Práctica 7: Sistemas y tipos de Datos

Ejercicio 1: Sistemas de tipos:

1. ¿Qué es un sistema de tipos y cuál es su principal función?

Sistema de Tipos: Conjunto de reglas que usa un lenguaje para estructurar y organizar sus tipos. El objetivo de un sistema de tipos es escribir programas seguros. Conocer estos sistemas nos permite conocer mejor los aspectos semánticos de los diversos lenguajes.

Funciones: Provee mecanismo de expresión que permite expresar tipos predefinidos, definir nuevos tipos y asociarlos con constructores del lenguaje. Define reglas de resolución que incluyen la equivalencia de tipos para determinar si dos valores tienen el mismo tipo, la compatibilidad de tipos para evaluar si un tipo puede usarse en este contexto y la inferencia de tipos para deducir cuál tipo se deriva del contexto.

2. Definir y contrastar las definiciones de un sistema de tipos fuerte y débil (probablemente en la bibliografía se encuentren dos definiciones posibles. Volcar ambas en la respuesta). Ejemplificar con al menos 2 lenguajes para cada uno de ellos y justificar.

Definición 1: Rigidez en la prevención de errores de tipo

Un sistema de tipos fuerte impide operaciones entre tipos incompatibles, evitando conversiones implícitas que puedan causar errores, ya sea en compilación (estático) o ejecución (dinámico), mientras que un sistema débil permite estas conversiones, priorizando flexibilidad, aunque pueda generar comportamientos inesperados.

Definición 2: Seguridad frente a errores de tipo en ejecución

Un sistema fuerte asegura que no haya errores de tipo no verificados en tiempo de ejecución, garantizando seguridad de tipos, mientras que un sistema débil permite operaciones que violan esta seguridad, como reinterpretar datos mediante punteros, lo que puede provocar errores o comportamientos indefinidos.

Ejemplos de lenguajes con sistemas de tipos fuertes

- Python, dinámicamente tipado, es fuertemente tipado porque no permite conversiones implícitas entre tipos incompatibles (como sumar una cadena y un entero, lo que genera un TypeError) y asegura seguridad de tipos en ejecución, evitando reinterpretaciones inseguras.
- Java, estáticamente tipado, es fuertemente tipado al exigir declaraciones de tipos y prohibir operaciones entre tipos incompatibles sin casting explícito, además de garantizar seguridad de tipos en la JVM, previniendo errores de tipo no verificados.

Ejemplos de lenguajes con sistemas de tipos débiles

- JavaScript, dinámicamente tipado, es débilmente tipado porque realiza conversiones implícitas (como "10" + 5 que resulta en "105", o "10" - 5 que da 5), lo que puede llevar a resultados inesperados, y permite ciertas reinterpretaciones de datos en operaciones de bajo nivel.
- C, estáticamente tipado, es débilmente tipado al permitir conversiones implícitas (como sumar un int y un float) y operaciones que violan la seguridad de tipos, como reinterpretar un int como un char mediante punteros, lo que puede causar errores.

3. Además de la clasificación anterior, también es posible caracterizar el tipado como estático o dinámico. ¿Qué significa esto? Ejemplificar con al menos 2 lenguajes para cada uno de ellos y justificar.

Ejercicio 2: Tipos de datos:

1. Dar una definición de tipo de dato.

Un tipo de dato es un conjunto de valores y un conjunto de operaciones asociadas al tipo para manejar esos valores.

2. ¿Qué es un tipo predefinido elemental? Dar ejemplos.

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, es decir, estos no se pueden descomponer a partir de otros.

Ejemplos: Enteros, reales, caracteres, booleanos, punteros, etc.

3. ¿Qué es un tipo definido por el usuario? Dar ejemplos.

Los lenguajes de programación permiten al programador especificar agrupaciones de objetos de datos elementales y de forma recursiva, agregaciones de agregados. Esto se logra mediante constructores. A estas agrupaciones se las conoce como tipos de datos definidos por el usuario.

Ejemplos: Enumerados, arreglos, registros, listas, etc.

Ejercicio 3: Tipos compuestos:

1. Dar una breve definición de: producto cartesiano (en la bibliografía puede aparecer también como *product type*), correspondencia finita, uniones (en la bibliografía puede aparecer también como *sum type*) y tipos recursivos.

Producto Cartesiano

El Producto Cartesiano se refiere a la combinación de n conjuntos de tipos variados, lo que permite producir estructuras como registros en Pascal o structs en C, donde se agrupan datos de diferentes tipos en una sola entidad.

Correspondencia Finita

La Correspondencia Finita es una función que mapea un conjunto finito de valores de un tipo de dominio DT (por ejemplo, un entero) a valores de un tipo de dominio RT, accesible a través de un índice. Este concepto se aplica a estructuras como listas indexadas, vectores, arreglos y matrices, que permiten organizar datos de manera estructurada y acceder a ellos mediante índices.

Unión y Unión Discriminada

La Unión o Unión Discriminada de uno o más tipos representa la disyunción de los tipos dados, con campos mutuamente excluyentes que no pueden tener valores al mismo tiempo, permitiendo manipular diferentes tipos en distintos momentos de la ejecución. Este tipo de estructura requiere chequeo dinámico, ya que no se puede asegurar en tiempo de compilación qué variante o tipo adquiere una variable. La Unión Discriminada mejora la unión al agregar un descriptor enumerativo que indica cuál de los campos tiene valor, brindando mayor seguridad al manipular el elemento según el discriminante, aunque este descriptor puede omitirse si se desea.

Recursión

Un tipo de dato recursivo T se define como una estructura que puede contener componentes de su mismo tipo T, como árboles o listas en Pascal, que son ejemplos comunes de esta categoría. Los datos agrupados definidos de esta manera tienen un tamaño que puede crecer arbitrariamente y una estructura que puede ser arbitrariamente compleja. Los lenguajes de programación soportan la implementación de tipos de datos

recursivos mediante el uso de punteros, que permiten referenciar elementos del mismo tipo de manera dinámica.

2. Identificar a qué clase de tipo de datos pertenecen los siguientes extractos de código. En algunos casos puede corresponder más de una:

Java
```
class Persona {
  String nombre;
  String apellido;
  int edad;
}
```

C
```
typedef struct _nodoLista {
  void *dato;
  struct _nodoLista *siguiente
} nodoLista;

typedef struct _lista {
  int cantidad;
  nodoLista *primero
} Lista;
```

C
```
union codigo {
  int numero;
  char id;
};
```

Ruby
```
hash = {
  uno: 1,
  dos: 2,
  tres: 3,
  cuatro: 4
}
```

PHP
```
function doble($x) {
  return 2 * $x;
}
```

Python
```
tuple = ('physics',
'chemistry', 1997, 2000)
```

Haskell
```
data ArbolBinarioInt =
  Nil |
  Nodo int
    (ArbolBinarioInt dato)
    (ArbolBinarioInt dato)
```

Ayuda para interpretar:
'ArbolBinarioInt' es un tipo de dato que puede ser Nil ("vacío") o un Nodo con un dato número entero (int) junto a un árbol como hijo izquierdo y otro árbol como hijo derecho

Haskell
```
data Color =
  Rojo |
  Verde |
  Azul
```

Ayuda para interpretar:
'Color' es un tipo de dato que puede ser Rojo, Verde o Azul.

- Java (class Persona): Producto Cartesiano.
- C (typedef struct _nodoLista y _lista): Recursión en lista (principalmente), con elementos de Producto Cartesiano con el struct.
- C (union codigo): Unión.
- Ruby (hash): Correspondencia Finita.
- PHP (function doble): Correspondencia Finita.
- Python (tuple): Producto Cartesiano (principalmente), con elementos de Correspondencia Finita.
- Haskell (data ArbolBinarioInt): Recursión y Unión.
- Haskell (data Color): Unión.

Ejercicio 4: Mutabilidad/Inmutabilidad:

1. Definir mutabilidad e inmutabilidad respecto a un dato. Dar ejemplos en al menos 2 lenguajes. TIP: indagar sobre los tipos de datos que ofrece Python y sobre la operación #freeze en los objetos de Ruby.

Mutabilidad e inmutabilidad son términos que se refieren a la capacidad o no de un dato de ser modificado después de su creación. Un dato mutable es aquel que se puede cambiar después de haber sido creado, mientras que un dato inmutable es aquel que no se puede cambiar después de haber sido creado.

En Python, algunos ejemplos de datos inmutables son los enteros, las cadenas y las tuplas. Por otro lado, los datos mutables en Python incluyen listas, conjuntos y diccionarios, ya que se pueden modificar después de haber sido creados.

En Ruby, la mayoría de los objetos son mutables por defecto, pero se puede hacer que un objeto sea inmutable utilizando el método *freeze*. Después de que un objeto es congelado, no se puede modificar.

2. Dado el siguiente código:

```
a = Dato.new(1)
a = Dato.new(2)
```

¿Se puede afirmar entonces que el objeto "Dato.new(1)" es mutable? Justificar la respuesta sea por afirmativa o por la negativa.

No se puede afirmar que el objeto Dato.new(1) es mutable basándonos en el código proporcionado. La razón es que el código no intenta modificar el estado del objeto Dato.new(1); simplemente reasigna la variable a a un nuevo objeto Dato.new(2). La reasignación no implica mutabilidad, ya que no altera el estado interno del objeto original. Sin conocer la definición de la clase Dato, no podemos determinar si Dato.new(1) es mutable (es decir, si permite modificar su estado) o inmutable (es decir, si cualquier cambio genera un nuevo objeto). Para responder definitivamente, necesitaríamos ver la implementación de la clase Dato y verificar si tiene métodos que permitan modificar su estado interno.

Ejercicio 5: Manejo de punteros:

1. ¿Permite C tomar el l-valor de las variables? Ejemplificar.

Sí, el lenguaje C permite tomar el l-value de las variables, lo que permite modificar el contenido de esa ubicación mediante asignaciones u otras operaciones.

```c
#include <stdio.h>

int main() {
    int x = 10;          // 'x' es un l-value, tiene una dirección de memoria
    int *ptr;            // Puntero para almacenar la dirección de 'x'

    ptr = &x;            // Tomamos el l-value de 'x' (su dirección) y lo asignamos a
'ptr'
    printf("Valor de x: %d\n", x);     // Imprime 10
    printf("Dirección de x: %p\n", (void*)ptr); // Imprime la dirección de memoria de
'x'

    *ptr = 20;           // Modificamos el contenido de la dirección de 'x' a través
del puntero
    printf("Nuevo valor de x: %d\n", x); // Imprime 20

    return 0;
}
```

2. ¿Qué problemas existen en el manejo de punteros? Ejemplificar.

Violación de tipos: La violación de tipos ocurre cuando un puntero se usa para acceder a datos de un tipo diferente al esperado, lo que puede llevar a comportamientos indefinidos o errores.

```c
int x = 10;
float *ptr = (float*)&x;
printf("%f\n", *ptr); // Acceso indebido: interpreta un int como float
```

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.

```
int *p = malloc(sizeof(int));
*p = 5;
free(p); // Desaloca la memoria
printf("%d\n", *p); // Error: referencia suelta
```

Punteros no Inicializados: Un puntero no inicializado contiene una dirección aleatoria, lo que puede provocar accesos descontrolados a memoria si se usa sin inicialización previa, requiriendo verificación dinámica.

```
int *p; // Puntero no inicializado
*p = 10; // Error: acceso a memoria inválida
```

Punteros y uniones discriminadas: El uso de punteros con uniones discriminadas puede permitir accesos indebidos si no se verifica correctamente el tipo de datos almacenado, accediendo a campos incorrectos.

```
union u { int i; char c; } var;
int *p = (int*)&var;
var.c = 'a';
printf("%d\n", *p); // Indebido: interpreta char como int
```

Alias: Cuando dos o más punteros comparten un alias (apuntan a la misma memoria), una modificación a través de uno afecta a los demás, lo que puede causar efectos secundarios inesperados.

```
int x = 10;
int *p1 = &x;
int *p2 = &x; // Alias
*p2 = 20;
printf("%d\n", *p1); // Imprime 20 por el alias
```

Liberación de Memoria - Objetos Perdidos: Si los objetos en la heap dejan de ser accesibles (porque ninguna variable en la pila los apunta directa o indirectamente), se convierten en basura y la memoria no se libera, causando fugas de memoria.

```
int *p = malloc(sizeof(int));
p = NULL; // Objeto perdido, memoria no liberada
// free(p) no se ejecuta
```

Ejercicio 6: TAD:

1. ¿Qué características debe cumplir una unidad para que sea un TAD?

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.

2. Dar algunos ejemplos de TAD en lenguajes tales como ADA, Java, Python, entre otros.

- Modula-2: módulo.
- Ada: paquete.
- C++, Python y Java: clase.

Ada, Java, Python y Haskell implementan TAD para estructuras como pilas, colas, conjuntos y listas, utilizando mecanismos como paquetes, interfaces, clases y tipos algebraicos.