Conceptos de Paradigmas y Lenguajes de Programación

Resumen teorías

Clase 1 - Sintaxis

Idea principal

Introducir, analizar y evaluar los conceptos más importantes de los lenguajes de programación

Criterios para evaluar los lenguajes de programación

Simplicidad y legibilidad

Los lenguajes de programación deberían:

- Poder producir programas fáciles de escribir y de leer
- Resultar fáciles a la hora de aprenderlo o enseñarlo

Ejemplo de cuestiones que atentan contra esto:

- Muchos componentes elementales
- Conocer subconjuntos de componentes
- El mismo concepto semántico pero distinta sintaxis
- Distintos conceptos semánticos pero la misma notación sintáctica
- Abuso de operadores sobrecargados

Claridad en los bindings

Los elementos de los lenguajes de programación pueden ligarse a sus atributos o propiedades en diferentes momentos:

- Definición del lenguaje
- Implementación del lenguaje
- En escritura del programa
- Compilación
- Cargado del programa
- En ejecución

La ligadura en cualquier caso debe ser clara

Confiabilidad

La confiabilidad está relacionada con la seguridad:

- Chequeo de tipos: cuanto antes se encuentren errores menos costoso resulta realizar los arreglos que se requieran
- Manejo de excepciones: la habilidad para interceptar errores en tiempo de ejecución, tomar medidas correctivas y continuar

Soporte

Debería ser accesible para cualquiera que quiera usarlo o instalarlo

Lo ideal sería que su compilador o intérprete sea de dominio público

Debería poder ser implementado en diferentes plataformas

Deberían existir diferentes medios para poder familiarizarse con el lenguaje: tutoriales, cursos textos, etc.

Abstracción

Capacidad de definir y usar estructuras u operaciones complicadas de manera que sea posible ignorar muchos de los detalles

Abstracción de procesos y de datos

Ortogonalidad

Significa que un conjunto pequeño de constructores primitivos puede ser combinado en número relativamente pequeño a la hora de construir estructuras de control y datos

Cada combinación es legal y con sentido

El usuario comprende mejor si tiene un pequeño número de primitivas y un conjunto consistente de reglas de combinación

Eficiencia

En tiempo y espacio, de esfuerzo humano. Es optimizable

Sintaxis

Conjunto componer de reglas que definen como componer letras, dígitos y otros caracteres para formar los programas

Proporciona mecanismos para decir si un programa es válido

Características de la sintaxis

La sintaxis debe ayudar al programador a escribir programas correctos sintácticamente

La sintaxis establece reglas que sirven para que el programador se comunique con el procesador

La sintaxis debe contemplar soluciones a características tales como:

- Legibilidad
- Verificabilidad
- Traducción
- Falta de ambigüedad

La sintaxis establece reglas que definen cómo deben combinarse las componentes básicas, llamadas “word”, para formar sentencias y programas

Elementos de la sintaxis

- Alfabeto o conjunto de caracteres
- Identificadores:
    - Elección más ampliamente utilizada: cadena de letras y dígitos, que deben comenzar con una letra
    - Si se restringe la longitud se pierde legibilidad
- Operadores
- Comentarios y uso de blancos
- Palabras clave y palabras reservadas:
    - Palabras clave o keywords son aquellas que tienen un significado dentro de un contexto
    - Palabras reservadas son palabras clave que además no pueden ser usadas por el programador como identificador de otra entidad
    - Ventajas:

- Permiten al compilador y al programador expresarse claramente
- Hacen los programas más legibles y permiten una rápida traducción

Estructura sintáctica

Vocabulario o words

Conjunto de caracteres y palabras necesarias para construir expresiones, sentencias y programas. Ej: identificadores, operadores, palabras claves, etc.

Las words no son elementales se construyen a partir del alfabeto

Expresiones

Son funciones que a partir de un conjunto de datos devuelven un resultado

Son bloques sintácticos básicos a partir de los cuales se construyen las sentencias y programas

Sentencias

Componente sintáctico más importante

Tiene un fuerte impacto en la facilidad de escritura y legibilidad

Hay sentencias simples, estructuradas y anidadas

Reglas léxicas y sintácticas

- Reglas léxicas: conjunto de reglas para formar las “word”, a partir de los caracteres del alfabeto
- Reglas sintácticas: conjunto de reglas que definen cómo formar las “expresiones” y “sentencias”

Tipos de sintaxis

- Abstracta: se refiere básicamente a la estructura
- Concreta: se refiere básicamente a la parte léxica
- Pragmática: se refiere básicamente al uso práctico

Ejemplo de sintaxis concreta y abstracta

C	Pascal

while (x!= y){
<código>
}
	while x<>y do begin
	<código>
	end;

Son diferentes respecto a la sintaxis concreta porque existen diferencias léxicas entre ellas (!= vs <>, {} vs begin end;)

Son iguales respecto a la sintaxis abstracta, ya que ambas tienen la misma estructura: <while> seguido de <condición> seguido de <bloque>

Ejemplo de sintaxis pragmática:

Ej.1: <> es más legible que !=

Ej.2: en C y Pascal {} o begin-end; pueden omitirse si el bloque está compuesto por una sola sentencia

Pragmáticamente puede conducir a error ya que si se necesitara agregar una sentencia debe agregarse el begin-end; o los {} (es decir que no define una única forma de escribirse)

Cómo definir la sintaxis

Se necesita una descripción finita para definir un conjunto infinito (conjunto de todos los programas bien escritos)

Formas para definir la sintaxis:

- Lenguaje natural. Ej.: fortran
- Utilizando la gramática libre de contexto, definida por Backus y Naun (BNF). Ej: algol
- Diagramas sintácticos que son equivalentes a BNF pero mucho más intuitivos

BNF

- Es una notación formal para describir la sintaxis
- Es un metalenguaje
- Utiliza metasímbolos: “<” “>” “::=” “|”
- Define las reglas por medio de “producciones”

Gramática

Conjunto de reglas finita que define un conjunto infinito de posibles sentencias válidas en el lenguaje

Una gramática está formada por una 4-tupla: G = (N, T, S, P)

N

Conjunto de símbolos no terminales

T

Conjunto de símbolos terminales

S

Símbolo distinguido de la gramática que pertenece a N (lo que vamos a definir)

P

Conjunto de producciones

Diagramas sintácticos (Conway)

- Es un grafo sintáctico o carta sintáctica
- Cada diagrama tiene una entrada y una salida, y el camino determina el análisis
- Cada diagrama representa una regla o producción
- Para que una sentencia sea válida, debe haber un camino desde la entrada hasta la salida que la describa

- Se visualiza y entiende mejor que BNF o EBNF

Clase 2 - Semántica

Semántica

Describe el significado de los símbolos, palabras y frases de un lenguaje ya sea lenguaje natural o lenguaje informático que es sintácticamente válido

Conjunto de reglas para dar significado a los programas sintácticamente válidos

Proporciona mecanismos para que, dado un programa válido, la máquina pueda decir que significa

Hay características de la estructura de los lenguajes de programación que son difíciles o imposibles de describir con las gramáticas BNF/EBNF

Semántica estática

Se la llama así porque el análisis para el chequeo se hace en compilación (antes de la ejecución)

No está relacionada con el significado de la ejecución del programa, está más relacionado con las formas válidas (con la sintaxis)

El análisis está ubicado entre el análisis sintáctico y el análisis de semántica dinámica, pero más cercano a la sintaxis

Gramática de atributos

Para describir la sintaxis y la semántica estática formalmente sirven las denominadas gramáticas de atributos, inventadas por Knuth en 1968

Son gramáticas sensibles al contexto (GSC): si se relacionan con el significado

La usan los compiladores, antes de la ejecución

Generalmente resuelven los aspectos de la semántica estática

A las construcciones del lenguaje se les asocia información a través de “atributos” asociados a los símbolos de la gramática (terminales o no terminales), que sirven para detectar errores

Un atributo puede ser: el valor de una variable, el tipo de una variable o expresión, lugar que ocupa una variable en la memoria, dígitos significativos de un número, etc.

Los valores de los atributos se obtienen mediante las llamadas “ecuaciones o reglas semánticas” asociadas a las producciones gramaticales

De la ejecución de las ecuaciones:

- Se ingresan símbolos a la tabla de símbolos
- Detectar y dar mensajes de error
- Detecta dos variables iguales
- Controla tipo y variables de igual tipo
- Ciertas combinaciones no permitidas (reglas específicas del lenguaje)
- Generar un código para el siguiente paso

Semántica dinámica

Es la que describe el significado de ejecutar las diferentes construcciones del lenguaje de programación

Su efecto se ve durante la ejecución del programa

Influirá la interacción con el usuario y errores de la programación

Formales y complejas

- Semántica axiomática
- Semántica denotacional: la usan los diseñadores de compiladores

No formal

Semántica operacional: la usan para manuales de lenguajes

Sirven para comprobar la ejecución, la exactitud de un lenguaje, comparar funcionalidades de distintos programas

Se pueden usar combinados, no todos sirven para todos los tipos de lenguajes de programación

Semántica axiomática

Considera al programa como “una máquina de estados” donde cada instrucción provoca un cambio de estado

Se parte de un axioma (verdad) que sirve para verificar "estados y condiciones" a probar

Los constructores de un lenguaje de programación se formalizan describiendo como su ejecución provoca un cambio de estado (cada vez que se ejecuta)

Se desarrolló para probar la corrección de los programas

La notación empleada es el “cálculo de predicados”

Un estado se describe con un predicado

El predicado describe los valores de las variables en ese estado

Existe un estado anterior y un estado posterior a la ejecución del constructor

Cada sentencia se precede y se continúa con una expresión lógica que describe las restricciones y relaciones entre los datos

Precondiciones (condiciones de estado previo)

Poscondiciones (condiciones de estado posterior)

Semántica denotacional

Se basa en la teoría de funciones recursivas y modelos matemáticos, es más exacto para obtener y verificar resultados, pero es más difícil de leer

Define una correspondencia entre los constructores sintácticos y sus significados

Describe la dependencia funcional entre el resultado de la ejecución y sus datos iniciales

Lo que hace es buscar funciones que se aproximen a las producciones sintácticas

Semántica operacional

El significado de un programa se describe mediante otro lenguaje de bajo nivel implementado sobre una máquina abstracta

Cuando se ejecuta una sentencia del lenguaje de programación los cambios de estado de la máquina abstracta definen su significado

Es un método informal porque se basa en otro lenguaje de bajo nivel y puede llevar a errores

Es el más utilizado en los libros de texto para explicar el significado de los lenguajes

Ejemplo:

Pascal	Máquina abstracta

for i := pri to ul do
begin
  <código>
end;
	i := pri (inicializo i)
	lazo if i > ul goto sal
	  <código>
	  i := i +1
	  goto lazo
	sal

Procesamiento de un programa

Primero: unos y ceros (alto embole y te cagas confundiendo cada 2x3)
Segundo: código mnemotécnico (ADD, SUB, NE, JMP)
Problemas:

- Cada máquina o familia de procesadores tiene su propio SET de instrucciones, entonces cada uno tiene que tener su propio:
    - Lenguaje ensamblador
    - Programa ensamblador
    - Código nemotécnico
- Imposible intercambiar programas entre distintas máquinas o de distintas familias de procesadores
- Diferentes versiones para una misma CPU pueden tener sets de instrucciones incompatibles
- Modelos evolucionados de una familia de CPU pueden incorporar instrucciones nuevas

Tercero: lenguajes de alto nivel (así sí 😎)

¿Cómo pasamos de lenguaje de alto nivel a unos y ceros?: con programas traductores (intérpretes, compiladores)

Interpretación

Hay un Programa que está escrito en lenguaje de programación interpretado

Hay un Programa llamado Intérprete que realiza la traducción de ese lenguaje interpretado en el momento de ejecución

El proceso que realiza cuando se ejecuta sobre cada una de las sentencias del programa es:

1. Leer
2. Analizar
3. Decodificar
4. Ejecutar
5. Vuelta a 1

Por cada una de las instrucciones del programa

Ventaja: solo pasa por algunas instrucciones, no por todo el programa (varía según la ejecución)

Desventaja: cada vez que vuelvo a ejecutar el programa se repite toda la secuencia

Compilación

Tenemos nuestro programa escrito en un lenguaje de alto nivel de este tipo

Hay un programa llamado compilador que realiza la traducción a lenguaje de máquina

Se traduce/compila antes de ejecución

Pasa por todas las instrucciones antes de la ejecución

Ventaja: el código que se genera se guarda y se puede reusar ya compilado

El compilador toma todo el programa escrito en un lenguaje de alto nivel que llamamos lenguaje fuente antes de su ejecución

Luego de la compilación va a generar:

- O un lenguaje objeto que es generalmente el ejecutable (en lenguaje de máquina)
- O un lenguaje de nivel intermedio (lenguaje ensamblador)

Comparación entre compilador e intérprete

Por cómo se ejecuta

Intérprete:

- Se utiliza el intérprete en la ejecución
- Ejecuta el programa línea por línea
- Por donde pase dependerá de la acción del usuario, de la entrada de datos y/o de alguna decisión del programa
- Siempre se debe tener el programa interprete
- El programa fuente será público (necesito ambos)

Compilador:

- Se utiliza el compilador antes de la ejecución
- Produce un programa ejecutable equivalente en lenguaje objeto
- El programa fuente no será público
- Una vez compilado no necesito tener el programa compilador

Por el orden de ejecución

Intérprete: sigue el orden lógico de ejecución (no necesariamente recorre todo el código)

Compilador: sigue el orden físico de las sentencias (recorre todo)

Por el tiempo consumido de ejecución

Intérprete:

- Por cada sentencia que pasa realiza el proceso de decodificación (lee, analiza y ejecuta) para determinar las operaciones y sus operandos
- Es repetitivo
- Si la sentencia está en un proceso iterativo (ej.: for/while), se realizará la tarea de decodificación tantas veces como sea requerido
- La velocidad de proceso se puede ver afectada por esto

Compilador:

- Pasa por todas las sentencias

- No repite lazos
- Traduce todo de una sola vez
- Genera código objeto ya compilado
- La velocidad de compilar dependerá del tamaño del código

Por la eficiencia posterior

Intérprete:

- Más lento en ejecución
- Se repite el proceso cada vez que se ejecuta el mismo programa o pasa por las mismas instrucciones
- Para ser ejecutado en otra máquina se necesita tener si o si el intérprete instalado
- El programa fuente será público

Compilador:

- Más rápido ejecutar desde el punto de vista del hardware porque ya está en un lenguaje de más bajo nivel
- Detectó más errores al pasar por todas las sentencias
- Está listo para ser ejecutado
- Ya compilado es más eficiente
- Por ahí tardó más en compilar porque se verifica todo previamente
- El programa fuente no será público

Por el espacio ocupado

Intérprete:

- No pasa por todas las sentencias entonces ocupa menos espacio de memoria
- Cada sentencia se deja en la forma original y las instrucciones interpretadas necesarias para ejecutarlas se almacenan en los subprogramas del intérprete en memoria
- Tablas de símbolos, variables y otros se generan cuando se usan en forma dinámica (Ej. Python, Ruby)

Compilador:

- Pasa por todas las sentencias

- Una sentencia puede ocupar decenas o centenas de sentencias de máquina al pasar a código objeto
- Cosas cómo tablas de símbolos, variables, etc. se generan siempre se usen o no
- El compilador en general ocupa más espacio

Por la detección de errores

Intérprete:

- Las sentencias del código fuente pueden ser relacionadas directamente con la sentencia en ejecución entonces se puede ubicar donde se produjo el error
- Es más fácil detectarlos por donde pasa la ejecución
- Es más fácil corregirlos

Compilador:

- Se pierde la referencia entre el código fuente y el código objeto
- Es casi imposible ubicar el error, pobres en significado para el programador
- Se deben usar otras técnicas (ej. Semántica Dinámica)

Combinación de Técnicas

Interpretación pura y Compilación pura son dos extremos

En la práctica muchos lenguajes combinan ambas técnicas para sacar provecho a cada una

Los compiladores y los intérpretes se diferencian en como reportan los errores de ejecución

Ciertos entornos de programación contienen las dos versiones

Primero interpretar y después compilar

Se utiliza el intérprete en la etapa de desarrollo para facilitar el diagnóstico de errores

Con el programa validado se compila para generar un código objeto más eficiente

Primero compilar y luego interpretar

Se hace traducción a un código intermedio a bajo nivel que luego se interpretará

Sirve para generar código portable, es decir, código fácil de transferir a diferentes máquinas y con diferentes arquitecturas

Ejemplos:

- Compilador Java genera un código intermedio llamado “bytecode” que luego es interpretado por máquina virtual Java (JVM) en la máquina cliente
- C# (C Sharp) de Microsoft .NET
- VB.NET (Visual Basic .NET de Microsoft)
- Python

Funcionamiento de los compiladores

Pueden generar un "código ejecutable" (.exe) o un "código intermedio" (.obj)
La compilación puede ejecutarse en 1 o 2 etapas
En ambos casos se cumplen varias sub-etapas, las principales son:

1. Etapa de Análisis:
    - Análisis léxico (Programa Scanner)
    - Análisis sintáctico (Programa Parser)
    - Análisis semántico (Programa de Semántica estática)

    Podría generarse un código intermedio entre el paso 1 y el 2

2. Etapa de Síntesis
    - Optimización del código
    - Generación del código final

1. Está más vinculado al código fuente
2. Está más vinculado a características del código objeto y del hardware y arquitectura

Etapa de análisis del programa fuente

Análisis léxico (Scanner)

Es un proceso que lleva tiempo

Hace el análisis a nivel de palabra (lexema)

Divide el programa en sus elementos/categorías: identificadores, delimitadores, símbolos especiales, operadores, números, palabras clave, palabras reservadas, comentarios, etc.

Analiza el tipo de cada uno para ver si son tokens válidos

Filtra comentarios y separadores (como: espacios en blanco, tabulaciones, etc.)

Lleva una tabla para la especificación del analizador léxico. Incluye cada categoría, el conjunto de atributos y acciones asociadas

Pone los identificadores en la tabla de símbolos

Reemplaza cada símbolo por su entrada en la tabla de símbolos

Genera errores si la entrada no coincide con ninguna categoría léxica

El resultado de este paso será el descubrimiento de los ítems léxicos o tokens y detección de errores

Análisis sintáctico (Parser)

El análisis se realiza a nivel de sentencia/estructuras

Usa los tokens del analizador léxico

Tiene como objetivo encontrar las estructuras presentes en su entrada

Estas estructuras se pueden representar mediante el árbol de análisis sintáctico, que explica cómo se puede derivar la cadena de entrada en la gramática que especifica el lenguaje

Se identifican las estructuras de las sentencias, declaraciones, expresiones, etc. ayudándose con los tokens

El analizador sintáctico (Parser) se alterna/interactúa con el análisis léxico y análisis semántico

Usualmente usan técnicas de gramática formal

Se usa una gramática para construir el "árbol sintáctico"/"árbol derivación" del programa

El objetivo principal de un árbol de derivación es representar una sentencia del lenguaje y validar de esta forma que pertenece o no a la gramática para ver que lo que entra es correcto

Análisis semántica (semántica estática)

Debe pasar antes bien Scanner y Parser

Es la fase medular, una de las más importantes

Procesa las estructuras sintácticas (reconocidas por el analizador sintáctico)

Agrega otro tipo de información implícita y la estructura del código ejecutable continúa tomando forma

Realiza la comprobación de tipos (aplica gramática de atributos)

Agrega a la tabla de símbolos los descriptores de tipos

Realiza comprobaciones de duplicados, problema de tipos, etc

Realiza comprobaciones de nombres (toda variable debe estar declarada en su entorno)

Es el nexo entre etapas inicial y final del compilador(Análisis y Síntesis)

Generación de código intermedio

Es realizar la transformación del "código fuente" en una representación de "código intermedio" para una máquina abstracta

Queda una representación independiente de la máquina en la que se va a ejecutar el programa

Idea de la generación de código intermedio:

- Debe ser fácil de producir
- Debe ser fácil de traducir al programa objeto

El código intermedio más habitual es el código de 3-direcciones

Pasa todo el código del programa a secuencia de proposiciones de la forma:

x:= y op z

Etapa de síntesis

Construye el programa ejecutable y genera el código necesario

Si hay traducción separada de otros módulos (módulos, unidades, librerías, procedimientos, funciones, subrutinas, macros, etc.)

Interviene el Linkeditor (Programa) y se enlazan los distintos módulos objeto del programa

Se genera el módulo de carga: programa objeto completo

Se realiza el proceso de optimización (Optativo)

El Loader (Programa) lo carga en memoria

Optimización

Es Optativo. No se hace siempre y no lo hacen todos los compiladores

Los optimizadores de código (programas) pueden ser herramientas independientes, o estar incluidas en los compiladores e invocarse por medio de opciones de compilación

Hay diversas formas y cosas a optimizar:

- Elegir entre velocidad de ejecución y tamaño del código ejecutable
- Generar código para un microprocesador específico dentro de una familia de microprocesadores
- Eliminar la comprobación de rangos o desbordamientos de pila
- Evaluación para expresiones booleanas
- Eliminación de código muerto o no utilizado
- Eliminación de funciones no utilizadas

Clase 3 - Variables

Semántica operacional

La semántica operacional es fundamental para diversos aspectos del proceso de desarrollo de software, como el diseño de lenguajes de programación, la verificación de programas y la comprensión de cómo se ejecutan los programas en un nivel más bajo

Entidades

- Variables: nombre, tipo, área de memoria, etc.
- Rutinas: nombre, parámetros formales y reales, convención de pasaje de parámetros, etc.
- Sentencias: acción asociada

Ligaduras

Es el momento en el que un atributo de una entidad se asocia con un valor
Los programas trabajan con entidades
Las entidades tienen atributos (valor de la variable, nombre, etc.)
Estos atributos tienen que establecerse antes de poder usar la entidad

Los lenguajes difieren en:

- El número de entidades
- El número de atributos que se les pueden ligar
- El momento de la ligadura (binding time) (estática y dinámica)
- La estabilidad de la ligadura: ¿una vez establecida se puede modificar o es fija? (¿y si se puede modificar y es una constante?)

Ligadura estática

1. Se establece antes de la ejecución
2. No se puede modificar

El término estática hace referencia al binding time (1) y a su estabilidad (2).

Ligadura dinámica

1. Se establece durante la ejecución
2. Se puede modificar durante la ejecución de acuerdo a alguna regla específica del lenguaje

Excepción: constantes (el binding es en runtime pero no puede ser modifica luego de establecida)

Algunos atributos pueden ligarse en el momento de la definición del lenguaje, otros en el momento de implementación, en tiempo de traducción (compilación), y otros en el tiempo de ejecución

Atributos de las variables

Nombre

String de caracteres que se usa para referenciar a la variable (identificador)

Es introducido por una sentencia de declaración

La longitud máxima varía según el lenguaje (se define en la etapa de definición del lenguaje)

Alcance

Es el rango de instrucciones en el que se conoce el nombre, es visible, y puede ser referenciada (visibilidad)

Las instrucciones del programa pueden manipular las variables a través de su nombre dentro de su alcance. Afuera de ese alcance son invisibles

Ligadura por alcance estático

Llamado alcance léxico

Se define el alcance en términos de la estructura léxica del programa

Puede ligarse estáticamente a una declaración de variables (explícita o implícita) examinando el texto del programa, sin necesidad de ejecutarlo

La mayoría de los lenguajes adoptan reglas de ligadura de alcance estático

Ligadura por alcance dinámico

Define el alcance del nombre de la variable en términos de la ejecución del programa

Cada declaración de variable extiende su efecto sobre todas las instrucciones ejecutadas posteriormente, hasta que una nueva declaración para una variable con el mismo nombre es encontrada durante la ejecución

Global

Son todas las referencias a variables creadas en el programa principal

Local

Son todas las referencias a variables que se han creado dentro de una unidad (programa o subprograma)

No local

Son todas las referencias que se utilizan dentro del subprograma pero que no han sido creadas en el subprograma. (son externas a él)

Espacio de nombre

Es una zona separada abstracta del código donde se pueden agrupar, declarar y definir objetos (variables, funciones, identificador de tipo, clase, estructura, etc.)

Ayudan a evitar problemas con identificadores con el mismo nombre en grandes programas, o cuando se usan bibliotecas externas para evitar colisión de nombres

Al espacio de nombre se le asigna un nombre o identificador propio

Son utilizados por los lenguajes de tipo dinámico

Es un recurso de ciertos lenguajes de programación

Ayudan a resolver el alcance dentro de ese espacio de nombres

Tipo

Es el tipo de variables definidas, tiene asociadas rango de valores y operaciones permitidas

Se define el tipo de una variable cómo la especificación de:

- El conjunto de valores que se pueden asociar a la variable
- Un conjunto de operaciones permitidas (crear, acceder, modificar)
- Una variable, de un tipo dado, es una instancia

Cuando se define el lenguaje, ciertos nombres de tipos están vinculados a ciertas clases de valores y conjuntos de operaciones. Por ejemplo, el tipo entero y sus operadores asociados (+-*/) están vinculados a su representación matemática

Cuando se implementa el lenguaje los valores y operaciones están vinculados a una determinada representación de máquina

El binding también puede restringir el conjunto de valores que se pueden representar según la capacidad de almacenamiento de la máquina de destino

Antes que una variable pueda ser referenciada debe ligarse a un tipo

El tipo de una variable ayuda a:

- Proteger a las variables de operaciones no permitidas
- Chequear tipos
- Verificar el uso correcto de las variables (cada lenguaje tiene sus reglas de combinaciones de tipos)
- Detectar errores en forma temprana y a la confiabilidad del código

Tipos predefinidos

Son los tipos base que están descritos en la definición del lenguaje (enteros, reales, flotantes, booleanos, etc....)

Cada uno tiene valores y operaciones

Tipo boolean valores: true, false operaciones: and, or , not

Los valores se ligan en la implementación a representación de máquina según la arquitectura

Tipos definidos por el usuario

Permiten al programador con la declaración de tipos definir nuevos tipos a partir de los tipos predefinidos y de los constructores

Son esenciales para la organización y la abstracción

Permite al programador crear abstracciones, encapsular lógica y datos, reutilizar código y mejorar la claridad y legibilidad del código

Son fundamentales para el desarrollo de programas complejos y para mantener un código organizado y mantenible

Tipos de datos abstractos (TDA)

Son estructuras de datos que representa a un nuevo tipo abstracto con un nombre que identifica

Está compuesto por una colección de operaciones definidas (rutinas). Las rutinas son usadas para manipular los objetos de este nuevo tipo

TAD comunes: listas, colas, pilas, árboles, grafos, etc.

Cada TAD define un conjunto de operaciones permitidas, pero oculta los detalles de implementación interna

No hay ligadura por defecto, el programador debe especificar la representación y las operaciones

Momento de ligadura estático

El tipo se liga en compilación y no puede ser cambiado en ejecución

La ligadura entre variable y tipo se hace con la declaración

El chequeo de tipo también será estático

Explícito

La ligadura se establece mediante una sentencia de declaración

La ventaja reside en la claridad de los programas y en una mayor fiabilidad, porque cosas como errores ortográficos en nombres de variables pueden detectarse en tiempo de traducción

Implícito

Si no fue declarada la ligadura se deduce por "reglas propias del lenguaje"

Esto ocurre sin que el programador tenga que especificar explícitamente el tipo de datos de la variable

Ej. Fortran 77: variables que empiezan con I a N son Enteras variables que empiezan con el resto de las letras son Reales

Inferido

El tipo se deduce automáticamente de los tipos de sus componentes.

Se basa en el contexto del código y en el valor asignado a la variable

Se realiza en la traducción

Aplica en general a Lenguajes Funcionales

Momento de ligadura dinámico

El tipo se liga a la variable en ejecución y puede modificarse

Cambia cuando se le asigna un valor mediante una sentencia de asignación (no declaración)

No se detectan incorrecciones de tipo en las asignaciones

El tipo de la parte izquierda simplemente se cambia al tipo de la derecha

El costo de implementación de la ligadura dinámica es mayor (sobre todo el tiempo de ejecución por: comprobación de tipos, mantenimiento del descriptor asociado a cada variable en el que se almacena el tipo actual, cambio en el tamaño de la memoria asociada a la variable, etc.)

Chequeo dinámico

Menor legibilidad

Más errores

Los lenguajes interpretados en general adoptan ligadura dinámica de tipos

L-valor

Es el lugar de memoria asociado con la variable, está asociado al tiempo de vida (variables se alocan y desalocan)

Las variables se alocan en un área de memoria

Esa área de memoria debe ser ligada a la variable en algún momento

El L-valor de una variable es el área de memoria ligada a la variable durante la ejecución

Las instrucciones de un programa acceden a la variable por su L-Valor

Tiempo de vida (lifetime) o extensión

Periodo de tiempo que existe la ligadura

El tiempo de vida es el tiempo en que está alocada la variable en memoria y el binding existe

Es desde que se solicita hasta que se libera

Alocación

Momento en que se reserva la memoria para una variable

Estática

Se hace en compilación (antes de la ejecución) cuando se carga el programa en memoria en zona de datos y perdura hasta fin de la ejecución (sensible a la historia)

Variables estáticas

Dinámica

Se hace en tiempo de ejecución

Puede ser:

1. Automática: cuando aparece una declaración en la ejecución
2. Explícita: requerida por el programador con la creación de una sentencia, a través de algún constructor (por ej. algún puntero (new en pascal))

Persistente

Los objetos persistentes que existen en el entorno en el cual un programa es ejecutado, su tiempo de vida no tiene relación con el tiempo de ejecución del programa

Persisten más allá de la memoria

Ejemplo: archivos una vez creados/abiertos permanecen y pueden ser usados en diversas activaciones hasta que son borrados con un comando del sistema operativo (lo mismo sucede con las bases de datos)

R-valor

Es el valor codificado almacenado en la ubicación de la variable

R-Valor de una variable es el valor codificado almacenado en la locación asociada a la variable (l-valor)

La codificación se interpreta de acuerdo con el tipo de la variable

Ejemplo: 01110011 almacenado en una ubicación de memoria:

- Interpretado como nro. entero si la variable es tipo int (115)
- Interpretado como cadena si la variable es tipo char (s)

Objeto: (l-valor, r-valor) (dirección memoria, valor)

Se accede a la variable por el l-valor (ubicación)

Se puede modificar el r-value (valor) (salvo un caso especial)

Binding Dinámico

Binding dinámico de una variable a su valor:

- El valor (r-valor) puede cambiar durante la ejecución con una asignación
- El valor (r-valor) no puede cambiar si se define como constante simbólica definida por el usuario

b := a (copia el r-valor de a en el l-valor de b y cambia el r-valor de b)

a :=17 (asigna un valor directamente)

Constante

Se congela el valor

Ligadura estática

El valor que proporciona una expresión debe evaluarse en tiempo de compilación

El binding es el tiempo de compilación

El compilador puede sustituir legalmente el valor de la constante por su nombre simbólico en el programa

- const pi = 3.14 (está bien)
- int cant_items = 23
  const nro_paginas = 3 * cant_items (está mal, porque se usa el valor de una variable, el cual se va a resolver en ejecución, para definir el valor de la constante)

Ligadura dinámica

El valor se puede dar como una expresión que involucra otras variables y constantes, en consecuencia, el enlace se puede establecer en tiempo de ejecución, pero sólo cuando la variable es creada

Variables anónimas y referencias

Algunos lenguajes permiten que variables sin nombre sean accedidas por el r-valor de otra variable

Ese r-valor se denomina referencia o puntero a la variable

La referencia puede ser a el r-valor de una variable nombrada o el de una variable referenciada llamada access path de longitud arbitraria

Algunos lenguajes permiten que el r-valor de una variable sea una referencia al l-valor de otra variable

Puntero

Variable que sirve para señalar la posición de la memoria en que se encuentra otro dato almacenado como valor, con la dirección de ese dato

Alias

Se da si hay variables comparten un objeto en el mismo entorno de referencia, y sus caminos de acceso conducen al mismo objeto

El objeto compartido modificado vía un camino se modifica para todos los caminos

Ventajas

Compartir objetos se utiliza para mejorar la eficiencia

Desventajas

Generar programas que sean difíciles de leer

Generar errores porque el valor de una variable se puede modificar incluso cuando no se utiliza su nombre

Sobrecarga

Un nombre está sobrecargado si en un momento referencia más de una entidad

Debe estar permitido por el lenguaje

No aplica a todos los lenguajes

Hay que tener suficiente información para permitir establecer la ligadura unívocamente (por ejemplo del tipo (los tipos en una suma permiten saber cuál de las funcionalidades del operador + se va a utilizar))

Clase 4 - Unidades de programa

Unidades

Los lenguajes de programación permiten que un programa esté compuesto por unidades

Una unidad es la abstracción de una acción. En general se las llama rutinas (procedimientos/funciones)

Nombre

String de caracteres que se usa para invocar a la rutina (identificador)

El nombre de la rutina se introduce en su declaración y es lo que se usa para invocarlas

Alcance

Rango de instrucciones donde se conoce su nombre

El alcance se extiende desde el punto de su declaración hasta algún constructor de cierre

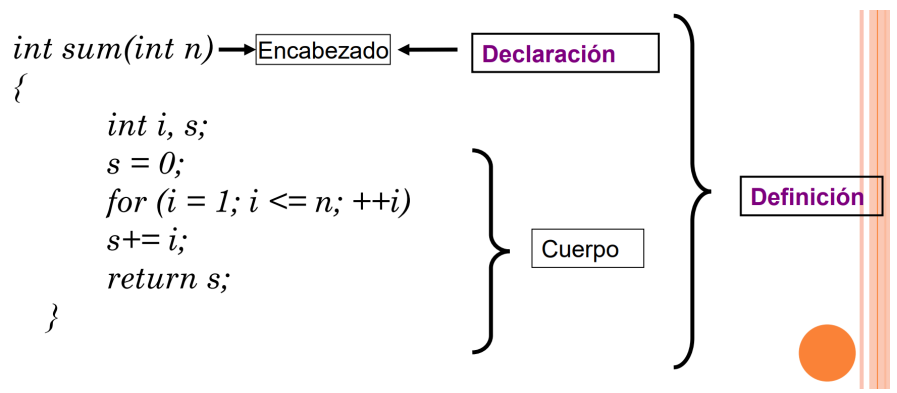
Según el lenguaje puede ser estático o dinámico

Activación: la llamada sólo puede estar dentro del alcance de la rutina

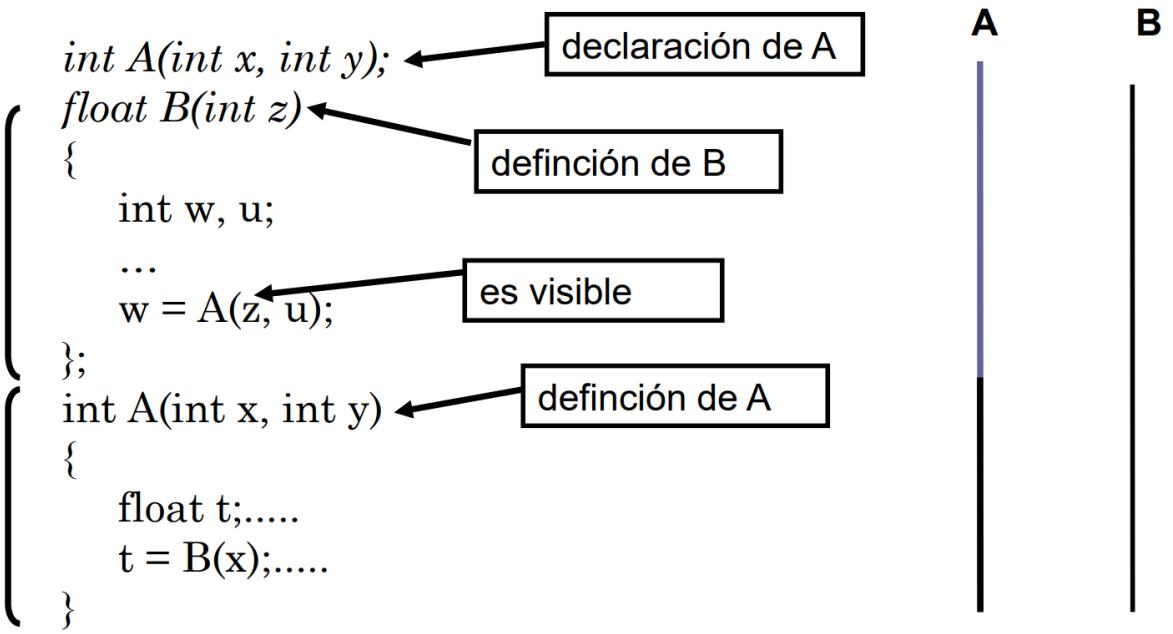
Definición vs declaración

Algunos lenguajes (C, C++, Ada, etc) hacen distinción entre definición y declaración de rutinas

Definición = declaración/encabezado + cuerpo



La separación de la declaración de la definición implica que se pueden hacer rutinas mutuamente recursivas:



Tipo

El encabezado de la rutina define el tipo de los parámetros y el tipo del valor de retorno (si lo hay)

Signatura: permite especificar el tipo de una rutina Una rutina fun que tiene como entrada parámetros de tipo T1, T2, Tn y devuelve un valor de tipo R, puede especificarse con la siguiente signatura fun: T1xT2x....Tn → R

Un llamado a una rutina es correcto si está de acuerdo al tipo de la rutina

La conformidad requiere la correspondencia de tipos entre parámetros formales y reales

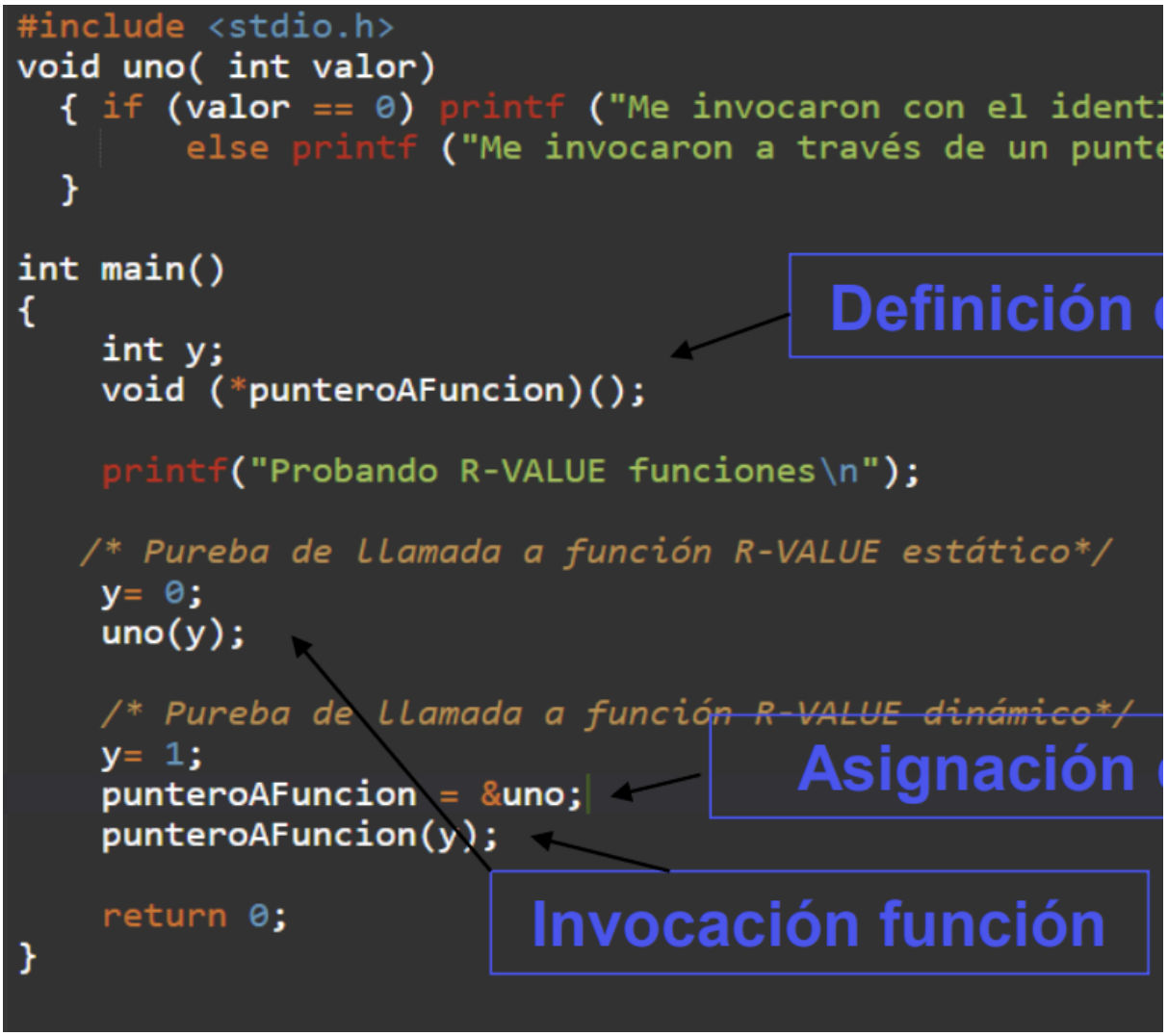
L-valor

Es el lugar de memoria en el que se almacena el cuerpo de la rutina

R-valor

La llamada a la rutina causa la ejecución de su código, eso constituye su r-valor

1. Estático: el caso más usual
2. Dinámica: variables de tipo rutina (se implementan a través de punteros a rutinas)



Comunicación entre rutinas

Parámetros formales: los que aparecen en la definición de la rutina

Parámetros reales: los que aparecen en la invocación de la rutina (dato o rutina)

Ligadura entre parámetros formales y reales

Método posicional

Se ligan uno a uno:

Rutina S (F1, F2, …, Fn) → Definición

S(A1, A2, …, An) → Llamado

Fi se liga a Ai para i de 1 a n. Deben conocerse las posiciones

Valores por defecto

C++: int distancia (int a = 0, int b = 0)

- distancia() → distancia(0, 0)
- distancia(10) → distancia(10, 0)
- distancia(10, 1) → distancia(10, 1)

Método por nombre

Se ligan por el nombre. Deben conocerse los nombres de los parámetros formales

Ada: procedure Ejem (A:T1; B: T2:= W; C:T3);

Si X, Y y Z son de tipo T1, T2 y T3

- Ejem (X, Y, Z) → asociación posicional
- Ejem (X, C => Z) → X se liga a A por posición, B toma el valor por defecto W C se liga a Z por nombre
- Ejem (C =>Z, A=>X, B=>Y) → se ligan todos por nombre

Representación en ejecución

La definición de la rutina especifica un proceso de cómputo

Cuando se invoca una rutina se ejecuta una instancia del proceso con los particulares valores de los parámetros

Instancia de la unidad: es la representación de la rutina en ejecución. Tiene dos zonas:

- Segmento de código: instrucciones de la unidad se almacenan en la memoria de instrucción (código: contenido fijo). Siempre es el mismo en una misma ejecución del programa
- Registro de activación: datos locales de la unidad se almacenan en la memoria de datos (datos: contenido cambiante). Se crea uno, por cada llamada a la rutina (con los datos particulares de esa llamada)

SIMPLESEM

- Memoria de Código: C(y) valor almacenado en la y-ésima celda de la memoria de código. Comienza en cero
- Memoria de Datos: D(y) valor almacenado en la y-ésima celda de la memoria de datos. Comienza en cero y representa el l-valor, D(y) o C(y) su r-valor
- Ip: puntero a la instrucción que se está ejecutando:
    - Se inicializa en cero en cada ejecución y se actualiza cuando se ejecuta cada instrucción
    - Direcciones de C

La ejecución de una rutina consiste en:

1. Obtener la instrucción actual para ser ejecutada (C[ip])
2. Incrementar ip
3. Ejecutar la instrucción actual
4. Repetir

Instrucciones

SET

Setea valores en la memoria de datos

set target, source: copia el valor representado por source en la dirección representada por target

set 10, D[20]: copia el valor almacenado en la posición 20 en la posición 10

E/S

read y write permiten la comunicación con el exterior

set 15, read el valor leído se almacena en la dirección 15

set write, D[50] se transfiere el valor almacenado en la posición 50

JUMP

Bifurcación incondicional

jump 47: la próxima instrucción a ejecutarse será la que esté almacenada en la dirección 47 de C

Direccionamiento indirecto:

jump D[30]

El ip va a a pasar a tener el valor que estaba en la posición 30 de la memoria de datos

JUMPT

Bifurcación condicional, bifurca si la expresión se evalúa como verdadera

jumpt 47, D[13]>D[8]: bifurca si el valor almacenado en la celda 13 es mayor que el almacenado en la celda 8

Punto de retorno

Es una pieza cambiante de información que debe ser salvada en el registro de activación de la unidad llamada

Ambiente de referencia

- Ambiente local: variables locales, ligadas a los objetos almacenados en su registro de activación
- Ambiente no local: variables no locales, ligadas a objetos almacenados en los registros de activación de otras unidades

Estructura de ejecución de lenguajes de programación

Estático

- Espacio fijo

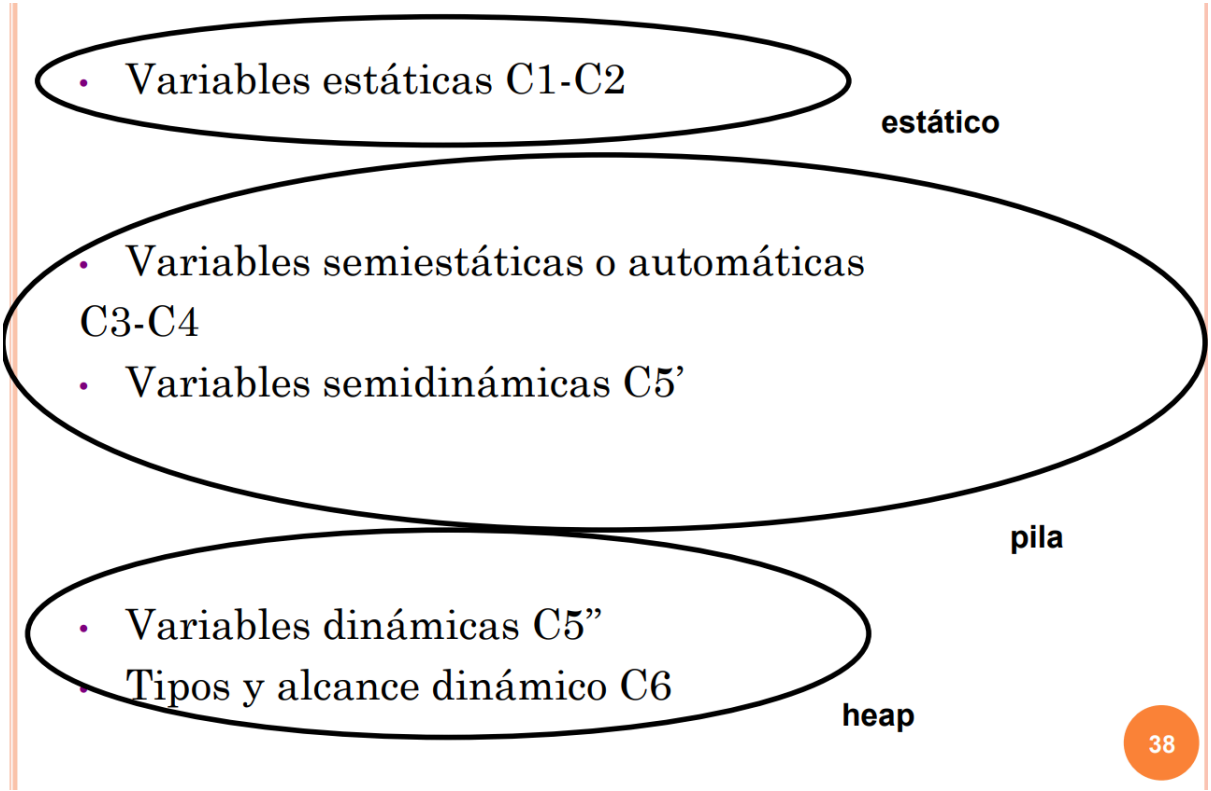
- El espacio necesario para la ejecución se deduce del código
- Todo los requerimientos de memoria necesarios se conocen antes de la ejecución
- La alocación puede hacerse estáticamente
- No puede haber recursión
- Todas las variables van a ser estáticas (en cuanto a tiempo de vida)

Basado en pila

- Espacio predecible (el registro de activación de cada unidad será de tamaño fijo y conocido, pero no se sabrá cuántas instancias de cada unidad se necesitarán durante la ejecución)
- El espacio se deduce del código (Algol-60, Pascal, Java)
- Programas más potentes cuyos requerimientos de memoria no pueden calcularse en traducción
- La memoria a utilizarse es predecible y sigue una disciplina last-in-first-out
- Las variables se alocan automáticamente y se desalocan cuando el alcance se termina
- Se utiliza una estructura de pila para modelizarlo
- Una pila no es parte de la semántica del lenguaje, es parte de nuestro modelo semántico

Dinámico

- Espacio impredecible
- Lenguajes con impredecible uso de memoria (Ruby, php, Python)
- Los datos son alocados dinámicamente sólo cuando se los necesita durante la ejecución
- No pueden modelizarse con una pila, el programador puede crear objetos de datos en cualquier punto arbitrario durante la ejecución del programa
- Los datos se alocan en la zona de memoria heap


Variables estáticas C1-C2
estático
Variables semiestáticas o automáticas
C3-C4
Variables semidinámicas C5'
pila
Variables dinámicas C5"
Tipos y alcance dinámico C6
heap
38

Clase 5 - Esquemas de ejecución

C1

- Sentencias simples
- Tipos simples:
    - Enteros
    - Reales
    - Arreglos
    - Estructuras
- Sin funciones
- Datos estáticos de tamaño fijo
- un programa = una rutina main()
    - Declaraciones
    - Sentencias
- E/S: read/write

Para este tipo de esquema se necesitan los siguientes elementos de ejecución:

- **IP**

Cada unidad de programa, en su registro de activación, va a tener solo sus datos locales

C2

- Definición de rutinas internas al main
- Programa:
    - Datos globales
    - Declaraciones de rutinas
    - Rutina principal
        - Datos locales
        - Se invoca automáticamente en ejecución
- Rutinas internas:
    - Disjuntas: no pueden estar anidadas
    - No son recursivas

    - Ambiente de las rutinas internas:
        - Datos locales
        - Datos globales

Para este tipo de esquema se necesitan los siguientes elementos de ejecución:

- IP
- **Punto de retorno**

Cada unidad de programa, en su registro de activación, va a tener sus datos locales y el punto de retorno

C3

- Esquema basado en pila
- Rutinas con capacidad de llamarse a sí mismas (recursión directa) o de llamar a otra rutina en forma recursiva (recursión indirecta)
- Rutinas con la capacidad de devolver valores, es decir, funciones

El registro de activación de cada unidad será de tamaño fijo y conocido, pero no se sabrá cuántas instancias de cada unidad se necesitarán durante la ejecución

Igual que en C2 el compilador puede ligar cada variable con su desplazamiento dentro del correspondiente registro de activación. El desplazamiento es estático

La dirección donde se cargará el registro de activación, es dinámica, por lo tanto, la ligadura con la direcciones absolutas en la zona de datos de la memoria, solo puede hacerse en ejecución

Cada nueva invocación aloca un nuevo registro de activación y se establecen las nuevas ligaduras entre el segmento de código y el nuevo registro de activación

Para este tipo de esquema se necesitan los siguientes elementos de ejecución:

- IP
- **Current**
- **Free**
- Punto de retorno
- **Valor de retorno**
- **Link dinámico**

Cada unidad de programa, en su registro de activación, va a tener sus datos locales, el punto de retorno, el valor de retorno y el link dinámico

C4

- Esquema basado en pila
- Controla el alcance de las variables
- Define el tiempo de vida de las variables
- Divide el programa en unidades más pequeñas
- Los bloques pueden ser:
    - Disjuntos (no tiene porción común)
    - Anidados (un bloque está completamente contenido en otro
- Puede haber sentencias compuestas, que contengan bloques

C4''

- Permite rutinas anidadas

Para este tipo de esquema se necesitan los siguientes elementos de ejecución:

- IP
- Current
- Free
- Punto de retorno
- Valor de retorno
- Link dinámico
- **Link estático**

Cada unidad de programa, en su registro de activación, va a tener sus datos locales, el punto de retorno, el valor de retorno, el link dinámico y el link estático

C5'

- Registro de activación cuyo tamaño se conoce cuando se activa la unidad
- Datos semidinámicos

type vector is array (integer range <>);

Define un arreglo con índice irrestricto:

- A: vector(0..N);
- B: vector(1..M);

N y M deben ligarse a algún valor entero para que A y B puedan alocarse en ejecución (referencia al ambiente no local o parámetros)

Compilación: se reserva lugar en el registro de activación para los descriptores de los arreglos dinámicos

Todos los accesos al arreglo dinámico son traducidos como referencias indirectas a través del puntero en el descriptor, cuyo desplazamiento se determina estáticamente

C5''

- Los datos pueden alocarse durante la ejecución
- Datos dinámicos: se alocan explícitamente durante la ejecución mediante instrucciones de alocación
- Los datos dinámicos se alocan en la heap

C6

- Lenguajes dinámicos
- Se trata de aquellos lenguajes que adoptan más reglas dinámicas que estáticas
- Usan tipado dinámico y reglas de alcance dinámicas
- Se podrían tener reglas de tipado dinámicas y de alcance estático, pero en la práctica las propiedades dinámicas se adoptan juntas
- Una propiedad dinámica significa que las ligaduras correspondientes se llevan a cabo en ejecución y no en compilación

Clase 6 - Parámetros

Rutinas

- También llamadas subprogramas
- Son una unidad de programa (función, procedimiento)
- Están formadas por un conjunto de sentencias que representan una acción abstracta
- Permiten al programador definir una nueva operación a semejanza de las operaciones primarias ya integradas en el lenguaje
- Permiten ampliar a los lenguajes, dan modularidad, claridad y buen diseño
- Se lanzan con una llamada explícita (se invocan por su nombre) y luego retornan a algún punto de la ejecución (responden al esquema call/return)
- Los subprogramas son el ejemplo más usual y útil presente desde los primeros lenguajes ensambladores

  Tienen dos formas:

  1. Procedimientos (ejecutan una acción)
  2. Funciones (devuelven un valor)

Procedimientos

- Un procedimiento es una construcción que permite dar nombre a un conjunto de sentencias y declaraciones asociadas que se usarán para resolver un subproblema dado
- Brindará una solución de código más corta, comprensible y fácilmente modificable
- Permiten al programador definir y crear nuevas acciones/sentencias
- El programador las invocará
- Pueden no recibir ni devolver ningún valor
- Los resultados los produce en variables no locales o en parámetros que cambian su valor

Funciones

- Mientras que un procedimiento ejecuta un grupo de sentencias, una función además devuelve un valor al punto donde se llamó

- El valor que recibe la función se usa para calcular el valor total de la expresión y devolver algún valor
- Permite al programador crear nuevas operaciones
- Similar a las funciones matemáticas ya que hacen algo y luego devuelven un valor y no producen efectos colaterales
- Se las invoca dentro de expresiones y lo que calcula reemplaza a la invocación dentro de la expresión
- Siempre deben retornar un valor

Conclusiones

- Cuando se diseña un subprograma el programador se concentra en el cómo trabajará dicho subprograma
- Cuando se usa un subprograma se ignorará el cómo. Sólo interesará el qué me permite hacer (la implementación permanece oculta (abstracción))
- Con una sola definición se pueden crear muchas activaciones. La definición de un subprograma es un patrón para crear activaciones durante la ejecución
- Un subprograma es la implementación de una acción abstracta y su invocación representa el uso de dicha abstracción
- Codificar un subprograma es como si hubiéramos incorporado una nueva sentencia a nuestro lenguaje

Formas de comunicar rutinas

Si utilizan variables locales no hay problema

Si no utilizan variables locales hay 2 formas

Acceso al ambiente no local

Se comparten variables que son de otra unidad con un acceso no local, no es muy claro, y es bastante propenso a errores

Se usa cadena estática o dinámica

Ambiente no local implícito

- Es automático
- Utiliza alguna de las 2 reglas:
    - Regla de alcance dinámico (quién me llamó y busco el identificador)
    - Regla de alcance estático (dónde está contenido y busco el identificador)

Ambiente común explícito

- Permite definir áreas comunes de código
- El programador debe especificar que es lo comparte
- Cada lenguaje tiene su forma de realizarlo

A través de parámetros

- Parámetro Real (Argumento):
    - Es un valor u otra entidad que se pasa a un procedimiento o función
    - Están colocados en la parte de la invocación de la rutina
- Parámetro Formal (Parámetro):
    - Es una variable utilizada para recibir valores de entrada en una rutina, subrutina, etc.
    - Están colocados en la parte de la declaración de la rutina

El pasaje de parámetros es mejor, ya que el uso intensivo de accesos al ambiente no local puede provocar alguna pérdida de control, y provocar que las variables terminen siendo visibles donde no es necesario y llevar a errores

Ventajas

- Brinda distintas posibilidades de compartir cosas (que veremos más adelante)
- Permite enviar distintos parámetros en distintas invocaciones a las rutinas
- Más flexibilidad, se pueden transferir más datos y de diferente tipo en cada llamada
- Permite compartir en forma más abstracta sólo especificamos el nombre y tipo a argumentos y parámetros

- Protección: el uso intensivo de accesos al ambiente no local decrementa la seguridad de las soluciones ya que las variables terminan siendo visibles aun donde no es necesario o donde no debería
- Legilibilidad:
    - Permite al programador encontrar más fácilmente los errores. (Ej. Si transformo en rutinas o funciones a los que les paso valores, es más fácil depurar y encontrar errores y no chequear cada repetición en el código). Esto le da modificabilidad, si hay errores uno se focaliza en qué cosas estoy compartiendo, qué argumentos y parámetros estoy utilizando y su tipo

Parámetros

Los parámetros formales son variables locales a su entorno

Se declaran con una sintaxis particular a cada lenguaje

Sirven para intercambiar información entre la función/rutina que hace la llamada y la que la recibe

Momento de vinculación de parámetros reales y formales

Comprende la evaluación de los parámetros reales y la ligadura con los parámetros formales

- Evaluación:
    - En general antes de la invocación primero se evalúan los parámetros reales, y luego se hace la ligadura
    - Se verifica que todo esté bien antes de transferir el control a la unidad llamada
- Ligadura:
    - Por posición:
        - Se corresponden con la posición que ocupan en la lista de parámetros
        - Van en el mismo orden
    - Por nombre o palabra clave: se corresponden con el nombre por lo tanto pueden estar colocados en distinto orden en la lista de parámetros

Datos como parámetros

In

El parámetro formal recibe el dato desde el parámetro real
La conexión es al inicio, se copia y se corta la vinculación

Por valor

El valor del parámetro real se usa para inicializar el correspondiente parámetro formal al invocar la unidad

Se transfiere el dato real y se copia

En este caso el parámetro formal actúa como una variable local de la unidad llamada, y crea otra variable

La conexión es al inicio para pasar el valor y se corta la vinculación

Es el mecanismo por default y el más usado

Desventajas:

- Consume tiempo para hacer la copia de cada parámetro
- Consume almacenamiento para duplicar cada dato (pensar grandes volúmenes)

Ventajas:

- Protege los datos de la unidad llamadora, el parámetro real no se modifica
- No hay efectos negativos o colaterales

Por valor constante

Se envía un valor, pero la rutina receptora no puede modificarlo, es decir queda con un valor fijo que no se puede cambiar

No indica si se realiza o no la copia (dependerá del lenguaje)

La implementación debe verificar que el parámetro real no sea modificado

No todos los lenguajes permiten el modo IN con pasaje por valor constante

Desventajas:

- Requiere realizar más trabajo para implementar los controles

Ventajas:

- Protege los datos de la unidad llamadora
- El parámetro real no se modifica

Out

Se invoca la rutina y cuando esta termina devuelve el parámetro formal al parámetro real

La conexión es al final

Por resultado

El valor del parámetro formal de la rutina se copia al parámetro real al terminar de ejecutarse la unidad que fue llamada

El parámetro formal es una variable local del entorno de la rutina

El parámetro formal es una variable sin valor inicial porque no recibe nada

Desventajas:

- Consume tiempo y espacio porque hace copia al final
- Debemos inicializar la variable en la unidad llamada de alguna forma (si el lenguaje no lo hace por defecto)

Ventajas:

- Protege los datos de la unidad llamadora, el parámetro real no se modifica en la ejecución de la unidad llamada

Por resultado de funciones

Es el resultado que me devuelven las funciones

Reemplaza la invocación en la expresión que contiene el llamado

Puede devolverse de distintas formas según lenguaje:

- Return como en Python, C, etc.
- Nombre de la función (último valor asignado) que se considera como una variable local (como en Pascal)

In/Out

El parámetro formal recibe el dato del parámetro real y el parámetro formal le envía el dato al parámetro real al finalizar la rutina

La conexión es al inicio y al final

Valor/Resultado

El parámetro formal es una variable local que recibe una copia a la entrada del contenido del parámetro real y a la salida el parámetro real recibe una copia de lo que tiene el parámetro formal

Cuando se invoca la rutina, el parámetro real le da valor al parámetro formal (se genera copia) y se desliga en ese momento

La rutina trabaja sobre ese parámetro formal pero no afecta al parámetro real porque trabaja sobre su copia

Cada referencia al parámetro formal es una referencia local

Una vez que termina de ejecutar el parámetro formal le devuelve un valor al parámetro real y lo copia

Se dice que hay una ligadura y una conexión entre parámetro real y el formal cuando se inicia la ejecución de la rutina y cuando se termina, pero no en el medio

Tiene las desventajas y las ventajas de ambos:

Desventajas:

- Consume tiempo para hacer la copia de cada parámetro
- Consume almacenamiento para duplicar cada dato (pensar grandes volúmenes)
- Consume tiempo y espacio porque hace copia al final

Ventajas:

- Protege los datos de la unidad llamadora, el parámetro real no se modifica
- No hay efectos negativos o colaterales

Por referencia

También llamada por "variable"

No es copia por valor es por referencia

Se asocia la dirección (l-valor) del PR al PF

La conexión es al inicio y permanece hasta el final

El PF será una variable local a su entorno que contiene la dirección al PR de la unidad llamadora que estará entonces en un ambiente no local. Así se extiende el alcance de la rutina (aliasing situation)

Cada referencia al PF será a un ambiente no local, entonces cualquier cambio que se realice en el PF dentro del cuerpo del subprograma quedará registrado en el PR. El cambio será automático

El PR queda compartido por la unidad llamadora y llamada. Será bidireccional

Desventajas:

- Se puede llegar a modificar el PR inadvertidamente. Es el peor problema
- El acceso al dato es más lento por la indirección a resolver cada vez que se invoque
- Se pueden generar alias cuando dos variables o referencias diferentes se asignen a la misma dirección de memoria
- Estas cosas afectan la legibilidad y por lo tanto la confiabilidad, se hace muy difícil la verificación de programas y depuración de errores

Ventajas:

- Ya que no se realizan copias de los datos: será eficiente en espacio y tiempo sobre todo en grandes volúmenes de datos
- La indirección es de bajo costo de implementación por muchas arquitecturas

Por nombre

No se verá en detalle en la materia

El parámetro formal es sustituido "textualmente" por una expresión del parámetro real, más un puntero al entorno del parámetro real (expresión textual, entorno real)

Se utiliza una estructura aparte que resuelve esto

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 (la dirección se resuelve en ejecución)

Distinto semánticamente a por referencia

El objetivo es otorgar evolución de valor diferida

No es de los más utilizados por los lenguajes

Thunks: es una unidad pequeña de código (función) que encapsula y representa a una expresión que pospone su evaluación hasta que sea necesario. Es un concepto clave en la evaluación perezosa/diferida

Para implementar el pasaje por nombre se utilizan:

- Los thunks que son procedimientos sin nombre
- Cada aparición del parámetro formal se reemplaza en el cuerpo de la unidad llamada por una invocación a un thunks
- En el momento de la ejecución activará al procedimiento que evaluará el parámetro real en el ambiente apropiado

Desventajas:

- Es un método que extiende el alcance del parámetro real, pero esto mismo puede llevar a errores
- Posee evaluación diferida al ejecutar
- Es más lento ya que debe evaluarse cada vez que se lo usa (ej.: si es un loop se evalúa cada vez)
- Es difícil de implementar y genera soluciones confusas para el lector y el escritor

Subprogramas como parámetros

En algunas situaciones es conveniente o necesario poder usar nombres de subprogramas como parámetro para ejecutar alguna acción

En general se usa con funciones matemáticas

No lo incorporan todos los lenguajes

Algunas cosas pueden ser confusas de resolver (chequeo de tipos de subprogramas, subprogramas anidados, etc.)

Ambiente de referencia para las referencias no locales dentro del cuerpo del subprograma pasado como parámetro

Debe determinarse cuál es el ambiente de referencia no local correcto para un subprograma que se ha invocado y que ha sido pasado como parámetro

Cuando un procedimiento que se pasó como parámetro se ejecuta, puede tener problemas con referencias que hay dentro del procedimiento

Si hay una variable que no pertenece a él y no es local entonces surge el problema a dónde va a buscarla porque este es un procedimiento que se mandó como parámetro

Si el lenguaje sigue la cadena estática para buscar su referencia se pregunta dónde está contenido este procedimiento, ahí no hay problema

Si el lenguaje sigue la cadena dinámica entonces se pregunta quién lo llamó

Hay varias opciones para resolverlo:

- Ligadura shallow o superficial: el ambiente de referencia, es el del subprograma que tiene declarado el parámetro formal del subprograma. Ejemplo: SNOBOL (usaremos este en la práctica)
- Ligadura deep o profunda: el ambiente es el del subprograma donde está declarado el subprograma usado como parámetro real. Se utiliza en los lenguajes con alcance estático y estructura de bloque
- Ligadura ad hoc: el ambiente del subprograma donde se encuentra el llamado a la unidad que tiene un parámetro subprograma. Menos fiable (poco natural)

Unidades genéricas

Son una opción para lenguajes que no permiten pasar parámetros de tipo subprogramas

Una unidad genérica es una unidad que puede instanciarse con parámetros formales de distinto tipo. Por ejemplo, una unidad genérica que ordena elementos de un arreglo podrá instanciarse para que ordene elementos enteros, flotantes, etc.

Permiten compartir cosas, procesos, procedimientos o lo que se defina, y reutilizar así código

Las unidades genéricas no se usan directamente, sino sólo para crear instancias no genéricas a partir de ellas

Las unidades genéricas representan una plantilla mediante la cual se indica al compilador cómo crear unidades no genéricas

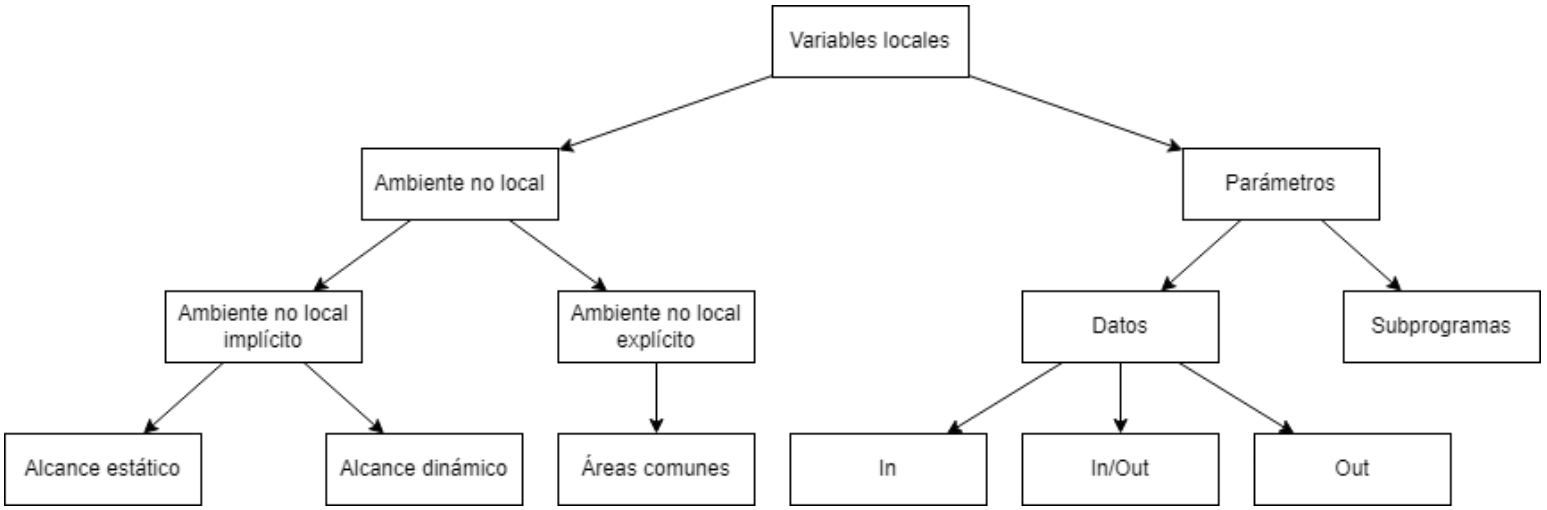
Como pueden usarse diferentes instancias con diferentes subprogramas, proveen la funcionalidad del parámetro subprograma

Ada permite crear unidades genéricas, con parámetros que se pueden concretar para diferentes instancias de la unidad, consiguiendo de esta manera la reutilización de código

Se debe anteceder la declaración de la unidad correspondiente con la palabra "generic"

Los parámetros formales de la unidad se sitúan en la zona comprendida entre la palabra "generic" y el comienzo de la declaración de la unidad

Para crear una instancia de una unidad genérica hay que especificar un nombre para la unidad no genérica que el compilador va a construir y los parámetros reales genéricos a utilizar en el lugar de los formales



Clase 7 - Tipos de datos

Concepto

Son más que un conjunto de datos, ya que tienen un comportamiento semántico o sentido

Podemos definir a un tipo como un conjunto de valores y un conjunto de operaciones que se pueden utilizar para manipularlos

Historia

Los primeros lenguajes tenían el inconveniente de que las estructuras de datos podían ser modeladas sólo con los pocos tipos de datos básicos definidos por el lenguaje

Se empieza a ver una clara intención de soportar varios y distintos tipos de datos con el objeto brindar un mayor apoyo al desarrollo de una amplia variedad de aplicaciones

Tomando el concepto de tipo de dato definido por el usuario arribamos al concepto de tipo de dato abstracto. Separa la representación y conjunto de operaciones (invisibles al usuario)

La evolución del TAD es el concepto de Clase

Tipo de dato

	Elementales/escalares	Compuestos
Predefinidos	● Enteros ● Reales ● Caracteres ● Booleanos	String
Definidos por el usuario	Enumerados	● Arreglos ● Registros ● Listas

Su dominio es el conjunto de valores posibles

Cualquier lenguaje de programación está equipado con un conjunto finito de tipos predefinidos (built-in / primitivos), que normalmente reflejan el comportamiento del hardware subyacente

A nivel de hardware, los valores pertenecen al dominio sin tipo, lo que constituye el dominio universal, estos son interpretados de manera diferente, según los diferentes tipos que se utilicen

Los lenguajes de programación deben permitir al programador especificar agrupaciones de objetos de datos elementales (o tipos predefinidos)

Estas definiciones de tipos de datos en función de otros denominan tipos de datos definidos por el usuario

Lo puede realizar de forma recursiva, o mediante agregaciones de agregados o uniones

Esto se logra mediante la prestación de una serie de constructores que permiten definir a estos tipos de datos definidos por el usuario

Entre los tipos compuestos definidos por el usuario se encuentran los tipos estructurados pues se utilizan estructuras de datos para su composición

Tipos predefinidos

Reflejan el comportamiento del hardware subyacente y son una abstracción de él

Las ventajas de los tipos predefinidos son:

- Invisibilidad de la representación
- Verificación estática
- Desambiguar operadores
- Control de precisión

Que un conjunto de valores de un tipo sea definido por la implementación del lenguaje significa que será seleccionado por el compilador. Mientras que si el tipo es definido por el lenguaje será definido en su definición

Tipos definidos por el usuario

Los lenguajes de programación permiten al programador especificar agrupaciones de objetos de datos elementales (o tipos predefinidos) y, de forma recursiva, agregaciones

de agregados. Esto se logra mediante la prestación de una serie de constructores que permiten definir lo que denominamos tipo de dato definido por el usuario

Separan la especificación de la implementación. Se definen los tipos que el problema necesita

Permiten:

- Instanciar objetos de las agregaciones
- Definir nuevos tipos de dichas agregaciones
- Hacer un chequeo de consistencia

Ventajas:

- Legibilidad: se puede elegir el nombre más apropiado para el nuevo tipo
- Estructura jerárquica de las definiciones de tipos
- Modificabilidad: solo se cambia en la definición
- Factorización: se usa la cantidad de veces necesarias
- 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

Constructores de tipos compuestos

Producto cartesiano

Es un registro

Por ej.: un registro de un polígono representa el producto cartesiano entre el conjunto de los enteros (para el número de lados) y el conjunto de los reales (para el tamaño de dichos lados)

Correspondencia finita

Es un arreglo (cada número dentro de la dimensión del arreglo se corresponde a un valor al que se accede mediante un subíndice)

Mapea valores de un tipo de dato hacia otros, a través del cumplimiento de una relación

Ej.Arreglos, vectores, matrices, listas de Python

Están indexados, ordenados. Algunos usan [], pero ADA usa () para acceder por posición

En APL, Algol 68, Ada, Python se pueden indexar por más de un elemento en el rango (slicing)

Los lenguajes interpretados permiten tener elementos de distinto tipo

Unión

Permite manipular diferentes tipos en distinto momento de la ejecución

En la unión y en la unión discriminada el chequeo de tipos debe hacerse en ejecución

La declaración es muy similar a la del producto cartesiano. La diferencia es que sus campos son mutuamente excluyentes

Unión discriminada:

Agrega un discriminante para indicar la opción elegida

Si tenemos la unión discriminada entre dos conjuntos S y T, y aplicamos el discriminante a un elemento **e** perteneciente a la unión discriminada, devolverá S o T

El elemento **e** debe manipularse de acuerdo al valor del discriminante

La unión discriminada se puede manejar en forma segura consultando el discriminante antes de utilizar el valor del elemento

Las uniones discriminadas son un poco más seguras pues permiten al programador manejar la situación a través del discriminante

Algunos lenguajes como Pascal, soportan unión discriminada a través de la definición de variantes:

```
type natural = 0. .maxint;
address_type = (absolute, offset);
safe_address = record
        case kind: address_type of
                absolute: (abs_addr: natural)
                offset: (off_addr: integer)
        end;
```

Problemas:

- El chequeo debe realizarse en ejecución. No se puede asegurar en compilación qué tipo o variante adquiere una variable

- El discriminante y las variantes pueden manejarse independientemente uno de otros
- La implementación del lenguaje puede ignorar los chequeos
- Puede omitirse el discriminante, con lo cual aunque se quisiera no se puede chequear

Recursión

Un tipo de dato recursivo T se define como una estructura que puede contener componentes del tipo T

Define datos agrupados:

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

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

Un puntero es una referencia a un objeto

Una variable puntero es una variable cuyo r-valor es una referencia a un objeto (el l-valor del otro objeto)

Punteros

Estructuras de tamaño arbitrario

Número de ítems no determinado: los punteros permiten conectar juntos muchos items sin tener un nombre explícito para todos ellos (recursión)

Relaciones múltiples entre los ítems: los punteros permiten que el dato sea puesto en varias estructuras sin necesidad de duplicarlo

Acceso a bajo nivel: los punteros están cerca de la máquina

Valores:

- Direcciones de memoria
- Valor nulo (no asignado) dirección no válida

Operaciones:

- Asignación de valor: generalmente asociado a la alocación de la variable apuntada
- Referencias: a su valor (como dirección), operaciones entre punteros al valor de la variable apuntada: desreferenciación implícita

Los punteros son un mecanismo muy potente para definir estructuras de datos recursivas

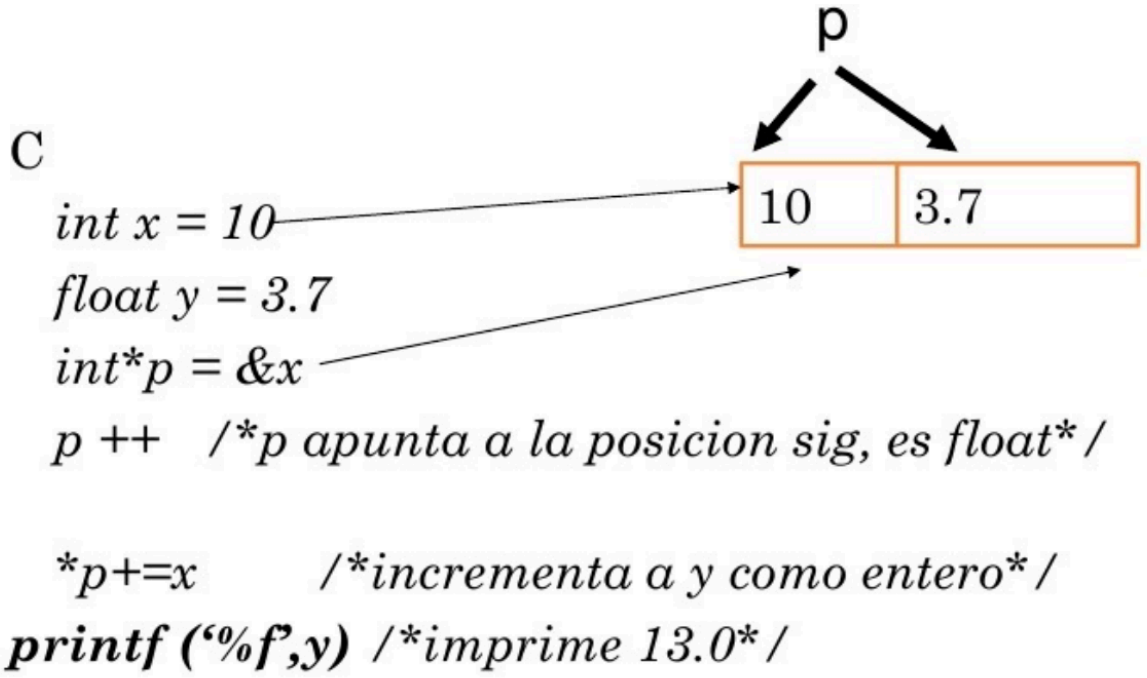
Por acceder a bajo nivel, pueden oscurecer o hacer inseguros a los programas que los usan

Se compara a los punteros con los go to:

- Los go to amplían el rango de sentencias que pueden ejecutar
- Los punteros amplían el rango de las celdas de memoria que pueden ser referenciadas por una variable y también amplían el tipo de los valores que puede contener un objeto

Inseguridades

Violación de tipos



Referencias sueltas (referencias dangling)

Si este objeto no está alocado se dice que el puntero es peligroso (dangling)

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/void/null)

Punteros y uniones discriminadas

```
union ojo{
    int int_var,
    int* int_ref
}
```

En el caso de C, este es el mismo efecto que causa la aritmética de punteros

Alias

```
int* p1;
int* p2;
int x;
p1 = &x;
p2 = &x;
```

p1 y p2 son punteros

p1 y x son alias

p2 y x también lo son

Se da si hay variables comparten un objeto en el mismo entorno de referencia, y sus caminos de acceso conducen al mismo objeto

El objeto compartido modificado vía un camino se modifica para todos los caminos

Ventaja: compartir objetos se utiliza para mejorar la eficiencia

Desventajas:

- Genera programas que sean difíciles de leer
- Puede generar errores porque el valor de una variable se puede modificar incluso cuando no se utiliza su nombre

Liberación de memoria: objetos perdidos

Las variables puntero se alocan como cualquier otra variable en la pila de registros de activación

Los objetos (apuntados) que se alocan a través de la primitiva new son alocados en la heap

La memoria disponible (heap) podría rápidamente agotarse a menos que de alguna forma se devuelva el almacenamiento alocado liberado

Manejo de memoria

Si los objetos en el 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

La liberación de espacio en la heap puede ser explícita (dispose()/delete()) o implícita (garbage collector)

El reconocimiento de la basura recae en el programador, quien notifica al sistema cuando un objeto ya no se usa

No garantiza que no haya otro puntero que apunte a esta dirección definida como basura (si eso pasa, este otro puntero se transforma en dangling)

Este error es difícil de chequear y la mayoría de los lenguajes no lo implementan por que es costoso

En caso de implementar un garbage collector el mismo debe ser muy eficiente y ejecutar en paralelo para no bajar el rendimiento del lenguaje

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)

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

¿Qué se debe tener en cuenta un lenguaje que provee manejo de excepciones?

- ¿Cuáles son las excepciones que se pueden manejar?
- ¿Cómo se definen?
- ¿Cómo se maneja una excepción y cuál es su ámbito?
- ¿Cómo se lanza una excepción?
- ¿Cómo especificar los controladores de excepciones que se han de ejecutar cuando se alcanzan las excepciones?
- ¿A dónde se cede el control cuando se termina de atender una excepción?
- ¿Cómo se propagan las excepciones?
- ¿Hay excepciones predefinidas?
- ¿Hay situaciones no controladas que lleven a mayores fallos?

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

Si se deseara continuar ejecutando las instrucciones de un bloque donde se lanza una excepción, es preciso “crear un bloque más interno”

Se usa Declare para amar una unidad y luego agregar instrucciones restantes abajo (simular reasunción)

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