Preguntas Finales

Evolución de los lenguajes

1. Desde el punto de vista de los lenguajes de programación, ¿cuál sería el objetivo que conduce a la evolución histórica de los lenguajes?

   Los lenguajes de programación evolucionan con los siguientes objetivos:

   - Lograr independencia de la maquina, pudiendo utilizar un mismo programa en diferentes equipos con la única condición de disponer de un programa traductor o compilador, que es suministrado por el fabricante, para obtener el programa ejecutable en lenguaje binario de la maquina que se trate. Además, no se necesita conocer el hardware especifico de dicha maquina.
   - Aproximarse al lenguaje natural, para que el programa se pueda escribir y leer de una forma más sencilla.
   - Incluir rutinas de uso frecuente de manera que se puedan utilizar siempre que se quiera sin necesidad de programarlas cada vez.
   - Lograr la estandarización de los lenguajes.
   - Oportunidad: Cuando estandarizar? Cuando esta maduro, o sea cuando su uso va en aumento y antes de que aparezcan nuevas versiones.
   - Conformidad: adherir a un Standard (compilador, programa)
   - Obsolescencia: cuando pierde obsolencia un estándar, característica desaprobada.

2. Explique cual fue el aporte de ALGOL en la evolución de los lenguajes de programación. Explique dos características de lenguajes con ALGOL.

   El aporte que dio ALGOL fue que introdujo el concepto de tipo de datos e introduce la noción de estructuras de bloques y procedimientos recursivos.

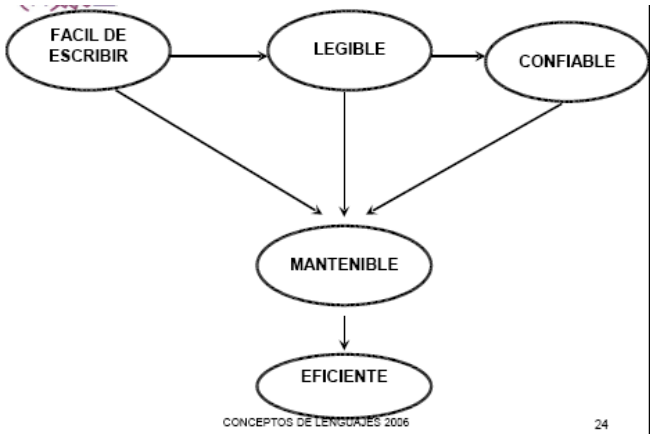
   Posee además tipos predefinidos, pero no admite tipos definidos por el usuario. Fue el primer lenguaje definido con gramática BNF. No tenia sentencias de E/S por tener independencia de la arquitectura.

3. Dentro de la evolución de los lenguajes, describa la importancia de FORTRAN y de ADA.

   Sobre FORTRAN podemos decir que fue el primer lenguaje de alto nivel y que junto a ALGOL 60 su objetivo fue definir herramientas para resolver problemas de cálculos numéricos. Incluyo modularidad mediante el desarrollo separado de subprogramas y permite compartir datos globales mediante la sentencia COMMON. No es independiente de la máquina.

   ADA hereda de Pascal, Simula67, suministro los conceptos de abstracción y encapsulamiento de tipos de datos y operaciones definidas por el programador. Introduce un extenso conjunto de características para el manejo de excepciones. Provee tareas capaces de ejecutarse concurrentemente. Cuenta con el tipo PACKAGE. Aporto la programación genérica, parametrizada y la modularización.

4. Defina y describa dos características deseables para un lenguaje de programación. **AGOSTO 08**



- Mantenibilidad: facilidad de introducir cambios, escribir una sola vez y luego reusar como por ejemplo las rutinas y módulos que favorecen la legibilidad y las modificaciones y lograr localidad, es decir, el efecto de una característica se restringe a una porción local del programa.
- Eficiencia: no solo medido por la velocidad de ejecución y por el espacio ocupado sino por el esfuerzo inicial de desarrollo y el necesario para su mantenimiento posterior. Un lenguaje además es eficiente si es optimizable, es decir, a la cualidad de permitir una optimización automática del programa.

Sintaxis y semántica

5. Sintaxis. Definición. Diferencias, formas de definirla y utilidad. **DICIEMBRE 08 – JUNIO 08**

La sintaxis es un conjunto de reglas que definen como componer letras, dígitos y otros caracteres para formar los programas. Se diferencia con la semántica ya que es un conjunto de reglas para dar significado a los programas sintácticamente válidos.

Las formas para definir la sintaxis pueden ser con el lenguaje natural (Ej.: Fortran), utilizando la gramática libre de contexto, definida por Backus y Naun: BNF(Ej: Algol) o con diagramas sintácticos que son equivalentes a BNF pero mucho mas intuitivos.

La utilidad de definir la sintaxis y la semántica de un lenguaje es determinar si un programa es válido y si lo es que significa.

6. Construya una gramática para describir la sintaxis de una estructura de control.

Se define a la sintaxis como un conjunto de reglas que definen como componer letras, dígitos y otros caracteres para formar los programas. Como la sintaxis es un conjunto infinito, se necesita una descripción finita. Existen diferentes formas para definir una sintaxis: Lenguaje natural: método no formal, utilizando la gramática: BNF o con diagramas sintácticos que son equivalentes a BNF pero más intuitivos.

La gramática se compone de un conjunto de reglas finitas que define un conjunto infinito de posibles sentencias válidas en el lenguaje. Una gramática esta formada por G = (N, T, S, P) donde N es un conjunto de símbolos no terminales, T es un conj de símbolos terminales, S es un símbolo distinguido de la gramática que pertenece a N y P son el conjunto de producciones.

Por ejemplo el if then else de ADA:

```
If (expresión lógica) then sentencias
        elsif sentencias
        elsif sentencias
else  sentencias
endif
```

T ={if, them else, elsif, endif, ;, 0..9, a..z, A..Z, and.., ==}

S = {<sent_cond>}

N = {<sent_cond>, <expresión_logica>, <sent_simple>, <bloque>, <grupo_else>, <grupo_elsif>, <operador>, <logico>, <relacional>, <id>, <letra>, <numero> … }

P = { <sent_cond>::= if <expresión_logica> then (<sent_simple>|<bloque>) [<grupo_else>| <grupo_elsif>] endif

<expresión_logica> ::= <id> {<operador> <expresión_logica>}*

<id> ::= (<letra> | <numero>)

<letra> ::= a|…|z|A|…|Z

<numero> ::= 0|..|9

<operador> ::= (<logico> | <relacional>)

<logico> ::= and|or|xor|not

<relacional> ::= ==|!=|<|>=|<=|>

< sent_simple > ::= <expresión_logica>|<for>|<while>

<bloque> ::= begin {<sent_simple>}$^{+}$ end

<grupo_else> ::= else (<sent_simple> | <bloque>)

<grupo_elsif> ::= elsif <expresión_logica> then (<sent_simple> | <bloque>)

}

7. **Compare la gramática BNF con el lenguaje natural como mecanismo de definición de la sintaxis de un lenguaje.**

8. Defina semántica de un lenguaje. Diferencie entre semántica estática y dinámica a través de al menos 2 ejemplos.

La semántica describe el significado de los símbolos, palabras y frases de un lenguaje ya sea lenguaje natural o lenguaje informático.

Sabiendo que la semántica estática es que se trata de comprobar en el momento de compilación y la semántica dinámica es la que se trata de comprobar en el momento de ejecución podemos ejemplificar de la siguiente manera:

```
Procedure Main;
        var y: integer;
        Procedure A;
        begin
                x:=x+1;
        end;
begin
        read (y);
        if (y>0) then A;
        write (y);
end;
end;
```

En este ejemplo tenemos un error semántico y se produce porque la variable x del procedure A no se encuentra declarada, dicho error se encontrarla en tiempo de compilación. No hay errores sintácticos.

```
#include <stdio.h>
main()
{
        int a, b;
        printf("\n Ingrese dos números:");
        scanf("%d%d", a, b);
        if (a=b) then
                printf("\n los números  ingresados son iguale
       else
       printf("\n los números ingresados son distintos");
}
```

En este ejemplo tenemos un error sintáctico en la función scanf ya que falta & antes de la variables a y b para indicar lo que se encuentra en esa posición de memoria. Además el then en la sentencia if no va.

Error semántico si a = b es una asignación, si a = 0 y b = 0, entonces a = b devuelve cero que es false en C. Por lo tanto entra por el else imprime que los números son distintos. La condición debería ser a==b. ESTO NO SE SI ES ASÍ. VERRRRRRRRRRRRRRRRRRRRRRRRRRR

9. Cite diferencias entre compilar e interpretar un programa.

Un intérprete lee el código fuente del programa línea a línea llevando a cabo cada vez las instrucciones específicas contenidas en la línea. A diferencia de un compilador lee un programa entero y lo convierte en código objeto que es la traducción del código fuente del programa a una forma que es directamente ejecutable por el computador. El código objeto se conoce también como código binario o código de máquina.

Los programas compilados se ejecutan mucho mas rápidamente que los interpretados desde el punto de vista del hardware aunque el proceso de compilación en si mismo lleva mas tiempo.

El intérprete ejecuta siguiendo un orden lógico de ejecución mientras que el compilador sigue el orden físico de las sentencias.

En cuanto al espacio ocupado el intérprete ocupa menos espacio. Cada sentencia se deja en la forma original; el compilador una sentencia ocupa cientos de sentencias de maquina.

La detección de errores es más difícil para el intérprete que para el compilador.

Semántica operacional – Ligadura - Variable

10. Desde el punto de vista de la semántica operacional cuando es indispensable usar la heap y porque?

    Es necesario tener una heap cuando existen variables cuyo tamaño sólo es conocido en tiempo de ejecución. **Ver que significa desde el punto de vista operaciones**

11. Compare las ligaduras estáticas con las dinámicas. De al menos 4 atributos de dos entidades. (Variable, Programa, Sentencia).

    Una ligadura es estática si se establece antes de la ejecución y no se puede cambiar.

    Una ligadura es dinámica si se establece en el momento de la ejecución y puede cambiarse de acuerdo a alguna regla específica del lenguaje.

    Por ejemplo la entidad variable tiene como atributos el nombre, tipo, área de memoria y la entidad rutina o procedimiento que tienen como atributos el nombre, código, los parámetros formales y los reales.

12. Defina los conceptos de alcance y tiempo de vida de una variable.

    El alcance es el rango de instrucciones en el que se conoce el nombre, es decir, todo el segmento de código en donde puede referenciarse la variable. Desde el momento de la creación de la variable, hasta el fin de la unidad que la contiene incluyendo los bloques anidados salvo que en alguno de ellos este definida de nuevo.

    El tiempo de vida es el lugar de memoria asociado con la variable, o sea, cuánto tiempo tiene esa variable asociado a ese espacio. Tiene que ver con la existencia.

13. Analice la ligadura de la variable con el tipo y el valor. ¿Es igual para todos los lenguajes?

    La ligadura de la variable con el tipo puede ser:

    - Estático: el tipo se liga en compilación
      - ✓ Explícito: mediante una declaración. Como Pascal , C y Ada
      - ✓ Implícito: se deduce por reglas. Como Fortran y Basic.
      - ✓ Inferencia: se deduce de los tipos de sus componentes. ML
    - Dinámico: la variable toma el tipo cuando se le asigna el valor. Como APL, Smalltalk y Snobol.

    La ligadura de la variable con el valor se presenta en tiempo de ejecución. Las constantes pueden ser manejadas de manera diferente si tenemos una sintaxis tan simple como la de Pascal, así que puede ser posible ligar estas en tiempo de compilación.

    La ligadura no es igual para todos los lenguajes. Cambia en compilación y en ejecución.

14. Construya ejemplos en los que se manifieste la ligadura estática y la ligadura dinámica de a menos dos atributos variables.

15. Desde el punto de vista operacional defina un mecanismo para manejar variables estáticas.

16. Qué significa que una variable estática sea sensible a la historia? Simule en pascal.

    Significa que su tiempo de vida excede el de su unidad.

17. Construya un ejemplo en Pascal que simule una variable estática.

    Una variable estática es una variable que mantiene su valor luego de la finalización del programa que la declaró. Es sensible a la historia, su tiempo de vida excede el de su unidad.

    Pascal lo simula con la definición de variables ABSOLUTE, ubicando la variable en una posición fija de memoria. Esto tiene como consecuencia que cada vez que se ejecute el programa, como la dirección será la misma, el valor tomado en la ejecución i+1 será el dejado por la ejecución i

(siempre y cuando otro programa no la haya modificado).

Para hacer esto en Pascal, es necesario, ante la ausencia de variables estáticas, utilizar una variable global y que esta se solo modificable por una sola función o procedimiento.

18. Describa a traves de un ejemplo, cuales son las diferencias entre una variable estática y una automática. **JUNIO 08**

19. Construya ejemplos en los que se aparecen variables estáticas y constantes y marque diferencias y similitudes.

20. Construya al menos 2 ejemplos en los que se generan alias e indique si pueden evitarse. **DICIEMBRE 08**

21. Alias. Construya dos ejemplos que los produzcan. Si tuviera que definir un nuevo lenguaje, como los evitaría.

Dos nombres comparten un objeto si sus caminos de acceso conducen al objeto. Un objeto compartido modificado vía un camino, se modifica para todos los caminos.
Por ejemplo

```
int x, y;
int *p;
…
p = &x ; // p apunta a x o contiene la dirección de x
y = *p; // y tiene el valor del objeto apuntado por p
```

Entonces a partir de estas instancias, *p y x son alias.

Ejemplo 1:

```
procedure swap(var x,y: integer){
    x := x+y;
    y := x-y;
    x := x-y;
}
```

entonces si lo invoco con la misma variable, siempre quedan en 0. X e y son alias

Ejemplo 2:

```
var p, q: prt int;
new(p)
q:= p;
```

Q y P son alias

Para evitar los alias eliminaría los punteros y sólo permitiría los pasajes por valor en los parámetros.

Hay dos caminos para eliminar el alias, uno consiste en eliminar los punteros, parámetros por referencia, datos globales y matrices. El otro camino consiste en poner restricciones al uso de tales características para excluir la posibilidad de alias.

En el caso de los parámetros por referencia el problema solo surgen si los parámetros reales se solapan, entonces si dichos parámetros son variables simples, es necesario asegurarse que sean distintas, de esta manera la llamada intercambio(a,a) sería ilegal. Estas formas de alias ilegales pueden detectarse en compilación, sin embargo la llamada intercambio(b[i], b[j]) genera alias solo si i = j, por lo tanto se podrían prohibir tales llamadas, pero el resultado sería un lenguaje torpe y difícil de usar. Sería conveniente que en tal caso el compilador genere la condición i distinto de j.

El manejo de alias en presencia de punteros es más complejo. La detección de alias ilegal entre dos punteros, causados por llamadas a procedimientos, es similar al caso de las matrices o vectores. La referenciación que exactamente igual que la indexación de una matriz. Por ejemplo, si p

y q apuntan a la misma colección y p^ y q^ se pasan ambos como parámetros, para que no haya solapamiento es necesario que se compruebe que p sea distinto de q.

Unidades – Rutinas - Parámetros

22. Enumere las formas de compartir datos entre unidades. Comparándolas entre si. **JUNIO 08**

Las formas de compartir datos entre unidades pueden ser:

- A través del acceso al *ambiente no local*: Esta forma la desventaja que tiene es que no permite una buena modularización. Cuando mas explicito es lo que se comparte mas se acerca al principio de diseño del software. Esto puede ser
  - ✓ Ambiente común explicito: áreas comunes con la desventaja que obliga al programador especificar que es lo que se comparte. Por ejemplo: COMMON de FORTRAN, Con uso de paquetes de ADA, Con variables externas de PL/1.
  - ✓ Ambiente no local implícito: es automático. Utilizando regla de alcance dinámico, utilizando regla de alcance estático.
- A través del uso de *parámetros*: es el más flexible, permite la transferencia de diferentes datos en cada llamada, proporciona ventajas en legibilidad y modificabilidad. Nos permite Determina que es exactamente lo que se comparte. Nos permiten compartir los datos en forma abstracta ya que indican con precisión qué es exactamente lo que se comparte

Los parámetros pueden ser *formales;* declarados en la especificación del subprograma y contienen los nombres y los tipos de los datos compartidos, en gral son similares a variables locales; o parámetros *reales*; son los que se codifican en la invocación del subprograma, pueden ser local a la unidad llamadora iser a su vez un parámetro formal de ella o un dato no local pero visible en dicha unidad o una expresión.

La evaluación de los parámetros reales es en general en el momento de la invocación primero se evalúa los parámetros reales, y luego se hace la ligadura antes de transferir el control a la unidad llamada.

La ligadura entre un parámetro real y uno formal puede ser posicional. Lo que quiere decir que el primer parámetro real corresponde al primero formal, el segundo al segundo y así sucesivamente. En el siguiente ejemplo se llama al procedimiento "Intercambia" haciendo corresponder el parámetro real X al formal A, y el real Y al formal B siendo la definición `procedure intercambia(X,Y: string) begin... end` y el llamado `intercambia(A, B)`.

Otros usan el método *variante*, una combinación con valores por defecto. Por ejemplo `int distance(int a=0, int b=0)` donde la invocación `call distance()` es lo mismo que `distance(0,0)`.

Y por último esta el método *por nombre*, en el cual se ligan por el nombre y deben conocerse los nombres de los formales y pueden ser colocados indistintamente en la lista. En Ada pueden mezclarse ambos métodos. En C++ y en Ada los parámetros formales pueden tener valores por defecto, con lo cual a veces no es necesario listarlos todos en la invocación.

23. Cómo esta compuesto el ambiente de referencia de una unidad?

El ambiente de referencia de una unidad está compuesto por:

- Ambiente local: objetos locales, ligados a los objetos almacenados en su registro de activación.
- Ambiente no local: objetos no locales.
- Alias
- Efecto lateral

Parámetros

24. Describa conceptualmente las diferencias y puntos de contacto entre el pasaje de parámetros por nombre y el pasaje de parámetros por procedimiento.

Las diferencias podemos decir que el momento de ligadura en el pasaje de parámetro por nombre la ligadura se difiere hasta el momento en que se la utiliza. En el pasaje por procedimiento, la ligadura se realiza en el momento de la invocación. El pasaje de parámetro por nombre es modo IN/OUT. En parámetro rutina depende del tipo de la rutina (procedimiento o función) y de los tipos de parámetros que estos tienen. Los parámetros por nombre utilizan los thunks y los parámetros rutina no. Y por ultimo podemos decir que en el pasaje por nombre se pasa la dirección de memoria del parámetro. En cambio en el caso de parámetros rutina se pasan los parámetros de la rutina, el ambiente de referencia y el encabezamiento de la rutina y una referencia al segmento de código del parámetro real.

Y los puntos de contacto en ambos casos se necesitan referencias a otras zonas de memoria, debido al uso de thunks y por las referencias al registro de activación y al segmento de código. Los dos realizan una llamada a un subprograma.

25. Ejemplifique dos casos de pasaje de parámetro. Uno en el que el efecto sea el mismo por valor-resultado y por referencia y otro que sea distinto.

Pascal: Normalmente pasa los parámetros a funciones y subrutinas internas por valor. Se puede pedir que los pase por referencia agregando var antes de los nombres de los parámetros.
Ejemplo:
Por valor: Function algo(x:Integer):Integer;
Por referencia: Function algo(var x:Integer):Integer;
En este caso tendrá el mismo resultado la llamada por valor-resultado que la llamada por referencia.

C/C++: Las distintas versiones de C pasan normalmente los parámetros a funciones y subrutinas internas por valor. Se puede pedir que los pase por referencia utilizando &. Otra forma de lograr lo mismo es usando punteros, para manejar la dirección de memoria de las variables directamente.
Ejemplo:
Por valor: int algo(int x);
Por referencia: int algo(int &x);

El cambio entre un sistema y otro radica en que si en el subprograma se llama cambia el valor del parámetro, en un caso vamos a notar el cambio de valor en el parámetro real y en otro no.

26. Construya un ejemplo que muestre la necesidad de retener mayor información en el pasaje de parámetros procedimientos.

Esto se refiere a la consistencia de tipos, por eso tenemos:

```
Proc ejemplo (i, j: int; proc f)
Var k:boolean;
Begin
        K := j < i
        If k -> f(k) else f(j)
End;
```

Así se pueden tener una o dos llamadas incorrectas al parámetro formal f, debido a la inconsistencia de tipos y/o diferencia en el numero de parámetros.

En ALGOL 68 en los procedimientos que se pasan por parámetro, se requiere que se especifiquen los tipos (llamados modos) de sus parámetros y sus valores devueltos. De esta manera puede verificarse la consistencia de tipos estáticamente.

```
Proc ejemplo (i, j:int; proc f (bool) void f ) void
Begin
        If i < j then f (k)
                else f (j)  ---- daría error
end
```

```
Int u,v;
A(){
   Int y;
   …
}
B(routine x ){
   Int u,v,y;
   C(){
      y = …;
   }
   x(); (*)
   B(C);
}
Main(){
   B(A);
}
```

Cuando A es invocado en (*), debe ejecutarse normalmente como si hubiera sido llamado directamente. En particular, la invocación de A debe poder acceder a su entorno no local, en este caso las variables u y v. Se debe notar que estas variables no son visibles en B porque están enmascaradas.
El llamado a una rutina se traduce en varias instrucciones que permiten reservar espacio para el registro de activación de la rutina llamada y setear su link estático. En el caso del llamado a x() esto no es posible porque no sabemos que es x. Esta información será conocida en tiempo de ejecución.
Podemos resolver este problema pasando el tamaño del registro de activación (AR) y el link estático requerido en el momento de la llamada (fp(d)).

27. Enumere al menos dos ventajas y dos desventajas del pasaje de parámetros por nombre contra el pasaje por parámetro por referencia.

Las ventajas que tiene el pasaje de parámetros por nombre contra el pasaje de parámetro por referencia es que tiene mayor flexibilidad ya que se establece la ligadura entre parámetro formal y parámetro real en el momento de la invocación pero la ligadura de valor se difiere hasta el momento en que se lo utiliza.

FALTA DESVENTAJAS

28. Ejemplo donde se pone de manifiesto la diferencia entre el pasaje por referencia y por nombre.

```
Procedure inter (x, y: integer)
Var
        Temp: integer;
Begin
        temp:=x;
        x:= y
        y:=temp;
End;
```

Si en inter los parámetros se pasan por referencia se realiza un intercambio de variables correctamente ya que se intercambian los valores de las referencias de x e y.

En cambio si los parámetros son pasados por nombre ante la invocación a inter (i, a(i)) la ejecución quedaría:

Temp:=i;

1) i := a(i);

2) a(i) ;= temp;

pero aquí a(i) de 1) no será el mismo que a(i) de 2), pues como el valor de i ya fue modificado queda la posición del arreglo sin modificar y se modifica otra. Luego entonces no se intercambian los valores. Esto se da pues los parámetros pasados son un índice y un elemento de un arreglo.

29. ¿Cree que existe algún punto de contacto entre el pasaje de parámetros por procedimientos y el pasaje por parámetros por nombre?

    Si, los dos conducen a programas difíciles de leer cuando contienen la existencia de variables que pertenecen al entorno de referencia y no al registro de activación. Los ejemplos los hice en papel. (manda el papel).

30. Explicite las ventajas de que un lenguaje de programación permita parámetros procedimientos.

31. Compare el pasaje de parámetros por nombre con otras formas de pasaje de parámetros.

32. Es equivalente el resultado de pasar parámetros por valor/resultado o pasarlos por referencia. Justifique.

33. Enumere las formas de pasaje de parametro dato e indique que lo haria elegir cada una de ellas. **AGOSTO 08**

Sistema de Tipos – Tipo de datos

34. Que es un sistema de tipos y para que sirve?

    Un sistema de tipos es un conjunto de reglas usadas por un lenguaje para estructurar y organizar sus tipos. Las operaciones definidas para un tipo son la única forma de manipular sus instancias (crearlas, destruirlas modificarlas), ellas protegen a los objetos de usos ilegales.

    Si dice que un programa es seguro en cuanto a los tipos si garantiza no tener errores de tipos.

    Su función es capturar la naturaleza de los datos del problema que serán manipulados por los programas. Componente semántica importante.

35. Que encierra el concepto de 'tipo de datos'?

    El tipo de datos es el conjunto de valores que puede tomar durante el programa.

    Si se le intenta dar un valor fuera del conjunto se producirá un error. La asignación de tipos a los datos tiene 2 objetivos principales; detectar errores en las operaciones y determinar como ejecutar esas operaciones.

36. Que es un subtipo? Lo implementan ADA y Pascal?

37. En todos los lenguajes ¿Una variable se liga estáticamente con su tipo? En caso de respuesta negativa, de al menos un ejemplo.

    No en todos los lenguajes una variable se liga estáticamente con su tipo sino que se le asigna el tipo con una sentencia de asignación. Se liga al tipo del valor de la parte derecha de la asignación. Los lenguajes que tienen este tipo de ligadura son por ejemplo APL, Smalltalk.

    Por ejemplo en APL se puede indicar:

    valores <- 3.5 8.3 0.7 10.0 (en donde el tipo de la variable de nombre valores seria una lista de reales de longitud 4)

    valores <- 15 (el tipo de la variable es entera)

38. Que significa que un lenguaje sea fuertemente tipado?

    Este concepto está estrechamente relacionado con el de si un lenguaje tiene "Verificación estática" o "Verificación dinámica" de los tipos de datos.

Si se puede realizar una verificación estática de todos los errores de tipo de un programa, se dice que el lenguaje es fuertemente tipado.

Un lenguaje fuertemente tipado es aquel donde toda declaración de variables o tipos de datos, necesariamente debe tener declarado de manera explícita de que tipo son dichas variables y tipos de datos.

Aquellos lenguajes que no son fuertemente tipados posee lo que se denomina “Inferencia de Tipos”, es decir que infier cualquier información faltante en cuanto a tipos a partir de otros tipos declarados.

Los lenguajes fuertemente tipados son Java, Pascal o C. Un lenguaje débilmente tipado es Visual Basic (Basic). En VB se permite concatenar la cadena '12' con el entero 3 y después tratar el conjunto como un entero sin conversión de tipos.

39. ¿Es equivalente decir que un lenguaje es fuertemente tipado a decir que realiza verificación estática de tipos?

    Si porque si el lenguaje es fuertemente tipado el compilador puede garantizar la ausencia de errores de tipo del programa.

40. ¿Se puede decir que Pascal es fuertemente tipado?

    Si, Pascal es fuertemente tipado, esto significa que todos los datos utilizados deben tener sus tipos declarados explícitamente y el lenguaje limita la mezcla de tipos en las expresiones.

41. Describa la evolución de los tipos de datos, hasta llegar al paradigma de objetos.

    Evolución de los tipos de datos:

    - ASSEMBLER: Los lenguajes de máquina veían a los datos almacenados como cadenas de bits que podían ser manipulados por un conjunto de instrucciones de máquina.
    - FORTRAN, COBOL y ALGOL 60: Dieron un paso hacia la primera abstracción en los datos, ya que los datos no se ven como una secuencia de bits sino como un valor entero, real, lógico, etc... Existía un conjunto fijo de abstracciones.
    - Algol 68 y Pascal: Tratan de alcanzar la generalidad creando mecanismos para crear abstracciones. Estas abstracciones hasta pueden considerarse como parte del lenguaje. Ejemplos de esto son los arreglos, registros, etc.
    - Simula 67: Proporciona una estructura (class) que la representación y las operaciones concretas puedan especificarse en una sola unidad sintáctica. Mejora altamente la legibilidad. Es el primer acercamiento hacia los TAD.
    - ADA: Permite generar estructuras en la cual se cumplen los requerimientos de un TAD:
        - ✓ Representación asociada a sus operaciones. Lo tenía simula.
        - ✓ Ocultamiento de la representación del tipo definido.

    La diferencia con simula es que este sólo cumplía la primer característica y no la segunda.

Podemos plantear que los TAD son el mecanismo sobre el cual funciona el paradigma de objetos, donde cada clase es un tad. Y los mensajes son las operaciones definidas sobre el TAD que manipulan su representación interna. En esta caso, el estado del objeto.

42. Diferencias y similitudes entre un tipo de dato predefinido, un tipo creado por el usuario y un tipo de datos abstracto (TAD).

    Las diferencias pueden ser que un tipo predefinido indica cuales son las operaciones que se le pueden aplicar y el programador se limita a ello como así también a su comportamiento. En un tipo definido por el usuario se pueden definir nuevas operaciones de acuerdo al problema. Y en un tipo de dato abstracto pasa como un predefinido.

    El TAD provee ocultamiento de la información a diferencia de los otros tipos de datos.

    Y las similitudes son que ocultan la especificación, proveen legibilidad.

43. Compare los tipos predefinidos con los definidos por el usuario.

    Un tipo predefinido pueden ser vistos como un mecanismo para clasificar los datos manipulados por un lenguaje. Son la forma de proteger a los datos contra usos indebidos. Instanciar

un tipo para crear un dato indica que operaciones se le pueden realizar legalmente. Por ejemplo, booleanos, caracteres, etc.

Las ventajas de los tipos predefinidos son:

- Invisibilidad de la representación: no se tiene acceso a la cadena de bit que representa a un valor de un cierto tipo. Solo se puede cambiar la cadena mediante las operaciones legales sobre ese tipo y lo que resulta es un nuevo valor del tipo predefinido, es decir, el programador no ve la modificación como una cadena de bits. Esto incrementa la legibilidad del programa y permite la portabilidad de los programas porque se puede cambiar la representación de las abstracciones sin afectar a los programas que las utilizan.
- Verificación estática: el compilador puede detectar operaciones ilegales sobre una variable.
- Desambiguar operadores en tiempo de compilación: si el tipo de cada variable es conocido en tiempo de compilación, el binding entre un operador sobrecargado y su correspondiente operación puede ser establecido en tiempo de compilación.
- Control de precisión: algunos lenguajes permiten controlar la precisión de los datos numéricos. Por ejemplo en ADA con los enteros: shortinteger, integer , large integer

En general los tipos predefinidos son elementales (sus valores son atómicos), pero hay excepciones como por ejemplo ADA tiene los strings como tipos predefinidos y sin embargo son arreglos de caracteres.

Los tipos definidos por el usuario permiten definir nuevos tipos e instanciarlos. Separa la especificación de la implementación. Se definen los tipos que el problema necesita. La instanciación de los objetos en un tipo dado implica una descripción abstracta de sus valores. Los detalles de la implementación solo quedan en la definición del tipo. Las ventajas son:

- Legibilidad: debido a la elección apropiada de los nuevos nombres de tipo.
- Modificabilidad: un cambio en las estructuras de datos que representan las variables de un tipo dado requiere cambios únicamente en la declaración de tipo, no en las declaraciones de todas las variables.
- Factorización: la definición de un tipo de dato complicado se escribe una sola vez y después se puede utilizar tantas veces como sea necesario.

44. Construya al menos dos ejemplos en los que se manifieste la utilidad de poder realizar verificación estática de tipos.

45. Cuáles son los beneficios y los inconvenientes que puede presentarse al elegir un tipo de datos Unión Discriminada? **JUNIO 08**

Como beneficio esta técnica permite ahorrar memoria, pues cada variante agrega un alias al mismo lugar en memoria sin embargo se corre el peligro de generar errores de interpretación. Tiene un chequeo dinámico lo que lo hace más flexible para programar. Permite manipular diferentes tipos en distinto momento de la ejecución.

Como desventaja el programador es el responsable de asegurarse que en una unión los datos están referenciados con el tipo de dato apropiado. Lo que hace que sea una estructura insegura. Además permiten una forma de alias, ya que los campos de las diferentes variantes son todos “alias” de las mismas posiciones de memoria.

46. Evalúe la implementación de la unión discriminada de Pascal y de Algol.

La unión discriminada es un tipo estructurado de dato, un tipo de dato definido por el usuario compuesto. Permite, en cualquier instante de la ejecución del código al que pertenece, la elección de una entre diferentes estructuras alternativas, cada una de las cuales se denomina **variante** dependiendo del valor que presenta un selector conocido como **discriminante**.

Una unión es un tipo derivado cuyos miembros pueden ser de cualquier tipo y comparten el mismo espacio de memoria. Esto permite que una variable de este tipo, puede contener en distintos momentos, objetos de diferentes tipos y tamaños. Las uniones deben contener dos o más miembros, y únicamente uno de estos puede ser referenciado en un momento dado. El programador es el

responsable de asegurarse que en una unión los datos están referenciados con el tipo de dato apropiado. Lo que hace que sea una estructura insegura.

En **Pascal**, la implementación de la unión discriminada es a través de registro con variante: Un tipo registro de pascal puede tener una parte variante, en cuyo caso, es posible definir uniones discriminadas.

```
TYPE
      mes = (enero, febrero, marzo, abril, mayo, junio, julio, agosto,
septiembre, octubre, noviembre, diciembre);
      articulo = record
                  precio: real;
                  case disponible: boolean of
                    true: (cantidad: integer;  descuento: real);
                    false: (mes_esperado: mes);
       end;
end;
```

El identificador de campo `disponible` es el componente discrimínate. Pascal permite, con la notación de punto, acceder a todos los campos de registro y manipularlos.

La comprobación de tipos debe hacerse en tiempo de ejecución porque depende del valor del discriminante. Esta comprobación requiere llevar un descriptor en tiempo de ejecución para cada registro con variante.

Los registros variantes ahorran memoria, porque como sólo una de las estructuras que están dentro de ellos puede usarse a la vez, el compilador sólo necesita reservar memoria para la mayor de ellas (en vez de tener que hacerlo para todos los componentes del registro).

Son muy eficientes y seguros, porque sólo permiten al programador realizar operaciones que tengan algún significado dentro del contexto dado y que sean permisibles dentro del registro.

Sin embargo, el problema con esta estructura de datos es que no requiere ser inicializada. Esto significa que podríamos tener cualquier valor dentro de un registro variante después de modificar el valor de un campo identificador. Este problema se produce porque Pascal (al igual que la mayoría de los lenguajes estructurados) no requiere que se inicialicen las variables al entrar a su entorno. El acceso a variables no inicializadas puede dar lugar a que se usen valores que fueron dejados previamente en el bloque y que compartían la misma zona de memoria. En el caso de los registros variantes, este valor puede incluso ser de un tipo diferente al esperado.

La raíz del problema es que los registros variantes permiten una forma de “aliasing”, ya que los campos de las diferentes variantes son todos “alias” de las mismas posiciones de memoria.

Algol permite uniones discriminadas seguras en forma de modos unidos. Por ejemplo `mode ent_bool = union(int, bool)` define un modo nuevo cuyos valores pueden ser enteros o lógicos. Si tenemos `ent_bool x` la variable declarada en todo momento contiene un valor de tipo `int` o de tipo `bool`, pero el tipo de `x` es siempre `ref ent_log`.

Las asignaciones a una variable x de tipo ent_log son igual que las demás sentencias de asignación:

```
ent_bool x, y;
x := 5;
x := y;
x := true;
```

Sin embargo, el acceso al valor de x no se puede hacer tan fácilmente debido a que el tipo de valor se debe establecer entes de poder utilizar el valor. La determinación del tipo se hace con una "clausula de conformidad". De esta forma se evita la incompatibilidad de tipos. Por ejemplo

```
case x in
     (int x1): ... x1 ...
     (bool x2): ... x2 ...
esac
```

Este ejemplo de tal clausula contiene dentro de cada alternativa de la cláusula **`case`** se establece el tipo de `x` y se utiliza un nombre nuevo para referirse a `x`. Es decir, `x1` y `x2` son constantes inicializadas con el valor de `x` a la entrada de la clausula de conformidad; sus tipos son

`int` y `bool` respectivamente y por lo tanto la comprobación de tipos en el uso de `x1` y `x2` se puede hacer estáticamente.

47. Construya un ejemplo en que se ponga de manifiesto las inseguridades de Pascal en el manejo de unión discriminada.

Pascal realiza un manejo de la unión discriminada a través del uso de registro con variantes. Sea el registro:

```
TYPE
      departamento = (domestico, deportes, droguería, alimentacion);
      mes = 1..12;
      articulo = record
                  precio: real;
                  case disponible: boolean of
                        true: (cantidad: integer;  donde: departamento);
                        false: (mes_esperado: mes);
       end;
end;
```

Ya que la comprobación de tipo para los registros con variante solo puede hacerse en tiempo de ejecución, se podría tener:

```
Var a= articulo
a.cantidad
```

Pero es una selección de campo correcta solo si la variante a tiene un valor true en su campo indicador. La comprobación dinámica de tipo requiere llevar el control de la variante actual en tiempo de ejecución para cada registro variante. Desgraciadamente la posibilidad de modificar independientemente el campo indicador y las variantes hace difícil implementar la comprobación en tiempo de ejecución.

Aún si el registro no tiene un campo indicador (ya que es opcional) Pascal deja en manos del programador la comprobación de la variante actual de un registro, lo que produce una programación peligrosa y códigos difíciles de leer y escribir.

48. Ejemplo que muestra la ventaja de ALGOL 68 sobre PASCAL en el manejo de registros variantes

En ALGOL tenemos

```
union (int, real) v1, v2;
int cont;
real suma;

v1 := 33
… código intermedio…
cont := v1;        *
…
case v1 in
        int valint:cont := valint;
        real valreal: suma := valreal;          (*)
end
```

valint y valreal son constantes inicializada con el valor de v1.

En * no podemos determinar estáticamente si v1 quedó entera o no luego del código intermedio. Entonces ALGOL usa una clausula de conformidad que es un CASE cambiando cada asignación donde aparece v1 por (*) que debe ser realizada por el programador.

En cambio Pascal realiza un manejo de la unión discriminada a través del uso de registro con variantes. Sea el registro:

```
TYPE
```

```
        departamento = (domestico, deportes, droguería, alimentacion);
        mes = 1..12;
        articulo = record
                    precio: real;
                    case disponible: boolean of
                          true: (cantidad: integer;  donde: departamento);
                          false: (mes_esperado: mes);
         end;
end;
```

Ya que la comprobación de tipo para los registros con variante solo puede hacerse en tiempo de ejecución, se podría tener:

```
Var a= articulo
a.cantidad
```

Pero es una selección de campo correcta solo si la variante a tiene un valor true en su campo indicador. La comprobación dinámica de tipo requiere llevar el control de la variante actual en tiempo de ejecución para cada registro variante. Desgraciadamente la posibilidad de modificar independientemente el campo indicador y las variantes hace difícil implementar la comprobación en tiempo de ejecución.

Aún si el registro no tiene un campo indicador (ya que es opcional) Pascal deja en manos del programador la comprobación de la variante actual de un registro, lo que produce una programación peligrosa y códigos difíciles de leer y escribir.

49. Muestre a traves de ejemplos cual es la diferencia semántica de los arreglos definidos como datos semidinámicos a los definidos como datos dinámicos. **AGOSTO 08**
50.
51. Muestre en un ejemplo la diferencia de implementacion de un arreglo con limites estático o uno con limites dinámicos. **DICIEMBRE 08**

52. Punteros. Necesidad y perjuicios.

Un puntero es una referencia a un objeto. Una variable puntero es una variable cuyo r-valor es una referencia a un objeto.

La necesidad de usarlo es que permite la implementación de tipos de datos recursivos. Un tipo de dato recursivo T se define como una estructura que puede contener componentes del tipo T como por ejemplo el tipo Lista y del cual surgen el tipo de lista Pila y Cola.

Los perjuicios es que hacen que los programas sean menos comprensibles y frecuentemente inseguros debido a:

- Violación de tipos debido a que los punteros no están cualificados por el tipo del objeto al que pueden apuntar. Por ejemplo, los punteros en PL/1 se declaran simplemente como punteros. Las variables declaradas como `BASED` se acceden solo a través de punteros. Por ejemplo en las siguientes declaraciones:

```
p POINTER
x int BASED
y float BASED
```

Se hace apuntar un puntero a una variable basada cuando se asigna la variable expresamente como en `ALLOCATE x SET p.`, es decir, `P` → `x`. Sin embargo, como `p` no esta cualificado para apuntar únicamente a enteros entonces se puede además intentar acceder a `y` a través del mismo puntero `p, P` → `y.`

En tiempo de compilación (estatica) es imposible garantizar que el puntero que se va a suministrar va a apuntar una variable del tipo correcto. Por esto el compilador asupone que el acceso es correcto y esto puede conducir a errores. La solución a esto es ligar los punteros a tipos como lo hace C. Por ejemplo:

```
int x = 10;
float y = 3.7;
int* p = &x; /* p apunta a x o p tiene la dirección de x */
p ++;  /* p apunta a la siguiente dirección, es decir que
          contiene un valor float */
*p += x;  /* incrementa y, interpreta como un int con 10*/
printf("%f", y); /* toma el valor resultante como float */
```

- Referencias sueltas – referencias dangling: si este objeto no esta alocado se dice que el puntero es peligroso. Una referencia suelta o dangling es un puntero que contiene una dirección de una variable dinámica que fue desalocada. Si luego se usa el puntero producirá error. Por ejemplo

```
int* px;
void trouble()
{
      int x; /* aloca x */
      …
      px = &x; /* asigna la dirección de x a la variable global px */
      …
      Return;     /* desaloca x */

}
main ()
{
      …
      trouble();
      …
}
```

El tiempo de vida y el alcance de la variable px es mayor que el de la variable apuntada.

- Liberación de memoria: objetos perdidos. los objetos (apuntados) que se alocan a través de la primitiva new son alocados en la heap. La memoria disponible (heap) puede agotarse. Si los objetos en la heap dejan de ser accesibles (objeto perdido) esa memoria podría liberarse. Esta liberación de la memoria se podría realizar de 2 maneras:
  - ✓ Explicita: como en Pascal con dispose() o en C++ delete(). No garantiza que no haya otro puntero que apunte a esa dirección definida como basura, este puntero se transforma en dangling (puntero suelto). El reconocimiento de la basura recae en el programador.
  - ✓ Implícita: el sistema dinámicamente, tomará la decisión de descubrir la basura por medio de una algoritmo de recolección de basura: garbage collector. Se ejecuta durante el procesamiento de las aplicaciones. Lo usa LISP, ADA, Java, Algol 68

- Punteros no inicializados: puede suceder que a través de ellos se accedan descontroladamente a posiciones de memoria. Para hacer dinámica la verificación los punteros deben inicializarse con valores especiales. Por ejemplo en Pascal es nil, en Ada y Java es null y void en C y C++.

- Punteros y uniones discriminadas: los punteros dentro de uniones pueden causar inseguridades si permiten la modificación independiente de los campos. Por ejemplo en C:

```
union trouble{
      int int_var;
      int *int_ref;
}
```

En el caso de C, este es el mismo efecto que causa la aritmética de punteros. Para resolver este problema asociado con los punteros Java elimina la noción de puntero explicito completamente.

- Alias:

```
int* p1
int* p2
int x
p1 = &x     p1 y p2 son alias
p2 = &x     p1 y x tambien lo son
            p2 y x tambien lo son
```

53. Punteros. Problemas y soluciones.

Los problemas son los mencionados en la pregunta anterior y las soluciones son:

- Violación de tipos: ligar los punteros a tipos.
- Referencias sueltas: la solución es reasignar una dirección al puntero antes de desreferenciarlo, norma general que también soluciona el problema de los punteros nulo.
- Punteros no inicializados: la solución seria asignarla al puntero un valor especial nulo.

54. Muestre en un ejemplo el peligro del uso con punteros.

Hay que mostrar un ejemplo en el cual se pude acceder a una variable diferente o perder el rango de un array.

55. Construya dos ejemplos donde ponga de manifiesto dos de los problemas que puede originar el uso de punteros. Proponga una solución.

56. ¿Es indispensable para que un lenguaje maneje recursión que la ejecución se maneje como un esquema de pila?

57. ¿Qué necesita un lenguaje para poder asegurar soporte para TADs? Hay alguna otra característica que favorezca el uso de Tads. Ejemplos y justificación de los lenguajes que los soportan y los que no.

A partir de Simula 67, Algol68 y Pascar permitieron al programador en diferente grado unos de otros definir un tipo abstracto de dato proporcionando construcciones especiales en el propio lenguaje.

Un TAD son tipos definidos por el usuario y se compone de un conjunto de *propiedades* con ciertas características comunes y un conjunto de *operaciones* para manipular dichos elementos. Un tipo de datos es abstracto (TAD) cuando se determinan las operaciones que manipularán los elementos sin decir cual será la forma exacta de éstos o aquellos. Disponer de un TAD nos proporciona encapsulamiento, reusabilidad y abstracción.

Un lenguaje necesita para definir Tads lo siguiente:

- *Encapsulamiento*: es decir que el usuario del nuevo tipo no pueda manipular los objetos de datos del tipo, excepto por el uso de las operaciones definidas.
- Separación de declaración e implementación
- Ocultamiento de información

Lo que debe proveer un lenguaje es el *encapsulamiento* de una definición de tipo de datos abstractos para imposibilitar ese acceso de forma extensa y que los únicos subprogramas que sepan como están representados los objetos de datos de tipo sean las operaciones definidas como parte del mismo tipo.

Con la aparición de los lenguajes de programación estructurados en la década de los 60, surge el concepto de tipo de datos (ing., data type), definido como un conjunto de valores que sirve de dominio de ciertas operaciones. En estos lenguajes (C, Pascal), los tipos de datos sirven sobre todo para clasificar los objetos de los programas (variables, parámetros y constantes) y determinar qué valores pueden tomar y qué operaciones se les pueden aplicar.

Esto era insuficiente en el desarrollo de software a gran escala, dado que el uso de los datos dentro de los programas no conocía más restricciones que las impuestas por el compilador, lo que era muy inconveniente en los nuevos tipos de datos definidos por el usuario, sobre todo porque no se restringía de ninguna manera su ámbito de manipulación. Para solucionar esta carencia, a mediados de la década de los 70 surge el concepto de tipo abstracto de que considera un tipo de datos no sólo como el conjunto de valores que lo caracteriza sino también como las operaciones que sobre él se pueden aplicar, juntamente con las diversas propiedades que determinan inequívocamente su comportamiento. Es por esto que ya los lenguajes como ADA, Clu, C++, Simula 67 permiten definir TAD.

El mecanismo que brinda **ADA** para definir TADs es a través de packages. Mediante los packages se logra separar la definición del tipo y sus operaciones de la implementación de dichas

operaciones, ya que se hacen en archivos separados. Estos package constan de la especificación (parte visible) y el cuerpo (implementación que se oculta). La sintaxis de la especificación es la siguiente:

```
package nombre_unidad is
       -- declaraciones visibles
private
       --declaraciones privadas
end nombre_unidad
```

La sintaxis del cuerpo es:

```
package body nombre_unidad is
       -- parte declarativa
end nombre_unidad
```

Ejemplo con ADA sería:

ESPECIFICACION

- PILA de enteros (100)

ENCAPSULA

```
package PILA IS
 type PILA limited private
 MAX: constant := 100
function EMPTY (P:in PILA) return boolean
prodedure PUSH (P:inout PILA,ELE:in INTEGER)
procedure POP (P: inout PILA)
procedure TOP (P:inPILA) return INTEGER
private
  type PILA is
  vecpila : array (1..MAX) of INTEGER
  tope: INTEGER range 0..MAX:=0
end PILA
```

OCULTA

IMPLEMENTACION

```
package body PILA is
   function EMPTY (P:in PILA) return boolean
        ..........
   end
   prodedure PUSH (P:inout PILA,ELE:in INTEGER)
        ..........
   end
   procedure POP (P: inout PILA)
        ........
   end
   procedure TOP (P:inPILA) return INTEGER
        ........
   end
end PILA
```

OCULTA

Instanciacion de una pila

```
with PILA
procedure USAR
  pil:PILA
  y: INTEGER
  ……..
  pil.PUSH (pil,y)
End USAR
```

En **CLU** el mecanismo para TAD son los `cluster`. El programador necesita utilizar la palabra CREATE para poder instanciar el TAD. En gral tiene la siguiente declaracion:

```
nombreTAD = cluster is operacion1, operacion2, . . ., operacionN
      variable_rep = representación interna del tipo
      implementación de operacion1
      implementación de operacion2
      . . .
      implementación de operacion
end nombreTAD
```

Un TAD en CLU provee operaciones para un tipo de datos, y no para un objeto, tal como lo hacen Smalltalk, Simula 67 y C++. Esto quiere decir que por ejemplo al definir la suma entre complejos no suponemos un complejo que recibe un mensaje con un parámetro, sino que partimos de la base de una operación entre dos complejos.

En **C++** se provee el mecanismo de *clases*, que es una extensión de las estructuras(struct) del C original. Las clases proveen encapsulamiento, separación de declaración e implementación y ocultamiento de información. No es necesario definir el procedimiento CREATE y provee TAD genérico a través de una forma de función genérica llamada *plantilla (template).*

La forma de crear una plantilla es la siguiente:

```
template < class tipo_pila, int tamanio> class Tpila
      { public: tipo_pila stg[tamanio]}
```

Esto crea una clase plantilla llamada `Tpila`, la cual tiene un parámetro entero `tamanio` que crea almacenamiento de datos de un arreglo llamado `stg` de `tamanio` y de tipo `tipo_pila`.

En **SIMULA 67** se usa el mecanismo `class` que soporta el encapsulamiento de la definición de los tipos abstractos de datos. En ella se encierran los procedimientos que realizan las operaciones sobre los datos.

Una declaración de class tiene la siguiente forma general `<cabecera_de_class>; <cuerpo_de_class>` donde `<cabecera_de_class>` contiene el nombre de la class y los parámetros formales y `<cuerpo_de_class>` es un bloque convencional. Por ejemplo:

```
numero_complejo (x, y): real x,y;
      real angulo, radio; -- declaración de variables
      …
      Implementación del tad
      …
numero_complejo
```

Y se instancia de la siguiente manera: `new numero_complejo (1.0, 1.0)`

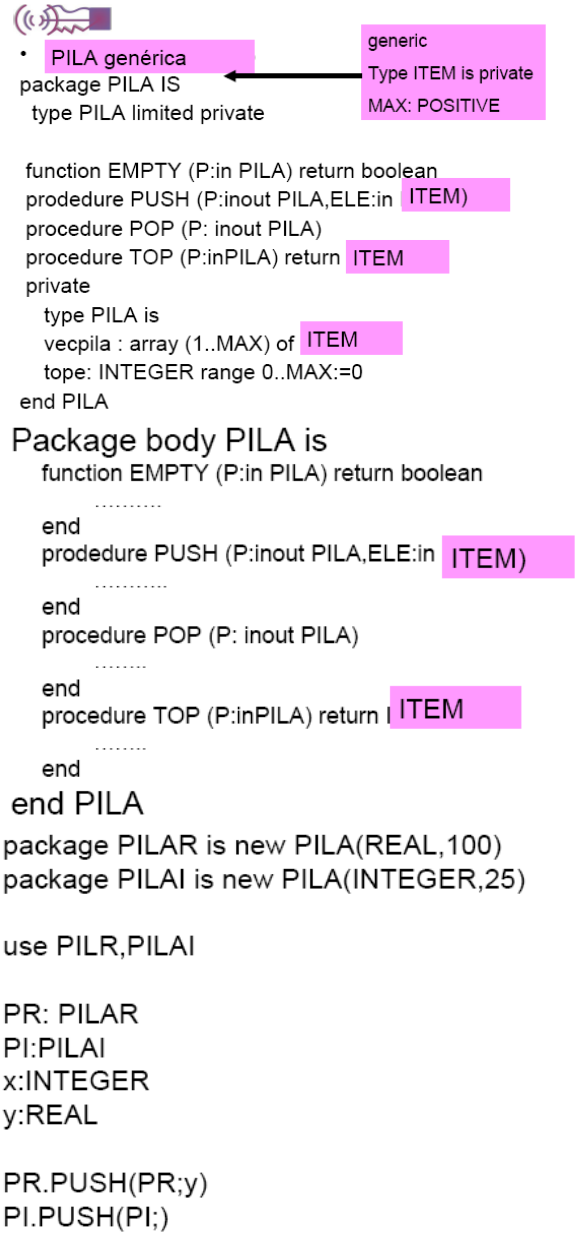
58. ¿Qué es el ocultamiento de información y el encapsulamiento?

El ocultamiento de la información consiste en definir módulos en donde cada uno esconde la definición de una estructura particular de datos. Se proporciona funciones que permite consultar y manipular la información contenida por la estructura de datos.

El encapsulamiento implica que el usuario del nuevo tipo no pueda manipular los objetos de datos del tipo, excepto por el uso de las operaciones definidas.

59. ¿La inicialización y la parametrización son indispensables para los TADs?

La inicialización y la parametrización son indispensables para los TAD's genéricos ya que se necesita indicar el tipo de dato que contendrá el TAD. Un ejemplo de de TAD parametrizado en Ada seria una pila para cualquier tipo de elemento y tamaño.



A menudo es conveniente parametrizar TAD. Por ejemplo para poder diseñar un TAD Pila que pueda almacenar cualquier tipo escalar.

En ADA no es necesario inicializar los TAD's ya que provee inicialización en la declaración, a veces no es necesario hacer una procedimiento CREATE para darle un valor al TAD.

60. ¿Cuál fue el aporte de SIMULA 67 para los TADs?

A diferencia de Algol y Pascal en Simula 67 un acceso solo se puede hacer a través de variables de referencia ya que las instancias de una class no tienen nombre. También diremos que una instancia de una class es un objeto inicializado ya que su cuerpo se ejecuta automáticamente con la sentencia new.

Desde un punto de vista más general, las class son mecanismos de encapsulamiento que soportan la definición de tipos abstractos de datos. Los class pueden encerrar los procedimientos que realizan las operaciones sobre los datos. Las operaciones son accesibles a través de la notación puntual y pueden tener parámetros.

61. Tiene relación los TAD con ADA a las cláusulas private y limited prívate?

Si, cuando en un paquete definimos un tipo para que no se acceda a los elementos internos de la implementación del mismo lo declaramos como private en la parte pública y se pone su definición en la parte privada. Al declarar un tipo como private su definición queda oculta y el usuario del paquete solo podrá utilizar con el las operaciones que se hallan declarado en la parte publica del paquete.
Cuando se define un tipo privado se predefinen inherentemente las operaciones de asignación, igualdad y desigualdad. Si no se quiere que exista ninguna operaciones sino únicamente las definidas en el paquete se debe emplear el tipo privado limitado.

62. Que es la compatibilidad de tipos? Como determino la compatibilidad de tipos? Que tipo de compatibilidad tiene algunos lenguajes?

La compatibilidad de tipo son reglas semánticas que determinan si el tipo de un objeto es válido en un contexto particular. Un lenguaje debe definir en que contexto el tipo Q es compatible con el tipo T. Si el sistema de tipos define la compatibilidad se puede realizar el chequeo de tipos.

Existen dos tipos de compatibilidad:

- Equivalencia por nombre: dos variables son del mismo tipo si y solo si están declaradas juntas (no para ADA) o si están declaradas con el mismo nombre de tipo. Extiende naturalmente los tipos predefinidos.
- Equivalencia por estructura: dos variables son del mismo tipo si y solo si los componentes de su tipo son iguales.

Las dos equivalencias son sintácticas y no semánticas, no se da la noción de comportamiento idéntico (TADS).
Los lenguajes tienen distintos tipos de compatibilidades. Ada y C ++tiene equivalencia por nombre. C tiene equivalencia por salvo para los registros. Pascal: por estructura salvo los parámetros formales que son por nombre. Y Algol 68 tiene compatibilidad de tipos por medio de la equivalencia estructural.

63. Que es la conversión de tipos? Coerción.

Con frecuencia es necesario convertir un valor de un tipo a un valor de otro: por ejemplo, cuando queremos sumar la variable entera v con la constante real 3.753. En la mayoría de los lenguajes tal conversión es implícita y se hace explícitamente por el compilador que genera código necesario.

La coerción significa convertir el valor de un tipo a otro.
Existe una clasificación de conversiones:

- Widening (ensanchar): cada valor del dominio tiene su correspondiente valor en el rango. (int a real)
- Narrowing (estrechar): cada valor del dominio puede no tener su correspondiente valor en el rango. En este caso algunos lenguajes producen un mensaje avisando la pérdida de información. (real a int)

Algol 68 aplica conversión de tipos implícitamente.

Estructuras de Control

64. Enumere los distintos niveles de estructuras de control y describa uno de ellos.

Las estructuras de Control son el medio por el cual los programadores pueden determinar el flujo de ejecución entre los componentes de un programa. Existen dos tipos de estructuras de control:

A Nivel de Sentencia: contribuyen en gran medida a la legibilidad y mantenimiento de los programas. Se dividen en tres grupos:

- *Secuencia*: indica la ejecución de una sentencia a continuación de otra. El delimitador más general y más usado es el ";" pero no necesariamente es la única manera de indicar secuencia.

Por ejemplo Fortran no tiene delimitador y utilizan implícitamente el fin de línea para separar instrucciones. Estos lenguajes se los llama 'orientados a línea'. Otros lenguajes como Smalltalk en particular, utiliza el delimitador ".".

Es posible agrupar un conjunto de sentencias en una secuenca para formar una única sentencia compuesta. . Algunos lenguajes (como Pascal y Algol 68) utilizan las palabras claves como begin y end, y otros como C, C++ y JAVA utilizan los caracteres "{" y "}" para agrupar sentencias.

- *Selección*: permite al programador especificar que ha hecho una elección entre un cierto número posibles de sentencias alternativas.

Evolución del IF:

- FORTRAN: especifica la ejecución de una sentencia seguida de una expresión lógica. Si la condición es verdadera ejecuta la sentencia.

```
if (condición lógica) sentencia
```

- ALGOL 60 : mas amplia y potente. La presencia de una rama else permite al programador escoger entre dos alternativas como consecuencia de una expresión lógica. A diferencia de FORTRAN una rama puede ser cualquier sentencia, por ejemplo, una sentencia compuesta.

```
if (condición lógica) then sentencia1
else sentencia2
```

El problema que tiene es que es ambiguo, ya que no se establece a quien corresponde el último else, con los if abierto. Por ejemplo

```
if (x>0) then if (x<10) then x := 0 else x :=1000
```

No esta claro si la alternativa else corresponde a la condición interna o a la mas externa. La ejecución de la sentencia para x = 15 asignara 1000 para x y según la ejeccion de la otra sentencia, x permanecerá invariable. Para eliminar la ambigüedad, la sentencia ALGOL requiere una sentencia incondicional en la rama then de una sentencia if. De modo que la sentencia anterior quedaría:

```
if (x>0) then begin if (x<10) then x := 0 else x :=1000 end
```

or

```
if (x>0) then begin if (x<10) then x := 0 end else x :=1000
```

- PL/1, Pascal y C: sin ambigüedad. Establecen por lenguaje que cada else cierra con el último if abierto pero aunque eliminan la ambigüedad tiene el problema que puede ser ilegible cuando posee muchos if anidados especialmente si el programa se ha escrito sin una indentación. Es recomendable utilizar Begin y End para hacer mas explicito la interpretación deseada.
- ADA Y ALGOL 68: sin ambigüedad y mas legibles. Incorporan la cláusula que cierra los if, la cual es el fi.

```
if i=0
       then j:= j+1;
       else j:=j-1;
fi
```

Ambos lenguajes permiten el `elif` que reemplaza al `else if`.

Sentencia SELECT

- PL/1 ha adoptado la construcción especial `select` con el fin de especificar la selección entre 2 o más opciones. Reemplazaria a los if anidados.

```
select
       when(A) sentencia1;
       when(B) sentencia2;
       .............
       otherwise sentencia n;
end;
```

Sentencia de selección múltiple en otros lenguajes

- ALGOL 68: la construcción de selección con múltiples alternativas se expresa con la sentencia `case`. La selección de una rama de un `case` debe ser de tipo entera. El valor de cada expresión i selecciona la rama i-ésima para ejecutarla.
  Tiene una cláusula opcional `out` que se ejecuta cuando el valor dado por la expresión no está expresado en el conjunto de valores. Si se omite la clausula `out` equivale a una clausula `out` con una sentencia `skip`. La sentencia `skip` es una sentencia nula.
- PASCAL: Incorpora que los valores de la expresión sean ordinales y ramas con etiquetas. No importan el orden en que aparecen las ramas. Es inseguro porque no establece qué sucede cuando un valor no cae dentro de las alternativas puestas.

```
var operador: char;
      Oper1, oper2, resultado: boolean
…
case operador of
     ".": resultado := oper1 and oper2;
     "+": resultado := oper1 or oper2;
     "=": resultado := oper1 = oper2;
end
```

- ADA: Combina los aspectos positivos de Pascal y de Algol68. Las expresiones pueden ser de tipo entero o enumeración. Además es preciso que se estipule en las selecciones todos los valores posibles que puede tomar la expresión. Tiene la cláusula `others` que se puede utilizar para representar a aquellos valores que no se especificaron explícitamente.

```
case Operador is
      when  '+' => result:= a + b;
      when  '-' => result:= a - b;
      when  others => result:= a * b;
end;
```

- C , C++: Para múltiple selección provee el constructor `switch`. Cada rama es etiquetada por uno o más valores constantes. Cuando se coincide con una etiqueta del `switch` se ejecutan las sentencias asociadas y se continúa con las sentencias de las otras entradas. Existe la sentencia break, que provoca la salida del `switch`.
  Tiene una cláusula `default` que sirve para los casos que el valor no coincida con ninguna de las opciones establecidas. Esta opción es optativa y si no está, y ninguno de los casos coincide, no se toma ninguna acción.

```
switch Operador {
      case  '+' :result:= a + b;  break;
      case  '-': result:= a - b;   break;
      default :  result:= a * b;  }
```

- *Iteración*: Este tipo de instrucciones se utilizan para representar aquellas acciones que se repiten un cierto número de veces.
  Su evolución
  - FORTRAN: Sentencia Do de Fortran. La variable de control solo puede tomar valores enteros

```
Do  label var-de-control= valor1, valor2
.....
label continue
```

  El Fortran original evaluaba si la variable de control había llegado al límite al final del bucle, o sea que siempre una vez lo ejecutaba.
  - Pascal, Algol 68, ADA, C , C++: sentencia `for`. La variable de control puede ser de cualquier valor ordinal, no solo enteros.
    Pascal no permite que se altere en el bucle los valores del límite inferior y superior, ni el valor de la variable de control. El valor de la variable fuera del bloque se asume indefinido.
    Algol 68 no permite que se toque el valor de la variable de control, pero sí los valores de los límites superior e inferior, porque los evalúa en el comienzo. Además el alcance de la variable de control es solo en el bucle.

Ada encierra todo proceso iterativo entre las cláusulas loop y end loop. Permite el uso de la sentencia Exit para salir del loop y la variable de control NO necesita declararse, se declara implícitamente cuando se entra al bucle y desaparece cuando se sale de él.

C, C++ se compone de tres partes: una inicialización y dos expresiones. La primer expresión (2do. Parámetro) es el testeo que se realiza ANTES de cada iteración. Si no se coloca el for queda en LOOP. En el primer y último parámetro se pueden colocar sentencias separadas por comas.

A Nivel de Unidad: Cuando el flujo de control se pasa entre unidades. Intervienen los pasajes de parámetros.

65. El tipo de instrucción secuencia ¿en todos los lenguajes tiene delimitador ";"?

La secuencia es el mecanismo de estructuración más simple del que disponen los lenguajes, para indicar que una sentencia se ejecutará a continuación de otra.

El delimitador más general y más usado es el ";" pero no necesariamente es la única manera de indicar secuencia.

Por ejemplo Fortran no tiene delimitador y utilizan implícitamente el fin de línea para separar instrucciones. Estos lenguajes se los llama 'orientados a línea'. Otros lenguajes como Smalltalk en particular, utiliza el delimitador ".".

66. ¿Existen sentencias compuestas? ¿todos los lenguajes la implementan de la misma manera?

Sí, existen sentencias compuestas, es decir, se puede agrupar un grupo de sentencias en una secuencia para formar una única secuencia compuesta.

No todos los lenguajes la implementan de la misma manera. Algunos lenguajes (como Pascal y Algol 68) utilizan las palabras claves como begin y end, y otros como C, C++ y JAVA utilizan los caracteres "{" y "}" para agrupar sentencias.

67. C y Pascal implementan la asignación de la misma manera?

C y Pascal no implementan la asignación de la misma manera. En C, la asignación la define como una expresión con efectos laterales, es decir, que devuelve un valor. Evalúa de derecha a izquierda. Por ejemplo: a = b = c = 90.

Además C permite cualquier expresión sobre el lado izquierdo de la asignación. Por ejemplo ++p = *q. Otra diferencia es el operador de asignación siendo = en C y := en Pascal.

68. ¿Una expresión de asignación puede producir efectos laterales que afecten al resultado final, dependiendo de cómo se evalúe? Dé ejemplos.

Sí. Una expresión de asignación puede tener diferentes resultados dependiendo de cómo se evalúe.(izq a der o der a izq) Por ejemplo:

```
z:= 1;                  Function F(var z: integer): integer
x := z + F(z);          begin
                                z := z + 1;
                        end
```

Si se evalúa de izq a der se obtiene x = 3 y si se evalúa de der a izq se obtiene 4

69. Compare conceptualmente una cascada de if then else contra un case.

El `if-then-else` y el `case` son estructuras de control de selección.

La sentencia `if-then-else` permite a partir de una condición elegir dos caminos posibles. En el caso de necesitar más de dos caminos, es necesario anidar esta sentencia en si misma, lo que hace que el código sea difícil de leer y producir ambigüedad.

El case permite elegir una rama a partir del valor de una expresión. Por un lado se gana mayor legibilidad, pero las expresiones que se pueden comparar generalmente son de tipos primitivos. En cambio los `if-then-else` permiten cualquier tipo de expresión booleana.

Las dos estructuras permiten tomar una acción por defecto si las condiciones no se cumplen (else o otherwise).

70. Muestre a través de ejemplos como resuelven los lenguajes el problema de if anidados.

Uno de los problemas que pueden presentarse es el de la ambigüedad. Por ejemplo en el siguiente caso:

```
if (x>0) then if (x<10) then x := 0 else x :=1000
```

No esta claro si la alternativa else corresponde a la condición interna o a la mas externa. La ejecución de la sentencia para x = 15 asignara 1000 para x y según la ejecución de la otra sentencia, x permanecerá invariable. Para eliminar la ambigüedad, la sentencia ALGOL 60 requiere una sentencia incondicional en la rama then de una sentencia if. De modo que la sentencia anterior quedaría:

```
if (x>0) then begin if (x<10) then x := 0 else x :=1000 end
```

or

```
if (x>0) then begin if (x<10) then x := 0 end else x :=1000
```

La solución que plantearon PL/1, Pascal y C, es establecer que cada else cierra con el último if abierto pero aunque eliminan la ambigüedad tiene el problema que puede ser ilegible cuando posee muchos if anidados especialmente si el programa se ha escrito sin una indentación.

Otra solución podría ser utilizar begin-end como delimitadores de bloques para hacer mas explicito la interpretación deseada.

ADA y Algol 68 implementan un if then else mas legible y sin ambigüedad usando la palabra reservada fi como un delimitador de cierre de la sentencia if. Ambos lenguajes permiten el `elif` que reemplaza al `else if`.

```
if i=0
      then j:= j+1;
      else j:=j-1;
fi
```

71. ¿Puede suceder en Pascal que un for quede en Loop?

No, no puede quedar en loop infinito porque Pascal no permite que se toquen ni los valores del limite inferior y superior ni el valor de la variable de control. La variable de control puede ser de cualquier valor ordinal.

72. Cite diferencias entre while y until.

WHILE	UNTIL
La condición se verifica antes de que se ejecuta el cuerpo del bucle	La condición se verifica después que se ejecua el cuerpo del bucle.
El cuerpo del bucle puede no ser ejecutado	El cuerpo del bucle se ejecuta al menos una vez
Las variables de la condición se deben inicializar antes de alcanzar la sentencia while	Las variables de la condición no necesitan ser inicializadas. Se les puede dar valor dentro del cuerpo.
Si la condición es verdadera se ejecutara el cuerpo y continuará el bucle.	Si la condición es verdadera el cuerpo del bucle habrá sido ejecutado pero se detiene el bucle.

Excepciones

73. Que es una excepción?

Una excepción es un proceso anómalo. Es un proceso, el cual la unidad donde se provocó está incapacitada para atenderlo de manera que termine normalmente.

74. ¿El lenguaje debería proveer algo especial para el manejo de las excepciones? ¿Todos los lenguajes los proveen?

Si, un lenguaje debe proveer un manejador para trabajar con excepciones.

Un manejador de excepciones realiza un procesamiento especial requerido cuando se detecta una excepción. Una excepción se levanta cuando ocurre un evento asociado y en ese momento se llama al manejador que lo resuelve. Algunas cuestiones a tener en cuenta en el diseño de un lenguaje de
programación que contenga manejo de excepciones son:

- ¿Cómo se maneja una excepción y cuál es su ámbito?
- ¿Cómo se alcanza una excepción?
- ¿Cómo especificar la unidades (manejadores de excepciones) que se han de ejecutar cuando se alcanza las excepciones?
- ¿A dónde se cede el control cuando se termina de atender las excepciones?

No todos los lenguajes proveen manejo de excepciones como por ejemplo Pascal y C. Y los que los provee son ADA, Delphi, C++, Java, PL-1 y CLU.

75. ¿Qué es un manejador?

Un manejador de excepciones realiza un procesamiento especial requerido cuando se detecta una excepción. Una excepción se levanta cuando ocurre un evento asociado y en ese momento se llama al manejador que lo resuelve. Algunas cuestiones a tener en cuenta en el diseño de un lenguaje de
programación que contenga manejo de excepciones son:

- ¿Cómo se maneja una excepción y cuál es su ámbito?
- ¿Cómo se alcanza una excepción?
- ¿Cómo especificar la unidades (manejadores de excepciones) que se han de ejecutar cuando se alcanza las excepciones?
- ¿A dónde se cede el control cuando se termina de atender las excepciones?

No todos los lenguajes proveen manejo de excepciones como por ejemplo Pascal y C. Y los que los provee son ADA, Delphi, C++, Java, PL-1 y CLU.

76. Para que sirve que un lenguaje tenga manejo de excepciones? **Diciembre 08**

77. ¿Qué ocurre cuando un lenguaje no provee manejo de excepciones? ¿Se podría simular?

Cuando un lenguaje no maneja excepciones, deben contemplarse los casos excepcionales dentro del programa. Por ejemplo, contemplar explícitamente que el divisor en una división sea distinto de cero, o la existencia de un archivo antes de abrirlo, etc.

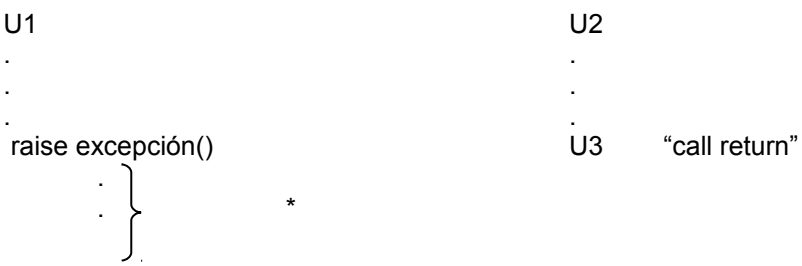
También lo que ocurre es que se aborta el programa que produce el error.

Se podría simular haciendo todos los controles para manejar todos lo posibles casos excepciones que pudieran ocurrir.

78. Diferencias entre el MANEJO DE EXCECPIONES y el esquema CALL-RETURN

Una unidad no es, en ningún caso, una parte del programa independiente ni autosuficiente. En caso de que sea un subprograma, se lo puede activar mediante una llamada realizada por otra unidad, a la que se le devolverá el control después de la ejecución. Por lo tanto, el punto de retorno no es una información que se debe preservar en el registro de activación cuando se hace la llamada al subprograma.

Para explicar las diferencias tenemos el siguiente ejemplo:



.

Una unidad U1 levanta una excepción Excepción(), la cual implícitamente llama a un manejador que se ejecuta, y dependiendo del modelo del manejo de excepción; si es por reasunción, se devuelve el control a la siguiente instrucción donde se produjo la excepción. En cambio, si es por terminación, se aborta la ejecución de U1.

En la unidad U2, se hace un llamado explícito a la unidad U3, la cual se ejecuta, y luego devuelve el control a la instrucción siguiente a su llamado.

79. ¿Qué modelo de manejo de excepciones conoce?

Existen dos tipos de modelo de manejo de excepciones:

Reasunción: se maneja la excepción y se devuelve el control al punto siguiente donde se invoco la excepción, permitiendo la continuación de ejecución de la unidad. El lenguaje que lo utilizan es PL/1

Terminación: se termina la ejecución de la unidad que alcanza la excepción y se transfiere el control al manejador. Los lenguajes que lo utilizan son ADA, CLU, DELPHI, C++.

80. Compare el modelo de excepciones de PL/1, ADA y CLU.

PL/1:

Utiliza el criterio de Reasunción, es decir que se maneja la excepción y se devuelve el control al punto siguiente donde se invoco la excepción, permitiendo la continuación de ejecución de la unidad.

Las excepciones son llamadas CONDITIONS y los manejadores se declaran con la sentencia ON y el manejador puede ser una instrucción simple o un bloque.

```
ON CONDITION (nombreExcepcion) manejadroDeExcepcion
```

Las excepciones se alcanzan explicitamente con la sentencia

```
SIGNAL CONDITION (NombreExcepcionALanzar)
```

Ya tiene un conjunto de excepciones incorporadas (built_in exceptions) con sus propios manejadores y se alcanzan automáticamente en la ejecución de algunas sentencias. Por ejemplo ZERODIVIDE cuando en la evaluación de una expresión el denominador de una división se encuentra en cero.

La acción realizada por un manejador la especifica el lenguaje pero puede ser redefinida como una excepción del usuario.

```
ON ZERODIVIDE BEGIN;
…
        END;
```

Estas excepciones pueden ser habilitadas y deshabilitadas explícitamente. Se habilitan anteponiendo el nombre de la built_in antes del bloque, instrucción o procedimiento al que van a afectar. Se deshabilitan anteponiendo NO al nombre de la built_in antes del bloque, instrucción o procedimiento al que van a afectar. Por ejemplo `(ZERODIVIDE): BEGIN ... END;` //habilitada `(NOZERODIVIDE): BEGIN ... END;` //deshabilitada.

Los manejadores se ligan dinámicamente con las excepciones con el último manejador definido. Si en el mismo bloque hay varias sentencias ON para la misma excepción, cada nuevo enlace anula al anterior. Si aparece una sentencia ON para la misma excepción en un bloque interno, el nuevo enlace permanece vigente solo hasta que acabe el bloque interno. Cuando salimos de un bloque, se restauran los enlaces que existían en la entrada del bloque anterior.

El alcance de un manejador de una excepción termina cuando finaliza la ejecución de la unidad donde fue declarado. El alcance de un manejador de una excepción se acota cuando se define otro manejador para esa excepción y se restaura cuando se desactivan los manejadores que la enmascararon.

Podemos concluir que el enlace es dinámico entre una excepción y su manejador por lo tanto los programas en PL/1 pueden ser poco manejables, difíciles de comprender y escribir. Además no se permiten pasar parámetros desde el punto donde se produce la excepción al correspondiente manejador. Por consiguiente, solo se puede establecer el flujo de información mediante as variables globales. Por ejemplo, cuando se alcanza la excepción `STRINGRANGE` que indica un intento de

acceder mas allá del limite de una cadena de caracteres, no hay manera de que el manejador sepa a que cadena se refiere si hay dos o mas cadenas visibles en el ámbito. Tales situaciones hacen que el manejador de excepciones en PL/1 sea a menudo ineficaz.

Ada:

Utiliza el criterio de Terminación. Cada vez que se produce una excepción se termina el bloque, procedimiento, paquete o tarea donde se levanto y se ejecuta el manejador asociado.

Las excepciones se declaran en la zona de definición de variables, colocando la palabra `Exception`, `e: exception`. El alcance de una excepción es el mismo alcance de las variables. Las excepciones se alcanzan explícitamente con la palabra clave `raise`.

```
RAISE (NombreExcepcionALanzar)
```

Los manejadores se encuentran en el final de cuatro diferentes unidades de programa: Bloque, Procedimiento, Paquete o Tarea. Forma de definirlos:

```
…
Begin
….
exception
       when nomExcep1 => Manejador1
       when  nomExcep2 => Manejador2
       …..
       when OTHERS  => Manejador (opcional);
End;
```

Ada ya tiene cuatro excepciones predefinidas con su manejador asociado. Estas pueden ser:

- Constraint_Error: Falla un chequeo en tiempo de ejecución sobre alguna limitación (por ejemplo: fuera del limite en un arreglo.)
- Program_Error: Falla un chequeo en tiempo de ejecución sobre alguna regla del lenguaje.
- Storage_Error: Falla un chequeo de tiempo de ejecución sobre la disponibilidad de memoria (por ejemplo por alocación dinámica.).
- Tasking_Error: Falla un chequeo de tiempo de ejecución en el sistema de task (por ejemplo en el manejo y la comunicación de tareas)

En algunas ocasiones, se necesita levantar de nuevo la excepción colocando solamente la palabra raise SIN NOMBRAR la excepción.

El pasaje de Parámetros únicamente se puede pasar como parámetro una cadena de caracteres.

```
 procedure A is
x,y,a : integer;
e1, e:exception;
procedure B (x : integer; y: integer) is
   m : integer;
   e:exception;
      begin
      ....
      Exception
            when e ⇒ x:=x+1; raise;
            when e1 ⇒m:=m + 1;
         end B;
begin
 ......
Exception
            when e ⇒ x:=x+1;
                when others ⇒ x:=x+2;
 end A;
```

Si se levanta la excepción **e …**

Es manejada primero por el manejador e de B y luego por el manejador others de A

CLU:

Utiliza el criterio de terminación.

Las excepciones sólo pueden ser alcanzadas por los procedimientos. Las excepciones que un procedimiento puede alcanzar han de declararse en la cabecera del mismo. Las excepciones se asocian a sentencias.

Las excepciones se alcanzan explicitamente con la palabra clave `signal`.

```
SIGNAL (NombreExcepcionALanzar)
```

El manejador de excepciones se relaciona con las sentencias simples o complejas por medio de la cláusula `except`, según la sintaxis:

```
<sentencia> except
                    when nomExcep1: Manejador1
                    when  nomExcep2: Manejador2
                    …..
                    when OTHERS  => Manejador (opcional);
              end
```

El lenguaje ya tiene excepciones predefinidas con su manejador asociado.

Al producirse una excepción:

- Se termina el procedimiento donde se levanto la excepción y devuelve el control al llamante inmediato donde se debe encontrar el manejador.
- Si el manejador se encuentra en ese ámbito entonces se ejecuta y luego el control pasa a la sentencia siguiente a la que esta ligado dicho manejador.
- Si el manejador no se encuentra en ese ámbito, la excepción se propaga estáticamente. Esto significa que el proceso se repite para las sentencias incluidas estáticamente.
- En caso de no encontrar un manejador en el procedimiento que hizo la llamada, el procedimiento señala implícitamente a la excepción predefinida FAILURE y devuelve el control.

En CLU las excepciones pueden devolver parámetros a sus manejadores.

Las excepciones pueden ser vueltas a levantar con RESIGNAL.

Ejemplo:

Si se levanta en A una excepción se corta el proceso de A y se busca el manejador

```
if ( .. ) then A(); except
                    when Nombre-Excepción: Manejador1;
                    when Nombre-Excepción : Manejador2;
                          .....
                    When others Nombre-Excepción:
              end;
```

81. Excepciones en C++ y en Java. Diferencias.

C++

Utiliza el criterio de Terminación.

Las excepciones pueden alcanzarse explícitamente a través de la sentencia throw.

Los manejadores van asociados a bloques que necesitan manejar ciertos casos excepcionales. Los bloques que pueden llegar a levantar excepciones van precedidos por la palabra clave Try y al finalizar el bloque se detallan los manejadores utlizando la palabra clave Catch(NombreDeLaExcepción).

```
…
Try
{
      ….. /* Sentencias que pueden provocar una excepción*/
}
catch(NombreExcepción1)
```

```
{
      ….. /* Sentencias Manejador 1*/
}
….
catch(NombreExcepciónN)
{
      ….. /* Sentencias Manejador N*/
}
```

Al levantarse una excepción dentro del bloque Try el control se transfiere al manejador correspondiente. Al finalizar la ejecución del manejador la ejecución continúa como si la unidad que provocó la excepción fue ejecutada normalmente.

Permite pasar parámetros al levantar la excepción. Ejemplo: Throw (Ayuda msg); Se está levantando la excepción Ayuda y se le pasa el parámetro msg.

Las excepciones se pueden levantar nuevamente colocando simplemente Throw.

Las rutinas en su interface pueden listar las excepciones que ellas pueden alcanzar. Ejemplo: void rutina throw (Ayuda, Zerodivide);

¿Qué sucede si la rutina …?

- termina porque alcanzó otra excepción que no está contemplada en el listado de la Interface? En este caso NO se propaga la excepción y una función especial se ejecuta automáticamente: unexpected(), que generalmente causa abort(), que provoca el final del programa. Unexpected puede ser redefinida por el programador.
- no colocó en su interface el listado de posibles excepciones a alcanzar? En este caso Si se propaga la excepción. Si una excepción es repetidamente propagada y no machea con ningún manejador, entonces una función terminate() es ejecutada automáticamente.
- colocó en su interface una lista vacía (throw())? Significa que NINGUNA excepción será propagada.

JAVA

Al igual que C++ las excepciones son objetos que pueden ser alcanzados y manejados por manejadores adicionados al bloque donde se produjo la excepción.

Diferencias:

- TODAS las excepciones que puedan ser alcanzadas explícitamente por la rutina y son chequeadas por el compilador, DEBEN ser listadas en la interface de la misma o ser manejadas por un manejador. La justificación de esta obligatoriedad es que el usuario de la rutina DEBE conocer qué excepciones puede alcanzar la misma.
- Uso de FINALLY

  Ej:
```
try
      bloque
  catch(NombreExcepciónN)
            bloqueManejadorN
  finally
      bloqueFinal
```

`finally` puede estar o no. Si está, la ejecución de su código se realiza cuando se termina la ejecución del bloque Try, se haya o no levantado una excepción. Salvo que el bloque Try haya levantado una excepción que no macheo con ningún manejador.

82. Defina un mecanismo que simule el manejo de excepciones en un lenguaje de programación que no las provea. Debe acercarse lo más posible a alguno de los modelos de excepciones.

Por ejemplo si tenemos el siguiente código en Pascal:

```
Procedure Manejador;
  ...
  end;
Procedure P(X:Proc);
  begin
```

```
    ....
        if Error then X;
  end;
Procedure A;
  begin
    ....
    P(Manejador);
  end;
....
```

Podemos decir que corresponde al modelo de Reasunción porque después de ejecutar la sentencia `if Error then X,` se sigue ejecutando las sentencias siguientes.

Para que encuadre con el modelo de Terminación el proceso P que es quien tiene al manejador debería terminar luego de ejecutar la sentencia `if Error then X.` Para ellos todas las sentencias que se encuentran debajo de la sentencia "if ERROR then X", deberían ubicarse en un else.

```
Procedure P(X:Proc);
begin
    ....
if Error then X;
else
…
end;
```

83. Supongamos que debe diseñar un lenguaje de programación. Que modelo de excepciones elegiría, por que?
84. Para una situación concreta definida por usted, compare una solución desde un lenguaje que provee manejo de excepciones y otro que no.
85. Definir un mecanismo de excepciones. Justificar.
86. Muestre un ejemplo de la potencia del manejo de excepciones en ADA.
87. Compare el modelo de excepciones de PL/I y el de ADA.

POO

88. Analice los orígenes del paradigma orientado a objetos dentro de la evolución histórica de los lenguajes.

    El paradigma orientado a objetos surge en la década del 70 donde la premisa era construir soft mantenible, abstracto y confiable. Es aquí donde decidieron crear un entorno y lenguaje llamado Smalltalk.

    Smalltalk tuvo como objetivo ser un lenguaje orientado a objetos en el cual su sistema era un conjunto de objetos que se comunican con mensajes.

    En los años 80 quisieron crear un sucesor al lenguaje C, incorporaron las principales ideas de Smalltalk y de Simula, creando el lenguaje C++. Puede afirmarse que se debe a este último la gran extensión de los conceptos de la orientación a objetos.

89. Según su criterio que es lo que conduce en la evolución histórica de los lenguajes a la POO.

    La primera y más importante razón es la reutilización. Hasta ahora sólo se podía obtener reutilización de dos formas: rutinas de bajo nivel o subsistemas completos. Esto limitaba fuertemente la reutilización, porque reaprovechar una aplicación quería decir aceptarla como era al 100%.

    El paradigma (la forma de pensar y representar la realidad) de la orientación a objetos es mucho mas potente que el estructurado y permite obtener más reusabilidad, por dos razones. En primer lugar porque se puede tener reusabilidad por separado, tanto del análisis como del diseño y la programación; la segunda razón es la herencia. Si una aplicación tiene algunas partes que no se adecuan a nuestras necesidades, podemos modificarlos mediante la herencia.

    La programación orientada a objetos también es mucho más fiable por diversas razones. En primer lugar por el desarrollo incremental y la programación por diferencia, al poder ir añadiendo funcionalidad vía herencia. La utilización masiva de librerías de clases garantiza la fiabilidad, ya que los componentes sólo se añaden a la librería cuando se ha verificado la corrección de su funcionamiento.

Provee las siguientes ventajas:

- Flexibilidad. Si partimos del hecho que mediante la definición de clases establecemos módulos independientes, a partir de los cuales podemos definir nuevas clases, entonces podemos pensar en estos módulos como bloques con los cuales podemos construir diferentes programas.
- Reusabilidad. Por medio de la reusabilidad podemos utilizar una clase definida previamente en las aplicaciones que nos sea conveniente. Es claro que la flexibilidad con la que se definió la clase va a ser fundamental para su reutilización.
- Mantenibilidad. Las clases que conforman una aplicación, vistas como módulos independientes entre sí, son fáciles de mantener sin afectar a los demás componentes de la aplicación.
- Extensibilidad. Gracias a la modularidad y a la herencia una aplicación diseñada bajo el paradigma de la orientación a objetos puede ser fácilmente extensible para cubrir necesidades de crecimiento de la aplicación.

90. Cuáles son los elementos mas importantes y hable sobre ellos en POO.

Los elementos que intervienen en POO son los siguientes:

- Objetos: son entidades que poseen estado interno y comportamiento. Es el equivalente a un dato abstracto. Es cualquier entidad del mundo real. Por ejemplo auto, animal, ventanas, menúes, vectores.
- Mensajes: Es una petición de un objeto a otro para que este se comporte de una determinada manera, ejecutando uno de sus métodos. TODO el procesamiento en este modelo es activado por mensajes entre objetos.
- Métodos: Es un programa que está asociado a un objeto determinado y cuya ejecución solo puede desencadenarse a través de un mensaje recibido por éste o por sus descendientes.
- Clases: Es un tipo definido por el usuario que determina las estructuras de datos y las operaciones asociadas con ese tipo. Cada objeto pertenece a una clase y recibe de ella su funcionalidad. Primer nivel de abstracción de datos: definimos estructura, comportamiento y tenemos ocultamiento. Un objeto: una caja negra cuya parte interna permanece oculta. La información contenida en el objeto solo puede ser accedida por la ejecución de los métodos correspondientes.
- Instancia de clase: Cada vez que se construye un objeto se está creando una INSTANCIA de esa clase. Una instancia es un objeto individualizado por los valores que tomen sus atributos.

91. Puntos de contacto y diferencia entre el POO y la programación operativa. Compare
92. Defina conceptualmente el paradigma OO. Comparelo con el imperativo. **JUNIO 08**

Programación funcional

93. Que es un programa escrito en un lenguaje funcional? Que rol cumple la computadora?

La programación en un lenguaje funcional consiste en construir definiciones y en usar el computador para evaluar expresiones.

El computador actúa como un evaluador o calculadora, su tarea es evaluar expresiones y mostrar los resultados. Lo que distingue un calculador funcional del resto es la capacidad del programador de introducir definiciones que incrementan su potencia de cálculo.

La programación funcional es aquélla en la que en lugar de construir el programa utilizando instrucciones se construye utilizando funciones. El esquema de un programa en este tipo de lenguajes se puede expresar de la siguiente forma: PROGRAMA = FUNCIONES + ESTRUCTURAS DE DATOS

Una función es una regla de asociación o correspondencia entre los elementos de un conjunto, que se llama el dominio, y los elementos de otro conjunto, que se llama el rango. La función siempre devuelve un valor.

Un lenguaje funcional tiene las siguientes componentes:

- Un conjunto de funciones primitivas.

- Un conjunto de formas funcionales.
- La operación aplicación.
- Un conjunto de objetos o datos.

94. Como se define el lugar donde se definen las funciones en un lenguaje funcional?

El lugar donde se definen las funciones en un lenguaje funcional es el script. Un script es una lista de definiciones que

- Pueden someterse a evaluación. Ejemplos: ?cuadrado (3 + 4) es 49, ?min 3 4 es 3
- Pueden combinarse, Ejemplo: ?min(cuadrado (1 + 1) 3)
- Pueden modificarse, ejemplo: Al script anterior le agrego nuevas definiciones:
  lado = 12
  área = cuadrado lado,
  En otras sesiones puedo utilizar área

95. Cual es el concepto de variables en los lenguajes funcionales?

No existe el concepto de variable en los lenguajes funcionales. Tampoco existen punteros.

96. Construya un ejemplo en el que se muestre el concepto de transparencia referencial.

Transparencia referencial es una propiedad de las expresiones en la cual dice que dos expresiones sintácticamente iguales darán el mismo valor porque no existen efectos laterales.

Gracias a la transparencia referencial podemos usar la operación de sustitución: la aplicación de una función se puede sustituir por su definición. Por ejemplo, par(doble(2*5))=par(doble 10)=par(2*10)=par(20)=true.

97. Diferencias entre programación funcional y programación estructurada

Los lenguajes imperativos se basan en ordenadores convencionales, son más eficaces en términos de tiempo de ejecución ya que reflejan la estructura y operaciones de la máquina, pero requieren que el programador ponga atención en detalles de nivel de máquina.

Los lenguajes funcionales se basan en funciones matemáticas. Su principal característica es la simplicidad y uniformidad de los objetos de datos permiten el diseño de estructura de datos sin preocuparse por las celdas de memoria. En lugar de asignarse, los valores se producen de la aplicación de funciones y se pasan como argumentos a otras funciones.

La programación funcional es una programación de más alto nivel que la programación procedural.

98. Principales características del paradigma funcional.

Las principales características del paradigma funcional son las siguientes:

- Sus programas están constituidos por funciones, entendiendo no como subprogramas clásicos sino como funciones puramente matemáticas. Estas funciones son de alto nivel de abstracción.
- Sus expresiones cumplen la propiedad de transparencia referencial en la cual dice que dos expresiones sintácticamente iguales darán el mismo valor porque no existen efectos laterales, es decir que además de retornar un valor, no modifica el estado de su entorno.
- No existe la asignación ni el cambio de estado en un programa. Las variables son identificadores de valores que no cambian en toda la evaluación (como constantes definidas con un DEFINE de C). Sólo existen valores y expresiones matemáticas que devuelven nuevos valores a partir de los declarados.
- Tipado fuerte: detección de errores en tiempo de compilación. El compilador, mediante un algoritmo, infiere el tipo de las expresiones. Si se declara el tipo de una expresión -> compilador lo chequea.
- Tipos Polimorficos: permite que el tipo de una función pueda ser instanciado de diferentes maneras en diferentes usos.
- Los bucles se modelan a través del uso de la recursividad, ya que no hay manera de incrementar o decrementar el valor de una variable. Las estructuras de control en los

programas funcionales son la composición funcional, construcción condicional y la recursividad.

- Currificación: Mecanismo que reemplaza argumentos estructurados por argumentos más simples. Por ejemplo: sean dos definiciones de la Función "Suma"

  Suma(x,y) = x + y

  Suma' x y = x + y

  Existen entre estas dos definiciones una diferencia sutil: "Diferencia de tipos de función"

  El tipo de Suma es : (num,num) => num y

  El tipo de Suma' es : num => (num=>num) /* por cada valor de x devuelve una función */

  Aplicando la función:

  Suma(1,2) =>3

  Suma' 1 2=> Suma'1 aplicado al valor 2 /*para todos los valores devuelve el siguiente*/

99. Describa los objetivos de los lenguajes funcionales y analice por que este tipo de lenguajes no tiene "difusión comercial".

    Los lenguajes funcionales han tenido éxito solamente dentro de la comunidad dedicada a la investigación (ejemplos son ML y Lisp).

    Los lenguajes aplicativos en general, nunca han alcanzado un gran éxito comercial; debido a que es difícil usar estos lenguajes ya que carecen del concepto de un estado del programa. Los datos dentro del registro de activación son una cuestión fundamental en el diseño de casi todos los lenguajes, y es difícil tratar con un concepto como este en la mayoría de los lenguajes aplicativos.

    Otra causa puede ser que los lenguajes funcionales no proveen un entorno de desarrollo amigable, ya que los programas desarrollados en estos lenguajes suelen no ser más que meros scripts.

100. Que es un lenguaje híbrido?
101. Introduzca el pradigma funcional en la evolucion de los lenguajes y comparelos con el impertaivo estableciendo ventajas y desventajas. **AGOSTO 08**

Del archivo Top 40 faltan responder

1) Construya dos ejemplos de pasaje por referencia y valor-resultado. Uno que el efecto sea el mismo y otro que sea diferente.
2) Construya un ejemplo que muestre un ejemplo de transparencia referencial.
3) Explique cual fue el aporte de AGOL en los lenguajes de programación.
4) Dentro de la evolución histórica de los lenguajes de programación marque la evolución histórica de Pascal.
5) Enumere al menos dos ventajas y dos desventajas del pasaje por nombre comparado con el pasaje por referencia.
6) Cree que hay algún punto de contacto entre los mecanismos necesarios para un parámetro procedimiento y un parámetro por nombre.
7) Explique las ventajas del registro variante de AGOL sobre el de ADA.
8) En cada uno de los lenguajes ejemplificadores (CLU, ADA, PL/1) identifique:
   a- Cuál es la forma sintáctica del constructor.
   b- Cómo se instacian los objetos.
   c- Cuál es el alcance y el tiempo de vida de los objetos creados.
   d- Permite segmentos de código inicial.
   e- Se puede parametrizar una definición.

Del archivo Top 30 faltan responder

1) Compare el pasaje by-name con otras formas de pasajes.
2) Heap. ¿Para qué se necesita?
3) Muestre un ejemplo que muestre porque es necesaria mayor información en los parámetros por procedimientos.
4) Puntualice al menos dos ejemplos con las variables apuntadas y proponga una solución para cada uno de ellos.
5) Describa y compare el dispose y el garbage colector.
6) Si estuviera diseñando un lenguaje: ¿qué lo llevaría a elegir cada uno de los lenguajes de equivalencia de tipos?
7) Cómo asocia un manejador a una excepción y cuáles son las reglas de alcance de los manejadores en cada uno de los modelos. (Reasunción y terminación).
8) Construya un ejemplo que muestre la ventaja de AGOL-68 sobre PASCAL en el manejo de registros variantes.
9) Ejemplificar y explicar el riesgo de las referencias sueltas.
10) Acceda en un procedimiento recursivo a las variables del llamador recursivo anterior.