Conceptos de Paradigmas y Lenguajes de Programación

Resumen teorías

Clase 7 - Tipos de datos

Tipos abstractos de datos(TADs)

TAD = Representación (datos) + Operaciones (funciones y procedimientos)

Los tipos de datos son abstracciones y el proceso de construir nuevos tipos se llama abstracción de datos

Los nuevos tipos de datos definidos por el usuario se llaman tipos abstractos de datos

Tipo abstracto de dato (TAD) es el que satisface:

- 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 la información: la representación de los objetos y la implementación del tipo permanecen ocultos. Refleja los niveles de abstracción

Especificación de un TAD

La especificación formal proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones

Ha de incluir una parte de sintaxis y una parte de semántica

Por ejemplo:

- Operación(Tipo argumento, ...) -> Tipo resultado (sintaxis)
- Operación(valores particulares argumentos) expresión resultado (semántica)

Hay operaciones definidas por sí mismas que se consideran constructores del TAD

Normalmente, se elige como constructor la operación que inicializa

Clases

En términos prácticos, una clase es un tipo definido por el usuario

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce (atributos + métodos)

Agrega un segundo nivel de abstracción que consiste en agrupar las clases en jerarquías de clases

De forma que la clase hereda todas las propiedades de la superclase

Sistema de tipos

Conjunto de reglas usadas por un lenguaje para estructurar y organizar sus tipos

El objetivo de un sistema de tipos es escribir programas seguros

Conocer el sistema de tipos de un lenguaje nos permite conocer de una mejor forma los aspectos semánticos del lenguaje

Provee mecanismos de expresión para:

- Expresar tipos intrínsecos o definir tipos nuevos
- Asociar los tipos definidos con construcciones del lenguaje

Define reglas de resolución:

- Equivalencia de tipos: ¿dos valores tienen el mismo tipo?
- Compatibilidad de tipos: ¿puedo usar el tipo en este contexto?
- Inferencia de tipos: ¿cuál tipo se deduce del contexto?

Mientras más flexible el lenguaje, más complejo el sistema (seguridad vs flexibilidad)

Se dice que el sistema de tipos es fuerte cuando especifica restricciones sobre cómo las operaciones que involucran valores de diferentes tipos pueden operarse

Lo contrario establece un sistema débil de tipos

Tipo y tiempo de chequeo

Tipos de ligadura

Tipado estático: ligaduras en compilación. Para esto puede exigir que:

- Se puedan utilizar tipos de datos predefinidos
- Todas las variables se declaren con un tipo asociado
- Todas las operaciones se especifican indicando los tipos de los operandos requeridos y el tipo del resultado

Tipado dinámico: ligaduras en ejecución, provoca más comprobaciones en tiempo de ejecución (no es seguro???)

Si el lenguaje es fuertemente tipado el compilador puede garantizar la ausencia de errores de tipo en los programas (Ghezzi)

Un lenguaje se dice fuertemente tipado (type safety) si el sistema de tipos impone restricciones que aseguran que no se producirán errores de tipo en ejecución

Un lenguaje se dice fuertemente tipado (type safety) si todos los errores de tipo se detectan

En esta concepción, la intención es evitar los errores de aplicación y son tolerados los errores del lenguaje, detectados tan pronto como sea posible

Reglas de equivalencia y conversión

Tipo compatible: reglas semánticas que determinan si el tipo de un objeto es válido en un contexto particular

Un lenguaje debe definir en qué contexto el tipo Q es compatible con el tipo T

Equivalencia por nombre: dos variables son del mismo tipo si y sólo si están declaradas juntas o si están declaradas con el mismo nombre de tipo

Equivalencia por estructura: dos variables son del mismo tipo si los componentes de su tipo son iguales

Un tipo es compatible con otro si:

1. Es equivalente
2. O se puede convertir

Coerción: significa convertir un valor de un tipo a otro

Reglas del lenguaje de acuerdo al tipo de los operandos y a la jerarquía

Estrategias de conversión

- Widening (ensanchar): cada valor del dominio tiene su correspondiente valor en el rango (entero a real el 1 es 1.0)
- 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 entero)
- Cláusula de casting: conversiones explícitas, se fuerza a que se convierta

Reglas de inferencia de tipo

La inferencia de tipos permite que el tipo de una entidad declarada se “infiera” en lugar de ser declarado

La inferencia puede realizarse de acuerdo al tipo de:

- Un operador predefinido: fun f1(n, m)=(n mod m=0)
- Un operando: fun f2(n) = (n*2)
- Un argumento: fun f3(n:int) = n*n
- El tipo del resultado: fun f4(n):int = (n*n)

Nivel de polimorfismo del lenguaje

Un lenguaje se dice monomórfico si cada entidad se liga a un único tipo (estáticos)

Un lenguaje se dice polimórfico si las entidades pueden estar ligadas a más de un tipo

Las variables polimórficas pueden tomar valores de diferentes tipos

Las operaciones polimórficas son funciones que aceptan operandos de varios tipos

Los tipos polimórficos tienen operaciones polimórficas

Polimorfismo ad-hoc

Permite que una función se aplique a distintos tipos con un comportamientos sustancialmente diferente en cada caso

Sobrecarga

Se utiliza el término sobrecarga para referirse a conjuntos de abstracciones diferentes que están ligadas al mismo símbolo o identificador

Coerción

Permite que un operador que espera un operando de un determinado tipo T puede aplicarse de manera seguro sobre un operando de un tipo diferente al esperado

Polimorfismo universal

Permite que una única operación se aplique uniformemente sobre un conjunto de tipos relacionados

Polimorfismo paramétrico

Si la uniformidad de la estructura de tipos está dada a través de parámetros, hablamos de polimorfismo paramétrico

Un tipo parametrizado es un tipo que tiene otros tipos como parámetros

List<T> = T*

Polimorfismo por inclusión

Es otra forma de polimorfismo universal que permite modelar subtipos y herencia

Si un tipo se define como un conjunto de valores y un conjunto de operaciones. Un subtipo T' de un tipo T puede definirse como un subconjunto de los valores de T y el mismo conjunto de operaciones

El mecanismo de herencia permite definir una nueva clase derivada a partir de una clase base ya existente. Podría agregar atributos y comportamiento

Clase 9 - Estructuras de control

Los lenguajes de programación permiten estructurar al código en relación al flujo de control entre los diferentes componentes de un programa a través de estructuras de control

Son el medio por el cual los programadores pueden determinar el flujo de ejecución entre los componentes de un programa

A nivel de unidad

Cuando el flujo de control se pasa entre unidades (rutinas, funciones, proc., etc.)

Interviene:

- Pasajes de parámetros
- call-return
- Excepciones
- otros

A nivel de sentencia

Secuencia

Es el flujo de control más simple

Ejecución de una sentencia a continuación de otra

El delimitador más general y más usado es el ";"

Hay lenguajes que NO tienen delimitador. Estos establecen que por cada línea haya sólo 1 instrucción

Se los llaman orientados a línea

Otros lenguajes permiten sentencias compuestas

Se pueden agrupar varias sentencias en una con el uso de delimitadores (begin y end, { })

Asignación

Sentencia que produce cambios en los datos de la memoria (x = a + b)

Asigna al l-valor de un objeto de dato (x), el r-valor de una expresión (a + b)

Nota: en la diapositiva dice l-valor, pero creo q sería r-valor, porque el r-valor de la expresión a+b se va a asignar a la posición de memoria (que no cambia) donde se encuentra x, es decir, a la posición donde apunta el l-valor de x, es decir, al r-valor de x

En cualquier lenguaje convencional, existe diferencia entre sentencia de asignación y expresión

Las sentencias de asignación devuelven el valor de la expresión y modifican la posición de memoria

En otros lenguajes tales como C definen la sentencia de asignación como una expresión con efectos colaterales

Selección

If

Estructura de control que permite expresar una elección entre un cierto número posible de sentencias alternativas (ejecución condicional)

Con el if else se pueden tomar dos caminos

Si se anidan los ifs se pueden tomar más caminos, pero no debe ser ambigua la ejecución, se debe saber que else corresponde a que if

Se pueden usar sentencias para darle cierre a los if (end, fi, end if)

Desventajas:

- Ilegibilidad, programas con muchos if anidados pueden ser ilegibles

Solución:

- En algunos lenguajes se utiliza una instrucción compuesta begin y end o {}. Es un bloque de instrucciones acotadas entre las palabras begin y end o {}

Operador ternario:

```
a > 2 ? a = 2 : a = 4
```

Es equivalente a:

```
if (a > 2){
        a = 2
} else {
        a = 4
}
```

Es equivalente a (Python):

```
a = 2 if a > 2 else a = 4
```

Circuito corto

Si hay una expresión lógica formada por &&, si la primera parte de la expresión es falsa, no es necesario evaluar la segunda parte, puesto que el resultado de la expresión conjunta será falso sin importar la evaluación del la segunda parte

Si hay una expresión lógica formada por ||, si la primera parte de la expresión es verdadera, no es necesario evaluar la segunda parte, puesto que el resultado de la expresión conjunta será verdadero sin importar la evaluación del la segunda parte

Permite evitar errores en casos donde por ejemplo hago:

```
if(b && b.isOld())
```

En ese caso, si b es null, no se va a intentar enviarle el mensaje isOld(), por lo que no va a causar error

Selección múltiple

Los lenguajes incorporan distintos tipos de sentencias de selección para poder elegir entre dos o más opciones posibles

```
case variable_ordinal of
        valor1: sentencia 1;
        valor2: sentencia2;
        valor3: sentencia3;
else
        sentencia4;
```

end;

Si el else es opcional (como en pascal), puede traer efectos colaterales

Iteración

La iteración permite que una serie de acciones se ejecuten repetidamente (loop)

La mayoría de los lenguajes de programación proporcionan diferentes tipos de construcciones de bucle para definir la iteración de acciones (llamado el cuerpo del bucle)

Comúnmente agrupados como:

- Tipo bucle for: bucles en los que se conoce el número de repeticiones al inicio del bucle (se repiten un cierto número de veces)
- Tipo bucle while: bucles en los que el cuerpo se ejecuta repetidamente siempre que se cumpla una condición
    - While: while condición do sentencia (se ejecuta sentencia 0 o más veces)
    - Do while: do sentencia while condición (se ejecuta sentencia 1 o más veces)

Clase 10 - Excepciones

Los programadores a menudo escriben programas bajo la suposición optimista de que nada saldrá mal cuando se ejecute el programa. Pero esto no es así

Ejemplos:

- Desbordamiento de pila
- División por cero
- Acceso no autorizado a memoria
- índices fuera de rango

Estas cuestiones pueden ser atendidas por:

- Mecanismos del lenguaje (construcciones especializadas, excepciones)
- Mecanismos de hardware (interrupciones)
- Mecanismos del Sistema Operativo (comunicación entre procesos (IPC), señales)

Definición

Condición inesperada o inusual, que ocurre durante la ejecución del programa y no puede ser manejada en el contexto local

Denota un comportamiento anómalo e indeseable que es necesario controlarlo

Supuestamente ocurre raramente pero, en realidad, suelen ocurrir con frecuencia

La excepción interrumpe el flujo normal de ejecución y ejecuta un controlador de excepciones registrado previamente

Los lenguajes pueden proporcionar estas facilidades, pero no todos funcionan igual

Para que un lenguaje trate excepciones debe proveer mínimamente:

- Un modo de definirlas
- Una forma de reconocerlas
- Una forma de lanzarlas y capturarlas
- Una forma de manejarlas especificando el código y respuestas
- Un criterio de continuación

Controlador/manejador de excepciones

Es una sección de código que se encarga de manejar una excepción en un programa

Su objetivo principal es proporcionar una forma de recuperarse de un error o falla, permitiendo que el programa continúe ejecutándose en lugar de detenerse abruptamente

Encargado de manejar la excepción

Puede tomar distintas acciones según la situación:

- Imprimir un mensaje de error
- Realizar acciones correctivas
- Lanzar otra excepción
- Finalizar la ejecución del programa

Debería elegir la solución menos perjudicial

Tipos de excepciones:

- Implícitas: definidas por el lenguaje (built-in)
- Explícitas: definidas por el programador

Los lenguajes deben proveer instrucciones para:

- Definición de una excepción
- Levantamiento de una excepción
- Manejador de una excepción

Punto de retorno

Después de atender a una excepción, el punto de retorno dependerá del flujo de ejecución del programa y de cómo se haya diseñado el manejo de excepciones en el código

También va a depender del lenguaje

Se puede tener en cuenta:

- Continuar la ejecución normal del programa:
    - Si después de manejar una excepción el programa puede continuar la ejecución del código restante sin problemas
    - El punto de retorno será definido por el lenguaje (por ejemplo, el siguiente bloque de código después del bloque de manejo de excepciones o siguiente instrucción)

- Retornar a un estado anterior:
    - Cuando el manejo de excepciones puede requerir que el programa regrese a un estado anterior o deshaga acciones realizadas antes de que se produjera la excepción
- Propagar la excepción:
    - En el controlador de excepciones no puede manejar completamente la excepción, puede optar por propagarla a un nivel superior en la jerarquía de llamadas
    - El punto de retorno sería el controlador de excepciones en el nivel superior que pueda manejar la excepción o decidir cómo manejarla
- Terminar la ejecución del programa:
    - En situaciones excepcionales o críticas, es posible que el controlador de excepciones determine que no se puede continuar ejecutando el programa de manera segura
    - El punto de retorno puede ser la finalización del programa o alguna acción específica de cierre antes de la finalización

Modelos de manejo de excepciones

Reasunción

Se refiere a la posibilidad de retomar la ejecución normal del programa después de manejar una excepción

El controlador de excepciones realiza las acciones necesarias para manejar la excepción (medidas correctivas) y luego el programa continúa su ejecución a partir de la sentencia siguiente a donde se produjo la excepción

Terminación

El controlador de excepciones realiza las acciones necesarias para manejar la excepción, pero no se retorna al punto donde se produjo la excepción (invocador), continúa su ejecución a partir de la finalización del manejador

Lenguajes

PL/I

Utiliza el criterio reasunción

Las excepciones se llaman CONDITIONS en PL/1

Los manejadores se declaran con la sentencia ON:

- ON CONDITION(Nombre-excepción)

El Manejador puede ser una instrucción o un bloque (entre begin y end)

Las excepciones se lanzan explícitamente con la palabra clave SIGNAL:

- SIGNAL CONDITION(Nombre-excepción)

Este lenguaje tiene una serie de excepciones ya predefinidas con su manejador asociado. Son las Built-in exceptions. Por ej. zerodivide, que se levanta cuando hay una división por cero

Los manejadores se ligan dinámicamente con las excepciones. Una excepción siempre estará ligada con el último manejador definido. (Manejo de pila de manejadores de excepciones)

El alcance de un manejador comienza en la siguiente línea a donde fue declarado y termina cuando finaliza la ejecución de la unidad donde fue declarado

ADA

Utiliza el criterio terminación

Cada vez que se produce una excepción, se termina el bloque dónde se levantó y se ejecuta el manejador asociado, y continúa luego

Las excepciones se definen/declaran en la zona de definición de variables y tienen el mismo alcance que las variables convencionales

Su formato para declarar es MiExcepcion: exception

La lista de controladores de excepciones lleva el prefijo de la palabra clave exception

Cada controlador lleva el prefijo de la palabra clave when (con un formato específico), seguido de las acciones

Se puede utilizar when others para capturar cualquier excepción no especificada:

- Debe colocarse al final del bloque de manejo de excepciones
- Posee efectos colaterales

Las excepciones se levantan explícitamente con la palabra clave raise

Los manejadores pueden agregarse y encontrarse al final de diferentes unidades de programa: subprograma, bloque, procedimiento, paquete o tarea que maneja la excepción

Tiene varias excepciones predefinidas built-in:

- Constraint_Error: cuando se intenta violar una restricción impuesta en una declaración (por ejemplo, indexar más allá de los límites de un array o asignar a una variable un valor fuera del rango de su subtipo o dividir por cero)
- Program_Error: cuando se intenta violar la estructura de control o regla del lenguaje (por ejemplo, una función termina sin devolver un valor)
- Storage_Error: cuando se produce una violación de memoria (por ejemplo, cuando se requiere más memoria de la disponible)
- Tasking_Error: cuando hay errores en la comunicación y manejo de tareas del sistema (por ejemplo, en concurrencia y la programación de tareas /threads)
- Name_Error: cuando hay error de nombre (por ejemplo, se produce cuando se intenta abrir un fichero que no existe)

Propagación

Si la unidad que genera la excepción proporciona un manejador para la misma, el control se transfiere inmediatamente a ese manejador:

- Se omiten las acciones que siguen al punto en el que se generó la excepción
- Se ejecuta el manejador
- Luego el programa continúa su ejecución normalmente, desde la instrucción que sigue al manejador

Si la unidad que genera la excepción no proporciona un manejador, se debe buscar ese manejador dinámicamente:

- Se termina la unidad (bloque, paquete, subprograma o tarea) dónde se produce la excepción
- Si el manejador no se encuentra en ese lugar la excepción se propaga dinámicamente (quién lo llamó). Esto significa que se vuelve a levantar en otro ámbito

- Siempre teniendo en cuenta el alcance, puede convertirse en anónima (esto se daría en caso de que la excepción que se está propagando ya no tenga alcance donde se propagó). Al propagarse a otras unidades la variable excepción declarada ya no está en el alcance y quedará sin nombre y entrará por when others

Uso del raise

La utilidad de raise es poder lanzar excepciones que pueden ser definidas por el programador

Una excepción se puede levantar nuevamente colocando solo la palabra raise

¿Para qué? el manejador podría realizar algunas acciones de recuperación y luego utilizar raise para volver a lanzar la excepción y permitir que se propague más arriba en la jerarquía de manejo de excepciones

C++

Utiliza el criterio terminación

Try para indicar los bloques donde pueden llegar a levantarse excepciones

Catch se utiliza para especificar los manejadores: catch(NombreDeLaExcepción)

Las cláusulas catch deben estar después del bloque try y antes de cualquier código que esté fuera del bloque try

Los manejadores van asociados a bloques { }

Throw Se utiliza para lanzar explícitamente una excepción

Funcionamiento:

- El bloque try que contiene código que puede lanzar una excepción
- Si se lanza una excepción el control se transfiere inmediatamente a la cláusula catch
- Si la excepción coincide con el tipo especificado en la cláusula catch, se ejecuta el bloque de código de esa cláusula catch
    - Si la excepción se maneja exitosamente, la ejecución continúa después del bloque try-catch
- Si no se encuentra un catch correspondiente o no se maneja la excepción, la excepción puede propagarse hacia bloques try-catch externos

- Sino puede resultar en una finalización abrupta del programa

Permite que el programador especifique de manera precisa la intención de una rutina, al especificar tanto el comportamiento normal esperado (los datos que puede aceptar y devolver) como sus comportamientos anormales pasándole los parámetros al throw

Ejemplo:

- void rutina ( ) throw (Ayuda, Zerodivide);
- rutina se declara como una función void que no devuelve ningún valor y que puede lanzar 2 tipos de excepciones: Ayuda y Zerodivide
- Si lanzó otra excepción que no está contemplada en el listado de la Interface:
    - No se propaga la excepción y una función especial unexpected() se ejecuta automáticamente, que eventualmente causa abort() que provoca el final del programa
    - Unexpected() puede ser redefinida por el programador
- Si colocó en su interface el listado de posibles excepciones a alcanzar
    - Sí se propaga la excepción
- Si no se proporciona ninguna lista throw Significa que **cualquier** excepción puede ser propagada
- Si se colocó en su interface una lista vacía throw() Significa que **ninguna** excepción será propagada

Si una excepción se propaga repetidamente y nunca se encuentra un manejador coincidente, llama automáticamente a una función especial llamada terminate()

terminate() puede ser redefinido por el programador. Su comportamiento predeterminado eventualmente aborta la ejecución del programa

CLU

Utiliza el criterio de Terminación

Solamente pueden ser lanzadas por los procedimientos

Si una instrucción genera una excepción, el procedimiento que contiene la instrucción retorna anormalmente al generar la excepción

Un procedimiento no puede manejar una excepción generada por su ejecución, quien llama al procedimiento debe encargarse de manejarla

Las excepciones que un procedimiento puede lanzar se declaran en su encabezado con la palabra clave signals

Se lanzan explícitamente con la palabra clave signal

Los manejadores se colocan al lado de una sentencia simple o compleja y llevan la palabra clave when

Forma de definirlos: <instruccion> except <lista_de_controladores> end

Donde <instruccion> puede ser cualquier instrucción (compuesta) del lenguaje

Si la ejecución de una invocación de procedimiento dentro de <instruccion> genera una excepción, el control se transfiere a <lista_de_controladores>

Posee excepciones predefinidas con su manejador asociado

Se pueden pasar parámetros a los manejadores

Una excepción se puede volver a levantar una sola vez utilizando resignal

Si no encuentra el manejador al producirse una excepción:

- Se termina el procedimiento donde se levantó la excepción y devuelve el control al llamante inmediato donde se debe encontrar el manejador
- Si el manejador se encuentra en ese ámbito, se ejecuta y luego se pasa el control a la sentencia siguiente a la que está ligado dicho manejador
- Si el manejador no se encuentra en ese lugar la excepción se propaga estáticamente en las sentencias asociadas. Esto significa que el proceso se repite para las sentencias incluidas estáticamente
- En caso de no encontrar ningún manejador en el procedimiento que hizo la llamada se levanta una excepción failure y devuelve el control, terminando todo el programa

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

Cada excepción está representada por una instancia de la clase Throwable o de una de sus subclases (Error y Exception)

La gestión de excepciones se lleva a cabo mediante cinco palabras clave: try, catch, throw, throws, finally

Se debe especificar mediante la cláusula throws cualquier excepción que se envía desde un método

Se debe poner cualquier código que el programador desee que se ejecute siempre, en el método finally

Fases del tratamiento de excepciones

- Detectar e informar del error:
    - Lanzamiento de Excepciones → throw
    - Un método detecta una condición anormal que le impide continuar con su ejecución y finaliza “lanzando” un objeto Excepción
- Recoger el error y tratarlo:
    - Captura de Excepciones → bloque try-catch
    - Un método recibe un objeto Excepción que le indica que otro método no ha terminado correctamente su ejecución y decide actuar en función del tipo de error

Python

Se pueden levantar excepciones explícitamente con “raise”

Se manejan a través de bloques try except

La declaración try funciona de la siguiente manera:

- Primero, se ejecuta el bloque try (el código entre las declaraciones try y except)
- Si no ocurre ninguna excepción, el bloque except se saltea y termina la ejecución de la declaración try (en caso de estar definido el bloque opcional else se ejecuta)
- Si ocurre una excepción durante la ejecución del bloque try, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada except, se ejecuta el bloque except, y la ejecución continúa luego de la declaración try
- Si ocurre una excepción que no coincide con la excepción nombrada en el except, esta se pasa a declaraciones try de más afuera; si no se encuentra nada que la maneje, es una excepción no manejada, y la ejecución se frena con un mensaje de error

¿Qué sucede cuando una excepción no encuentra un manejador en su bloque “try except”?:

- Busca estáticamente Analiza si ese try está contenido dentro de otro y si ese otro tiene un manejador para esa excepción
- Sino busca dinámicamente: analiza quién lo llamó y busca allí
- Si no se encuentra un manejador, se corta el proceso y larga el mensaje standard de error

php

Modelo de Terminación

Una excepción puede ser lanzada (thrown), y atrapada ("catched")

El código está dentro de un bloque try

Cada bloque try debe tener al menos un bloque catch correspondiente

Las excepciones pueden ser lanzadas (o relanzadas) dentro de un bloque catch

Se puede utilizar un bloque finally después de los bloques catch

El objeto lanzado debe ser una instancia de la clase Exception o de una subclase de Exception

Intentar lanzar un objeto que no lo es resultará en un Error Fatal de PHP

Cuando una excepción es lanzada, el código siguiente a la declaración no será ejecutado, y PHP intentará encontrar el primer bloque catch coincidente

Si una excepción no es capturada, se emitirá un Error Fatal de PHP con un mensaje "Uncaught Exception ..." ("Excepción No Capturada"), a menos que se haya definido un gestor con set_exception_handler()

Clase 11 - Paradigmas de Programación - Al fin, que materia de mierda

Un paradigma de programación es un estilo de desarrollo de programas, un modelo para resolver problemas computacionales

Los lenguajes de programación, necesariamente, se encuadran en uno o varios paradigmas a la vez, a partir del tipo de órdenes que permiten implementar, tiene una relación directa con su sintaxis

Principales paradigmas

- Imperativo: sentencias + secuencias de comandos
- Declarativo: los programas describen los resultados esperados sin listar explícitamente los pasos a llevar a cabo para alcanzarlos
    - Lógico. Aserciones lógicas: hechos + reglas, es declarativo
- Funcional: los programas se componen de funciones, pero no son como las funciones del paradigma imperativo, sino que son funciones matemáticas
- Orientado a Objetos : métodos + mensajes
- Dirigido por eventos: el flujo del programa está determinado por sucesos externos (por ejemplo, una acción del usuario)
- Orientado a aspectos: apunta a dividir el programa en módulos independientes, cada uno con un comportamiento y responsabilidad bien definido

Programación lógica

La programación lógica es un tipo de paradigma de programación dentro del paradigma de programación declarativa

Es un paradigma en el cual los programas son una serie de aserciones lógicas

El conocimiento se representa a través de reglas y hechos

Los objetos son representados por términos, los cuales contienen constantes y variables

PROLOG es el lenguaje lógico más utilizado

Elementos

La sintaxis básica es el “término”

- Variables:
    - Se refieren a elementos indeterminados que pueden sustituirse por cualquier otro. Ejemplo: “humano(X)”, la X puede ser sustituída por constantes como: juan, pepe, etc.
    - Los nombres de las variables comienzan con mayúsculas y pueden incluir números
- Constantes:
    - A diferencia de las variables son elementos determinados. “humano(juan)”
    - La constantes son string de letras en minúsculas (representan objetos atómicos) o string de dígitos (representan números)
- Término compuesto:
    - Consisten en un ”functor” seguido de un número fijo de argumentos encerrados entre paréntesis, los cuales son a su vez términos
    - Se denomina “aridad” al número de argumentos
    - Se denomina “estructura” (ground term) a un término compuesto cuyos argumentos no son variables
- Listas:
    - La constante [] representa una lista vacía
    - El functor “.” construye una lista a partir de un elemento y una lista
    - Ejemplo: .(alpha, []), representa una lista que contiene un único elemento que es alpha. Otra manera de representar la lista es usando [] en lugar de .(). La lista del ejemplo anterior quedaría: [alpha, []]
    - Y también se representa utilizando el símbolo | [alpha|[]]
    - La notación general para denotar lista es : [X|Y] X es el elemento cabeza de la lista e Y es una lista, que representa la cola de la lista que se está modelando

Clausulas de Horn

Un programa escrito en un lenguaje lógico es una secuencia de “cláusulas”
Las cláusulas pueden ser: un “Hecho” o una “Regla”

- Hecho:

    - Expresan relaciones entre objetos
    - Expresan verdades
    - Son expresiones del tipo p(t1, t2, ….tn)
    - Ejemplos:
        - tiene(coche, ruedas) representa el hecho que un coche tiene ruedas
        - longuitud([], 0) representa el hecho que una lista vacía tiene longuitud cero
        - moneda(peso) representa el hecho que peso es una moneda
- Regla:
    - Tiene la forma: conclusión :- condición
    - Dónde:
        - ‘:-’ indica “Si”
        - Conclusión es un simple predicado y condición es una conjunción de predicados, separados por comas que representan un AND lógico
    - En un lenguaje procedural una regla la podríamos representar como: if condición else conclusión

Programas y querys

Programa: conjunto de cláusulas. Ejemplo:

```
longuitud ([],0)
longuitud ([X|Y],N) :- longuitud(Y, M), N=M + 1
```

Query: representa lo que deseamos que sea contestado. Ejemplo:

```
longuitud([rojo| [verde | [azul | [] ] ] ] ,X)
```

Ejecución:

```
?-longuitud([rojo| [verde | [azul | [] ] ] ] ,X)
longuitud([verde | [azul | [] ] ],M) y X=M+1
longuitud([azul | [] ] ,Z) y M=Z+1
longuitud([],T) y Z=T+1
T=0 => Z=1 => M=2 => X=3
```

Ejecución de programas

Un programa es un conjunto de reglas y hechos que proveen una especificación declarativa de que es lo que se conoce y la pregunta es el objetivo que queremos alcanzar

La ejecución de dicho programa será el intento de obtener una respuesta

Desde un punto de vista lógico la respuesta a esa pregunta va a ser “YES” si la pregunta puede ser derivada aplicando “deducciones” del conjunto de reglas y hechos dados

Ejemplo de un programa que describe una relación binaria (rel) y su cierre (clos):

Programa:

1. rel(a,b)
2. rel(a,c)
3. rel(b,f)
4. rel(f,g)
5. clos(X,Y) :- rel(X, Y)
6. clos(X,Y) : - rel(X, Z), clos(Z, Y)

Query:

?-clos(a,f)

Ejecución:

1. Se intenta clos(a, f) según la regla de la línea 5, pero no existe ningún hecho que sea rel(a, f), entonces…
2. Se prueba clos(a, f) según la regla de la línea 6:
    a. La regla queda: clos(a, f) :- rel(a, Z), clos(Z, f)
    b. Si se reemplaza Z por b, tanto rel(a, b) como clos(b, f), son hechos que existen y el programa retorna YES

Programación orientada a objetos

“Un programa escrito con una lenguaje OO es un conjunto de **objetos** que **interactúan** mandándose **mensajes**”

Los elementos que intervienen en la programación OO son:

- Objetos:
    - Entidades que poseen estado interno y comportamiento

    - Es el equivalente a un dato abstracto
- 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
    - La información contenida en el objeto solo puede ser accedida por la ejecución de los métodos correspondientes
    - Instancia de una 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

Generalización/Especificación (Herencia)

El segundo nivel de abstracción consiste en agrupar las clases en jerarquías de clases (definiendo sub y super clases), de forma tal que una clase A herede todas las propiedades de su superclase B (suponiendo que tiene una)

Polimorfismo

Es la capacidad que tienen los objetos de distintas clases de responder a mensajes con el mismo nombre

Binding dinámico

Es la vinculación en el proceso de ejecución de los objetos con los mensajes

Paradigma Aplicativo o Funcional

Basado en el uso de funciones

Muy popular en la resolución de problemas de inteligencia artificial, matemática, lógica, procesamiento paralelo

Ventajas:

- Vista uniforme de programa y función
- Tratamiento de funciones como datos
- Liberación de efectos colaterales
- Manejo automático de memoria

Desventaja: ineficiencia de ejecución

Características

- Proveen un conjunto de funciones primitivas
- Proveen un conjunto de formas funcionales
- Semántica basada en valores
- Transparencia referencial (una función siempre da el mismo valor ante la misma entrada)
- Regla de mapeo basada en combinación o composición
- Las funciones de primer orden (no reciben a otras funciones como parámetro. Las de alto orden si lo hacen)

Funciones

El valor más importante en la programación funcional es el de una función

Matemáticamente una función es una correspondencia : f: A → B. A cada elemento de A le corresponde un único elemento en B

f(x) denota el resultado de la aplicación de f a x

Las funciones son tratadas como valores, pueden ser pasadas como parámetros, retornar resultados, etc.

Definición:

Se debe distinguir entre el valor y la definición de una función

Existen muchas maneras de DEFINIR una misma función, pero siempre dará el mismo valor, ejemplo:

DOBLE X = X + X

DOBLE' X= 2 * X

Denotan la misma función pero son dos formas distintas de definirlas

Tipo de una función:

Puede estar definida explícitamente dentro del script, por ejemplo:

cuadrado::num→ num (define el tipo)

cuadrado x= x * x (definición)

O puede deducirse/inferirse el tipo de una función

Expresiones y valores

La expresión es la noción central de la programación funcional

Característica más importante: "Una expresión es su valor"

El valor de una expresión depende **únicamente** de los valores de las sub expresiones que la componen

Las expresiones también pueden contener variables (valores desconocidos)

La noción de variable es la de "variable matemática", no la de celda de memoria

Las expresiones cumplen con la propiedad de "transparencia referencial", es decir, dos expresiones sintácticamente iguales darán el mismo valor

No existen efectos colaterales (no hay variables globales ni pasaje por referencia ya que ambas características pueden producir efectos colaterales)

Un script es una lista de definiciones que pueden someterse a evaluación

Ejemplo:

?cuadrado (3 + 4)

49

Algunas expresiones pueden no llegar a reducirse del todo, ejemplo: 1/0

A esas expresiones se las denominan canónicas, pero se les asigna un valor indefinido y corresponde al símbolo bottom(^)

Por lo tanto toda expresión siempre denota un valor

La forma de evaluar es a través de un mecanismo de reducción o simplificación
Ejemplo: cuadrado (3 + 4)

=> cuadrado 7 (se aplicó +)
=> 7 * 7 (se aplicó cuadrado)
=> 49 (se aplicó *)

Otra forma sería: cuadrado (3 + 4)

=> (3 + 4) * (3 + 4) (se aplicó cuadrado)
=> 7 * (3 + 4) (se aplicó +)
=> 7 * 7 (se aplicó +)
=> 49 (se aplicó *)

No importa la forma de evaluarla, siempre el resultado final será el mismo

Formas de reducción

- Orden aplicativo: aunque no lo necesite siempre evalúa los argumentos
- Orden normal (lazy evaluation):
    - No calcula más de lo necesario
    - La expresión no es evaluada hasta que su valor se necesite
    - Una expresión compartida no es evaluada más de una vez
    - Esto aplica por ejemplo a un && donde si el primer operando es falso el resto no se evalúa (cortocircuito)

Tipos

- Básicos: son los primitivos, ejemplo:
    - num(int y float) (números)
    - bool(valores de verdad)
    - char(caracteres)
- Derivados: se construyen de otros tipos, ejemplo:
    - (num,char) tipo de pares de valores
    - (num→char) tipo de una función (toda función tiene asociado un tipo)

Expresiones polimórficas

En algunas funciones no es tan fácil deducir su tipo

Ejemplo: id x = x

Esta función es la función Identidad

Su tipo puede ser de char→ char, de num→ num, etc.

Por lo tanto su tipo será de β→β

Se utilizan letras griegas para tipos polimórficos

Otro ejemplo: letra x = “A”

Su tipo será β→ char

Currificación

Mecanismo que reemplaza argumentos estructurados por argumentos más simples

Ejemplo: sean dos definiciones de la función “Suma”:

1. Suma(x,y) = x + y
2. Suma’ x y = x + y (por cada valor de x devuelve una función)

El tipo de Suma es : (num,num)→ num

El tipo de Suma’ es : num→ (num→ num)

Aplicando la función:

1. Suma(1,2)→ 3
2. Suma’ 1 2 => Suma’1 aplicado al valor 2→ 3

Cálculo Lambda

Es un modelo de computación para definir funciones

Se utiliza para entender los elementos de la programación funcional y la semántica subyacente, independientemente de los detalles sintácticos de un lenguaje de programación en particular

Es un modelo de programación funcional que se independiza de la sintaxis del lenguaje de programación

Las expresiones del Lambda cálculo pueden ser de 3 clases:

- Un simple identificador o una constante. Ej: x, 3
- Una definición de una función. Ej: λ x. x+1

- Una aplicación de una función. La forma es (e1 e2), dónde se lee e1 se aplica a e2
- Ejemplo de la funcion cubo(x) = x * x * x:
    - λ x. x * x * x
    - Evaluación: λ (x. x * x * x)2) se evalúa con 2 y da como resultado 8