Práctica 7

1) Conjunto de reglas que usa un lenguaje para estructurar y organizar sus tipos.
   Funciones principales:
   - Prevenir errores: Detecta operaciones inválidas, como sumar un número con una cadena.
   - Garantizar seguridad: Asegura que las variables se usen de manera coherente durante la ejecución.
   - Ayudar en la optimización: Los compiladores pueden generar mejor código si conocen los tipos.
   - Documentación implícita: Facilita la comprensión del código al indicar qué tipo de datos se espera.

   Funciones
   - Provee Mecanismos De Expresión:
     - Expresar tipos predefinidos, definir nuevos tipos y asociarlos con constructores del lenguaje.
   - Define Reglas De Resolución:
     - Equivalencia de tipos → ¿Dos valores tienen el mismo tipo?.
     - Compatibilidad de tipos → ¿Puede usarse el tipo en este contexto?.
     - Inferencia de tipos → ¿Cuál tipo se deduce del contexto?

   Mientras más flexible en el tipado sea el lenguaje, más complejo será su sistema de tipos.

   Tipado fuerte: Se dice que el sistema de tipos es fuerte cuando especifica restricciones de forma clara sobre cómo las operaciones que involucran valores de diferentes tipos pueden operarse. Un lenguaje es de tipado fuerte si no permite realizar operaciones entre tipos incompatibles sin conversión explícita.
   Tipado débil: Un lenguaje es de tipado débil si permite operaciones entre tipos incompatibles, realizando conversiones implícitas (a veces peligrosas).
   - Un Sistema Débil es menos seguro pero ofrece una mayor flexibilidad.
   - Un Sistema Fuerte es más seguro pero ofrece una menor flexibilidad. El compilador asegura la detección de todos los errores de tipos y la ausencia de estos en los programas.

   Ej:
   - Tipado fuerte:
     - Python: 3 + "4" lanza un error de tipo.
     - Java: No permite asignar una String a una variable int sin conversión.
   - Tipado débil:
     - JavaScript: 3 + "4" resulta en "34" (convierte el número a cadena).
     - C: Permite sumas entre char y int sin errores, haciendo conversiones implícitas.

   Tipado estático: Los tipos se verifican **en tiempo de compilación**. El código no compila si hay errores de tipo.
   Tipado dinámico: Los tipos se verifican **en tiempo de ejecución**. Puede compilar (o ejecutarse) pero fallar si ocurre un error de tipo.

Ej:

- Tipado estático:
  - Java: Debes declarar los tipos de las variables. El compilador revisa los tipos antes de ejecutar.
  - C++: Detecta errores de tipo en compilación, requiere declaración explícita.
- Tipado dinámico:
  - Python: Puedes asignar cualquier tipo a una variable sin declarar su tipo.
  - JavaScript: Las variables pueden cambiar de tipo durante la ejecución (let x = 5; x = "hola"; es válido).

2) Un tipo de dato es una categoría o clasificación de los datos que define:
   - El conjunto de valores posibles (por ejemplo: números enteros, booleanos, cadenas).
   - Las operaciones válidas que se pueden realizar sobre esos valores.

   Sirve para garantizar el uso correcto de los datos en un programa, ayudando a prevenir errores y a interpretar correctamente la información.

   Tipo Predefinido Elemental: Un tipo predefinido elemental (también llamado tipo primitivo o básico) es un tipo de dato que está incluido por defecto en el lenguaje de programación. Es fundamental, ya que sirve como base para construir otros tipos más complejos. Un tipo de dato predefinido refleja el comportamiento del hardware subyacente y es una abstracción de él. En particular los elementales/escalares son tipos de datos predefinidos indivisibles. Ej: Enteros, Reales, Boolean, etc.

   Tipo Predefinido por el Usuario: Un tipo definido por el usuario es un tipo de dato que el programador crea utilizando combinaciones de tipos existentes especificando agrupaciones de objetos de datos elementales y de forma recursiva, para representar estructuras más complejas o personalizadas. Esto se logra mediante constructores. Ej: Registros, Enumerados, Arreglos, etc.

3) **Producto Cartesiano:** Un producto cartesiano es un tipo compuesto que agrupa varios valores de distintos tipos en una sola estructura. Cada componente tiene un valor y un tipo fijo y conocido. Representa la idea de "tener todos estos valores". Permite construir, por ejemplo, los registros en Pascal o struct en C.
   Correspondencia Finita: Una correspondencia finita es una estructura que asocia claves con valores dentro de un conjunto finito de pares. Equivale a lo que en programación llamamos un diccionario, mapa o tabla de asociación. Es una función Un conjunto finito de valores de un tipo de dominio DT en valores de un tipo de dominio RT. Permite construir, por ejemplo, listas indexadas, arreglos, matrices, etc.
   Unión y Unión Discriminada: Una unión o sum type es un tipo compuesto que puede contener un valor de entre varios tipos posibles, pero solo uno a la vez. Permite manipular diferentes tipos en distintos momentos de la ejecución. Chequeo Dinámico, no se puede asegurar en compilación qué tipo o variante adquiere una variable. La Unión Discriminada agrega un descriptor (enumerativo) que me permite

saber con quien estoy trabajando y acceder correctamente a lo que tengo que acceder, ya que nos dice cual de los campos posee valor.
Recursión: Un tipo recursivo es un tipo de dato que se define en términos de sí mismo. Se usa para representar estructuras jerárquicas o repetitivas, como listas, árboles, etc.
Define Datos Agrupados:

- Cuyo tamaño puede crecer arbitrariamente.
- Cuya estructura puede ser arbitrariamente compleja.

Los lenguajes soportan la implementación de tipos de datos recursivos a través de los punteros.

Java: Producto Cartesiano.
C: Producto Cartesiano y Recursión.
C: Unión.
Ruby: Correspondencia Finita.
PHP: Correspondencia Finita.
Python: Correspondencia Finita.
Haskell: Recursión.
Haskell: Unión.

4) Mutabilidad significa que un dato puede ser modificado después de ser creado. Por el contrario, inmutabilidad indica que el contenido de un dato no puede cambiar una vez definido.
Ej en Python:
   - Mutable (Lista):
     ```
     lista = [1, 2, 3]
     lista.append(4)  # Se modifica: [1, 2, 3, 4]
     print(lista)
     ```
   - Inmutable (Tupla):
     ```
     tupla = (1, 2, 3)
     # tupla[0] = 10  # ❌ Error: las tuplas no se pueden modificar
     ```

   Ej en Ruby:
   - Mutable (String):
     ```
     mensaje = "hola"
     mensaje << " mundo"  # Se modifica: "hola mundo"
     puts mensaje
     ```
   - Inmutable (String + .freeze):
     ```
     nombre = "Ana"
     nombre.freeze
     # nombre << " María"  # ❌ Error: can't modify frozen String
     ```

   Dado el código proporcionado, no se puede afirmar que Dato.new(1) es mutable, ya que no lo está modificando, sino que está referenciando a un nuevo Dato. Por ende, no está modificando el contenido del objeto anterior, sino referenciando uno nuevo.

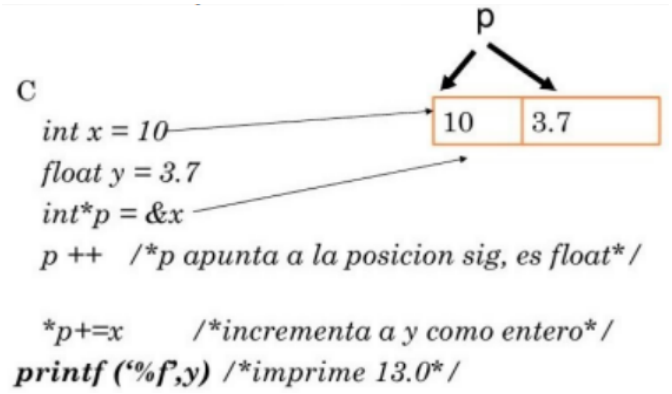
5) Sí. El lenguaje C permite tomar el l-valor (location value, o dirección de memoria) de las variables mediante el operador &.

```c
#include <stdio.h>

int main() {
    int x = 10;
    int *ptr = &x; // Se toma el l-valor de x (su dirección)
    printf("Valor de x: %d\n", x);
    printf("Dirección de x: %p\n", (void*)ptr);
    return 0;
}
```

Problemas:

○ Violación de Tipos



○ Referencias Sueltas

- 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.

○ Punteros no Inicializados

- Peligro de acceso descontrolado a posiciones de memoria.
- Verificación dinámica de la inicialización.
- Solución -> valor especial nulo:
  - nil en Pascal.
  - void en C/C++.
  - null en ADA,Python.

○ Punteros y uniones discriminadas

- Puede permitir accesos a cosas indebidas.
- Java lo Soluciona eliminando la noción de puntero explícito completamente.

○ Alias

- Si 2 o más Punteros comparten Alias, la modificación que haga uno se verá también reflejado en los demás.

○ Liberación de Memoria - Objetos Perdidos

- Si los objetos en la heap dejan de ser accesibles, esa memoria podría liberarse.
- Un objeto se dice accesible si alguna variable en la pila lo apunta directa o indirectamente.
- Un objeto es basura si no es accesible .

6) Un **TAD (Tipo Abstracto de Dato)** es una abstracción que define un conjunto de datos y operaciones válidas sobre ellos, ocultando su implementación interna.
   Encapsulamiento:
   - La Representación del tipo y las operaciones permitidas para los objetos del tipo se describen en una única unidad sintáctica.
   - Refleja las abstracciones descubiertas en el diseño.

   Ocultamiento de Información:
   - La representación de los objetos y la implementación del tipo permanecen ocultos.
   - Refleja los niveles de abstracción. Modificabilidad.

- En ADA se conocen como Paquete.
- En Modula-2 se conoce como módulo.
- En C++ y Java se conocen como clase.