**
b. Suponiendo que la secuencia de ejecución continua de la siguiente manera:
   SUB1 invoca a **SUB4**

17. Dado el siguiente esquema de un programa:

```
program
  var a, b;
  procedure P
    var a, x, c;
    procedure Q
      var x, y;
    begin
    ...
    x = c + b; → (1)
    ...
    end
    procedure R
      var d, b;
    begin
    Q
    end
  begin
  R
  end
begin
P
end.
```

a. Indica la profundidad estática de los distintos procedimientos y del programa principal.

b. Establece las referencias, según la cadena estática, para las variables no locales presentes en el punto **(1)** indicado en el código.
c. Considerando reglas de alcance dinámico, establece las referencias, según la estrategia de acceso profundo, para las variables no locales presentes en el punto **(1)** indicado en el código.

18. Para el siguiente código,

```
// main
int a;
f0(){
   f1(int b){
         f2( int c){int d; ...}
         ...
      }
   f3(int e, int f){int g, h; ...}
      ...
}
...
```

Considerando la siguiente secuencia de llamadas: el programa principal llama a `f0()`, `f0()` llama a `f1()`, `f1()` llama a `f2()`, y `f2()` llama a `f3()`.

a. ¿Cuál es la profundidad estática de cada función del programa?
b. Representa la pila de ejecución cuando se está ejecutando el código de f3(), incluyendo los enlaces dinámicos y estáticos.
c. ¿Cuál es el offset local de la variable local `h` en la instancia de registro de activación de `f3()`?
d. Representa el contenido del display.

19. Dado el siguiente código:

```
void main(){
int x, z;
void f(){
   int x = 3;
   g();
}
void g(){
   int z;
   z = 2;
   x = x + z;
   if (x < 5)  h(); else m();
}
void h(){
   printf("%d\n", x);
}
void m(){
   int x = z - 1;
   h();
}

z = 1;
x = 2;
f();
}
```

a. Muestra la salida del programa y la pila de ejecución, siguiendo la cadena estática.
b. Muestra la salida del programa y la pila de ejecución, siguiendo la cadena dinámica.

20. Dado el siguiente código:

```
Procedure Main;
   var x: integer;
   v: array[1..7] of integer;
```

```
    Procedure A;
          var f, x: integer;
          Procedure B;
          begin
            v(2):= v(2) + 3;
            v(5):= f + v(4) + x;
            C;
          end;
    begin
          x:= 2;
          f := 3;
          B;
    end;
    Procedure C;
          var i: integer;
    begin
          i:= x;
          v(i):= v(i) + i;
    end;
begin
    for x:= 1 to 7 do v(x):= 1;
    x:= 3;
    A;
    for x:= 1 to 7 do write(v(x));
end.
```

c. Muestra la salida del programa y la pila de ejecución, siguiendo la cadena estática.
d. Muestra la salida del programa y la pila de ejecución, siguiendo la cadena dinámica.

21. Considera el siguiente programa, escrito con una sintaxis similar a JavaScript:

```
// main program
var x, y, z;
function sub1() {
    var a, y, z;
    …
}
function sub2() {
    var a, b, z;
    …
}
function sub3() {
    var a, x, w;
    …
}
```

Resuelve las referencias de las variables no locales siguiendo la técnica de acceso superficial, para cada secuencia de invocación:

a. **main** invoca a **sub1**; **sub1** invoca a **sub2**; **sub2** invoca a **sub3**.
b. **main** invoca a **sub1**; **sub1** invoca a **sub3**.
c. **main** invoca a **sub2**; **sub2** invoca a **sub3**; **sub3** invoca a **sub1**.
d. **main** invoca a **sub3**; **sub3** invoca a **sub1**.
e. **main** invoca a **sub1**; **sub1** invoca a **sub3**; **sub3** invoca a **sub2**.
f. **main** invoca a **sub3**; **sub3** invoca a **sub2**; **sub2** invoca a **sub1**.