Paradigmas de Programación

2. Paradigma Imperativo

Departamento de Informática
Universidad de Valladolid

Curso 2023-24

Grado en Ingeniería Informática
Grado en Estadística

Indice

1. Código: Control, Subrutinas, Modulos
   i. Expresiones, Asignación
   ii. Estructuras de Control ← Programación Estructurada
   iii. Subrutinas ← Paradigma Procedimental
      - Tratamiento de excepciones
      - Aberrantia: Continuations, Closures, Coroutines
   iv. Módulos ← Paradigma Modular
2. Datos: Sistemas de Tipado
   i. Conceptos. Tipos de datos habituales.
   ii. Tipado Estático vs. Dinámico
   iii. Tipado Fuerte vs. Débil
   iv. Tipado Seguro vs. Inseguro

Paradigma Imperativo

- Describe **cómo** debe realizarse el cálculo, no el **porqué**.
- Un cómputo consiste en una serie de sentencias, ejecutadas según un control de flujo **explícito**, que **modifican el estado** del programa.
- Las variables son **celdas de memoria** que contienen datos (o referencias), pueden ser modificadas, y representan el **estado** del programa.
- La sentencia principal es la **asignación**.
- Basado en el modelo de cómputo de máquinas de Turing y sobre todo en las máquinas RAM (registro con acceso aleatorio a memoria)
- La gran mayoría de procesadores siguen una versión de ese modelo de cómputo con arquitectura Von Neumman.

Ensamblador

- El antecesor de los lenguajes imperativos
- Tipos de instrucciones de código máquina:
  - Mover datos de registros a direcciones de memoria y viceversa (con mecanismos de indirección)
  - Operaciones aritméticas sencillas sobre registros
  - Tests y saltos condicionales
- El ensamblador establece una capa de abstracción:
  - Identificadores de operaciones (opcodes), valores, direcciones de memoria
  - Secciones de datos
  - Directivas, macros

```
10110000 01100001
B0 61
MOV AL, 61h
MOV [ESI+EAX], CL
```

1.1. Elementos básicos

- La instrucción básica del paradigma imperativo es la asignación. Elementos que intervienen:
  - Variables : En este paradigma las variables son identificadores asociados a celdas de memoria, las cuales contienen valores o referencias y pueden ser modificadas.
  - Expresiones
  - Valores literales
  - Funciones predefinidas
  - Operadores
    - Niveles de Precedencia y Asociatividad
- Variantes entre distintos lenguajes:
  - Símbolo de asignación (:=, =), autoasignación, autoincrementos
  - Equivalencia expresión <=> sentencia

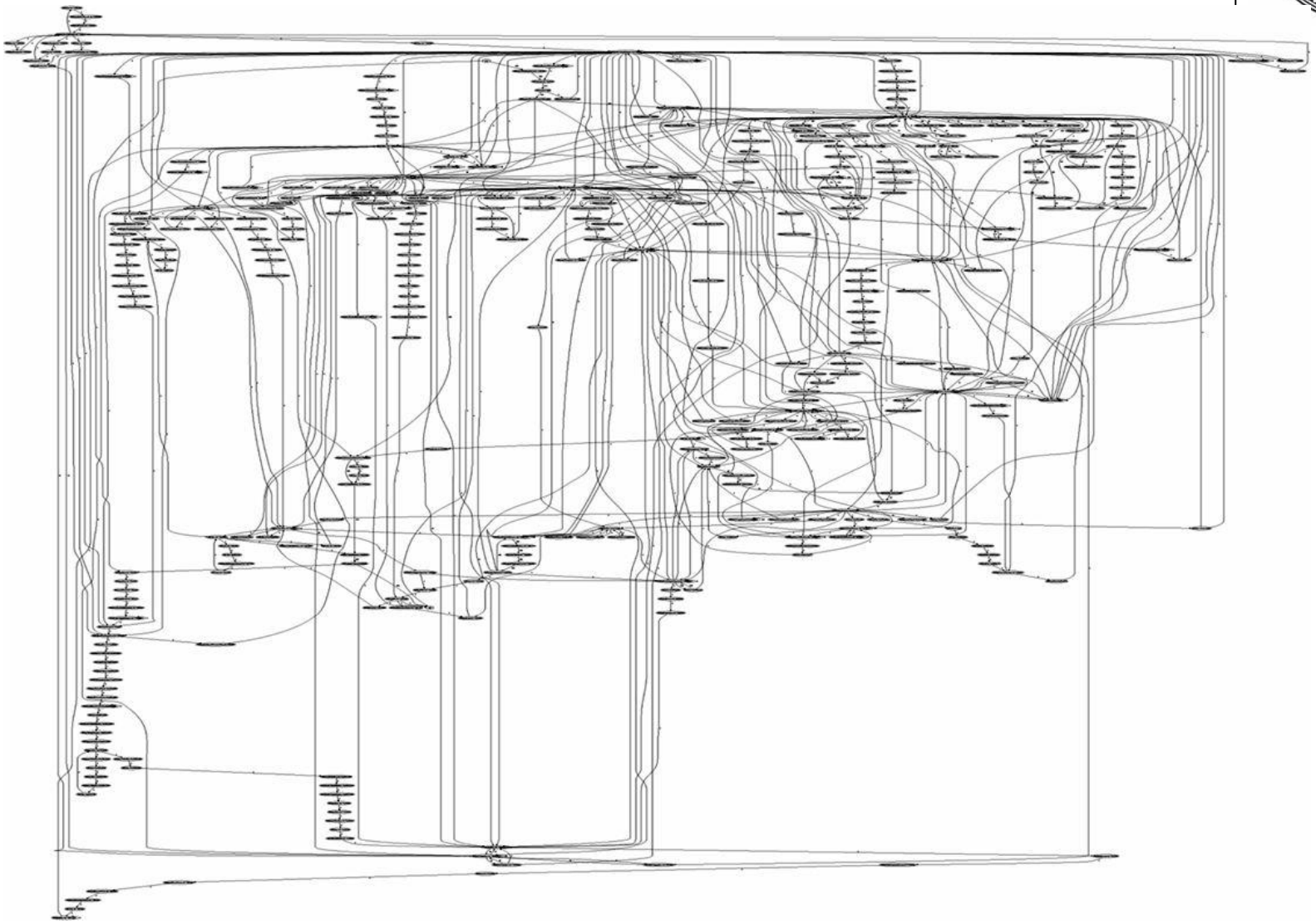
1.2. Estructuras de control

- Las estructuras de control permiten establecer el orden de ejecución y cambiar el flujo del programa dependiendo de los resultados de las acciones primitivas
- Tipos de estructuras de control
  - Salto
  - Secuencia
  - Bifurcación
  - Iteración
  - Invocación (llamada) a subrutinas
  - Tratamiento de excepciones

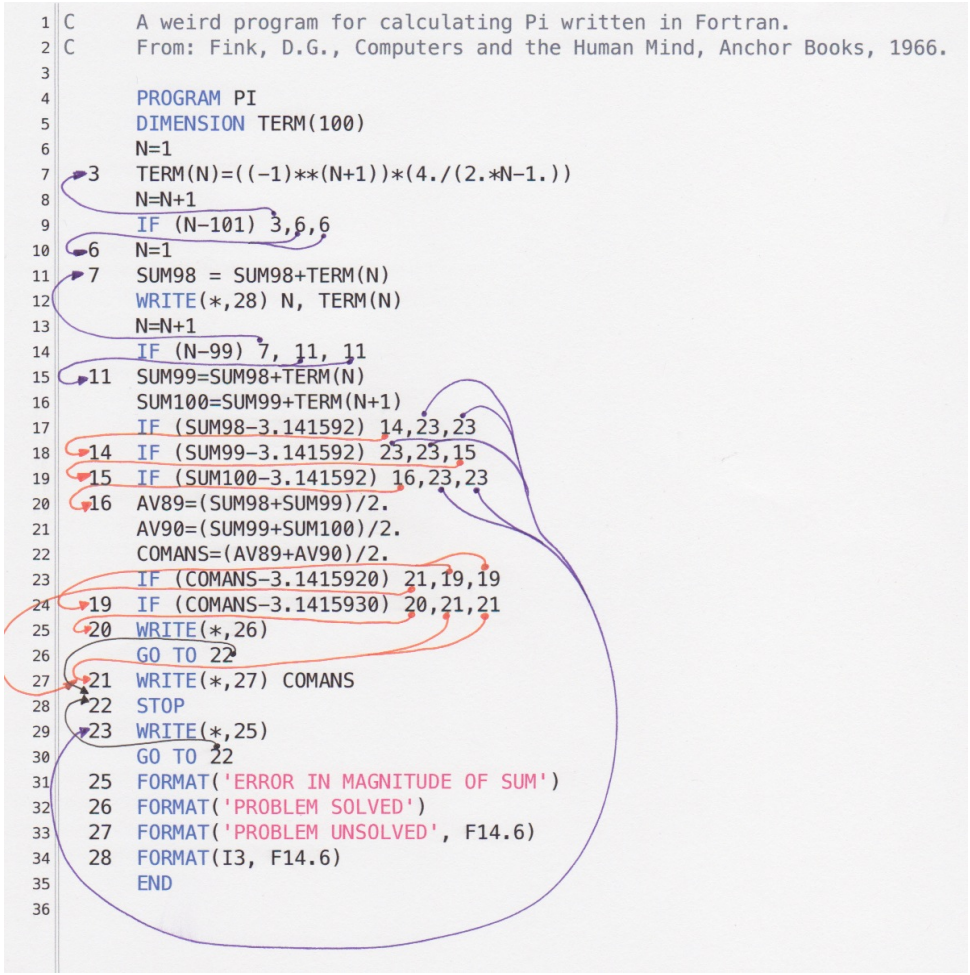
Saltos (goto)

- La estructura de control más básica es el salto condicional: Con ella es posible construir el resto de estructuras de control.
- Se necesita la posibilidad de definir posiciones de código (labels - etiquetas): `goto` *`etiqueta`*
- En algunos lenguajes es posible almacenar etiquetas en variables
- Tipos de saltos:
  - Condicionales: **`if`** *`cond`* **`then goto`** *`label`*
  - Computados (BASIC): **`goto`** *`i`*`*1000`
  - Modificables (COBOL): **`ALTER`** *`x`* **`TO PROCEED TO`** *`y`*

Spaghetti code



Spaghetti code



```
C     A weird program for calculating Pi written in Fortran.
C     From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.

      PROGRAM PI
      DIMENSION TERM(100)
      N=1
    3 TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
      N=N+1
      IF (N-101) 3,6,6
    6 N=1
    7 SUM98 = SUM98+TERM(N)
      WRITE(*,28) N, TERM(N)
      N=N+1
      IF (N-99) 7, 11, 11
   11 SUM99=SUM98+TERM(N)
      SUM100=SUM99+TERM(N+1)
      IF (SUM98-3.141592) 14,23,23
   14 IF (SUM99-3.141592) 23,23,15
   15 IF (SUM100-3.141592) 16,23,23
   16 AV89=(SUM98+SUM99)/2.
      AV90=(SUM99+SUM100)/2.
      COMANS=(AV89+AV90)/2.
      IF (COMANS-3.1415920) 21,19,19
   19 IF (COMANS-3.1415930) 20,21,21
   20 WRITE(*,26)
      GO TO 22
   21 WRITE(*,27) COMANS
   22 STOP
   23 WRITE(*,25)
      GO TO 22
   25 FORMAT('ERROR IN MAGNITUDE OF SUM')
   26 FORMAT('PROBLEM SOLVED')
   27 FORMAT('PROBLEM UNSOLVED', F14.6)
   28 FORMAT(I3, F14.6)
      END
```

Programación Estructurada

- Dijkstra, 1968: *"Goto statement considered harmful"*
- El (ab)uso de gotos crea código dificil de entender y de mantener.
- Se debe sustituir por un conjunto de estructuras de control privilegiadas
  - Versión **Dijkstra** → único punto de entrada y salida
  - Versión **Knuth** → Más permisiva: Único punto de entrada pero se permiten múltiples puntos de salida.
- Un lenguaje es (más) estructurado según la facilidad con que permita el uso de estructuras privilegiadas y su filosofía de programación:

BASIC → C → JAVA → EIFFEL

Estruc. Privileg. - Secuencia

- Expresan una secuencia temporal de ejecuciones de sentencias.
- Cada sentencia se ejecuta tras la finalización de la anterior.
- Distinción entre sentencias:
  - Separadores: Estilo PASCAL
  - Finalizadores: Estilo C
  - Indentación: Estilo Haskell, Python
- Agrupación:
  - Por corchetes: Estilo C
  - Por palabras reservadas: Estilo ALGOL
  - Por iniciadores y finalizadores: Estilo ADA

Estruc. Privileg. - Bifurcación

- Permite dividir la ejecución en dos o más lineas dependiendo del resultado de una expresión.
  - Condición lógica: Bifurcación simple o doble.
  - Expresión: Bifurcación múltiple (cascada → C, exclusiva → PASCAL, exhaustiva → EIFFEL).

```
switch(mes) {
  case 2: dias = 28; break;
  case 4: case 6: case 9:
  case 11: dias = 30; break;
  default: dias = 31;
}
```

```
case mes of
  1,3,5,7,8,10,12: dias := 31;
  4,6,9,11: dias := 30;
else dias := 28;
end;
```

```
inspect mes
  when 1,3,5,7,8,10,12 then dias := 31;
  when 4,6,9,11 then dias := 30;
  when 2 then dias := 28;
end;
```

Estruc. Privileg. - Iteración

- Bucles controlados por condición
  - Salida al principio (while)
  - Salida al final (repeat | do-while)
  - Salida en punto intermedio (ADA)

```
loop
   a := a+1
   exit when a > 10
   b := b+1
end loop
```

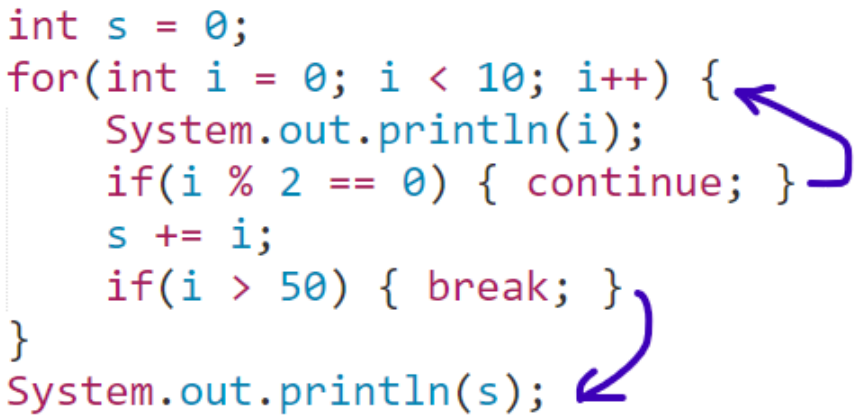
- Bucles controlados por índice (for)
- Bucles controlados por colección (foreach)
  - Recorren los elementos de una estructura de datos, o en general cualquier entidad que tenga definido un iterador sobre ella:

```
l = ['uno','dos']
for s in l:
   print(s)
```

```
int sum(int[] a) {
   int result = 0;
   for(int x : a) result += x;
   return result;
}
```

Ruptura de bucles

- Muchos lenguajes permiten modificar el flujo normal de un bucle:
  - Salida anticipada (break)
  - Paso a la siguiente iteración (continue)



Respecto a la estructuración:

- **Dijkstra**: Se permite continue, no se permite break.
- **Knuth**: Se permiten ambos.

Ejemplos (I)


```
int f(int n) {
    int j = 0;
    for(int i = 1; i < n; i++) {
        goto l2;
l1:     j++;
        if(j > 2*n) goto l4
l2:     printf("Dentro: %d", j);
        goto l3;
    }
l3: j++;
    printf("Fuera: %d", j);
    goto l1
l4: return j;
}
```

No Estructurado (Knuth y Dijkstra)

El bucle **for** tiene 2 entradas

son equivalentes

```
double f(double n) {
    double x = y = 1.0;
    if (n < 2.0) goto l2;
l1: if (x > n) goto l2;
    y *= x;
    x++
    goto l1
l2: return y;
}
```

Estructurado (Knuth y Dijkstra)

```
double f(double n) {
    double x = y = 1.0;
    if (n >= 2.0) {
        while (x <= n) {
            y *= x;
            x++
        }
    }
    return y;
}
```

Ejemplos (II)

```java
public int indice(String[] dic, String pal) {
    boolean existe = false;
    int i = 0;
    while(i < num && !existe) {
        existe = dic[i].equals(pal);
        if(!existe) { i++; }
    }
    if(existe) {
        return i;
    } else {
        return -1;
    }
}
```

Estructurado (Knuth y Dijkstra)

Nota: Si la última sentencia es una bifurcación en la que ambas ramas tienen un **return**, se considera un único punto de salida.

```java
public int indice(String[] dic, String pal) {
    for(int i = 0; i < num; i++) {
        if(dic[i].equals(pal)) { return i;}
    }
    return -1;
}
```

Estructurado según Knuth

No estructurado según Dijkstra
(bucle y función con 2 puntos de salida)

1.3. Subrutinas

- Equivalente a subprograma, función, procedimiento
- Las subrutinas proporcionan un mecanismo para encapsular código con los objetivos de:
  - Poder reutilizarlo
  - Poder invocarlo de una forma consistente
  - Elevar el nivel de abstracción del lenguaje
  - Permitir un diseño más elaborado de la aplicación (top-down)
- El paradigma procedimental añade más restricciones a las características que debe tener una subrutina.

Macros - *copybooks*

- El modelo más primitivo de definición de subrutinas
- Consiste en definir bloques de código que se insertarán en los puntos indicados del programa sustituyendo nombres de variables por otros
- COBOL: `COPY modulo REPLACING item-1 BY item-2`
- Otros lenguajes: Macros (ejemplo C)

```
#define NODIV(a,b) ((a) % (b) != 0)
...
while(NODIV(n-1,d) && d < 0) d+
+; ...
```

Preprocesador

```
while((n-1) % d != 0 && d < 0) d++;
```

Paradigma Procedimental

- Indica un modelo de organización de programas:
- Los programas se organizan como colecciones de subrutinas (procedimientos) relacionados mediante invocaciones (llamadas) entre ellas.
- Propiedades que deben cumplir:
  - **Encapsulamiento**: Las subrutinas se deben comportar como **cajas negras**, no es necesario conocer ningún detalle de su código para poder usarlas.
  - **Independencia**: Las subrutinas disponen de su propio ámbito (scope) de elementos (variables) y deben depender lo menos posible del exterior (variables globales)
  - **Interfaz**: Las subrutinas obtienen los valores y devuelven los resultados mediante mecanismos sintácticos bien definidos (parámetros, sentencia **return**)

Subrutinas: Modelo estándar

- **Ámbito**: Las subrutinas disponen de su propio espacio de memoria, independiente del resto, para almacenar sus variables y parámetros.
  - El ámbito se crea al invocar (llamar) a la subrutina y se destruye cuando ésta finaliza su ejecución.
  - Problema del **enmascaramiento**: Cuando los identificadores locales de una subrutina coinciden con identificadores de elementos globales a ella.
  - Subrutina **reentrante**: Cuando puede ser invocada sin necesidad de que haya terminado de ejecutarse una invocación anterior (recursividad, concurrencia).
- **Estructuración**: Las subrutinas tienen un único punto de entrada. No pueden existir gotos a subrutinas.

Paso de parámetros (I)

- Parámetro (formal): El identificador de la subrutina que representa un parámetro
- Argumento: El valor concreto de un parámetro con que se llama a una subrutina.
  - **Paso por valor:** El parámetro formal es una variable local de la subrutina. En la llamada se crea una copia del argumento y se asigna a esa variable local.
  - **Paso por referencia:** El parámetro formal es una variable local que almacena una referencia al argumento.
  - **Paso por variable:** El parámetro formal es un identificador con el que se renombra la variable que se usa como argumento.

Nota: En la familia C (C, C++, C#) se denomina llamada por referencia (&param en C, ref en C#) lo que en realidad es un paso por variable.

Paso de parámetros (II)

- Muchos lenguajes no disponen de paso por variable (C, Java, Python, Javascript). Pascal, C++, C# si lo tienen.
- Si existe el paso por variable, se puede usar como otro medio (además del return) de devolver resultados.
- La mayoría de lenguajes no disponen de mecanismos sintácticos para distinguir entre paso por valor y paso por referencia, se usa uno u otro según el **tipo de datos** del parámetro:
  - Los tipos de datos primitivos (int, double, char, etc.) se suelen pasar **por valor**.
  - Los objetos se suelen pasar **por referencia**.
  - Arrays y strings depende del lenguaje (típicamente **por referencia**)

Paso de parámetros (III)


```
function prueba(val a, var b, var c, ref d, ref e) {
  a[0] = 10; b[0] = 10; d[0] = 10;
  c = [3,2,1];
  e = [3,2,1];
}
...
a = [1,2]; b = [1,2]; c = [1,2]; d = [1,2]; e = [1,2];
prueba(a,b,c,d,e);
print a → "[1,2]"
print b → "[10,2]"
print c → "[3,2,1]"
print d → "[10,2]"
print e → "[1,2]"
```

Paso de parámetros (IV)

- Parámetros de entrada salida: Algunos lenguajes (ADA, Delphi) permiten indicar la semántica del parámetro, ocultando el mecanismo concreto:
  - Parámetro de entrada (in, const): Parámetro es variable interna (local), accesible pero no asignable.
  - Parámetro de salida (out): Parámetro representa a variable externa, asignable pero no accesible.
  - Parámetro de entrada salida (in out): Parámetro es variable externa, asignable y accesible.
- Asignación parámetro-argumento:
  - **Posicional:** La más habitual. El orden de declaración de los parámetros dicta el orden de asignación de argumentos.
  - **Nominativa:** El nombre de los parámetros puede ser usado para asignar los argumentos en la llamada. Es util en conjunción con argumentos por defecto. (ADA, Python)

Paso de parámetros (V)


```
procedure ecuacion(A,B,C: in Float; R1,R2: out Float; Valido: out Boolean)
is D: Float;
begin
  D := B**2 - 4.0*A*C;
  if D < 0.0 or A = 0.0 then
    Valido := False; R1 := 0.0; R2 := 0.0;
  else
    Valido := True; R1 := (-B+Sqrt(D))/(2.0*A); R2 := (-B-Sqrt(D))/(2.0*A);
  end if
end ecuacion
```

```
def logaritmo(valor, base = 10.0):
  return math.log(valor,base)
...
a = logaritmo(100.0,2.0) # Posicional
b = logaritmo(100.0)     # Posicional con base = 10.0
c = logaritmo(base=10.0,valor=2.0) # Nominativa
```

Paso de parámetros (VI)

- Parámetros con argumentos por defecto: Algunos lenguajes permiten indicar un valor por defecto para uno o varios parámetros, pudiendo omitir el argumento si su valor no va a ser distinto que el valor por defecto.
- Parámetros múltiples (indeterminados): Hay situaciones en las que puede ser conveniente declarar una subrutina que acepte un número arbitrario de argumentos, cuyo tipo se desconoce:
  - **Python:** Arbitrary Argument Lists: Parámetros empaquetados en una tupla, mediante el operador de (des)empaquetamiento, *
  - **C, C#, Java**: Open Array Parameters: Argumentos de tipos idénticos.
  - **C, C++:** Librería stdarg.h →macros.

Paso de parámetros (VII)

Cabecera de función python de la librería fastai:

```
vision_learner (dls, arch, normalize=True, n_out=None, pretrained=True,
                loss_func=None, opt_func=<function Adam>, lr=0.001,
                splitter=None, cbs=None, metrics=None, path=None,
                model_dir='models', wd=None, wd_bn_bias=False,
                train_bn=True, moms=(0.95, 0.85, 0.95), cut=None,
                init=<function kaiming_normal_>, custom_head=None,
                concat_pool=True, pool=True, lin_ftrs=None, ps=0.5,
                first_bn=True, bn_final=False, lin_first=False,
                y_range=None, n_in=3)
```

Ejemplo de función con parámetros arbitrarios:

```
def suma(*params):
    tot = 0
    for x in params:
        tot += x
    return tot
```

```
suma(1,2)       → 3
suma(1,2,3,4)   → 10
suma([1,2,3])   → Error
suma(*[1,2,3]) → 6
```

Tratamiento de excepciones

- Interpretaremos un **fallo** no como un error de programación, sino como una condición “inesperada” en el flujo “normal” del programa.
- Existen instrucciones que pueden fallar: divisiones por cero, apertura de ficheros bloqueados, entradas de usuario erroneas, etc.
- Cuando se incorpora a un programa el control de los posibles fallos, la lógica del programa puede complicarse exponencialmente.
- Es común que no coincidan el punto en que se detecta un fallo con el punto en el que puede tratarse adecuadamente (subrutinas distintas)

T.E.: Ejemplo


```
function proc_fichero(nomfich) {
  abrir fichero
  bucle: lee linea
         proc_linea(linea)
  cerrar fichero // Se debe ejecutar pase lo que pase
}

function proc_linea(linea) {
  bucle: extrae dato
         proc_dato(dato)
}

function proc_dato(dato) {
  .. aquí puede detectarse un error..
}
```

T.E.: Ejemplo


```
function proc_fichero(nomfich) {
  abrir fichero
  bucle: lee linea mientras no err2
         proc_linea(linea,err2)
  si err2 entonces .. tratar fallo ..
  cerrar fichero // Se debe ejecutar pase lo que pase
}

function proc_linea(linea, var error) {
  bucle: extrae dato mientras no err1
         proc_dato(dato, err1)
         if err1 then error = err1
}

function proc_dato(dato, var error) {
  if fallo then error = ...
}
```

T.E.: Ejemplo


```
function proc_fichero(nomfich) {
  abrir fichero
  try
    bucle: lee linea
           proc_linea(linea)
  except
    .. tratar fallo ..
  finally
    cerrar fichero // Se debe ejecutar pase lo que pase
}

function proc_linea(linea) {
  bucle: extrae dato
         proc_dato(dato)
}

function proc_dato(dato) {
 .. si hay un fallo se genera una excepción ..
}
```

Tratamiento de excepciones

- Los fallos generan excepciones, las cuales se pueden tratar (try-except|catch) o no.
- Si no se tratan se propagan a la subrutina que llamó a aquella donde se produjo el error.
- Es posible indicar código "a prueba de fallos" (finally)
- Ventajas:
  - No es necesario modificar subrutinas si no van a tratar los errores que detecten.
  - Protocolo uniforme de detección de fallos (en lenguajes OO las excepciones suelen ser objetos con información extra)
- Existen lenguajes que obligan a declarar (throws) las excepciones que detectan pero no tratan (Java)

Implementación Habitual

- Las excepciones se suelen representar por clases, existiendo una por categoría de fallo (jerarquía).
- El código protegido se rodea por try-except (Python) o try-catch (Java, C, C++, C#)
- A continuación existen bloques de tratamiento (except|catch) que permiten diferenciar las posibles excepciones. Pueden existir bloque genéricos (else).
- Puede existir bloque de finalización (finally),
- El programador puede definir nuevas excepciones y generarlas en código (raise|throw),
- Puede ser obligatorio que las subrutinas declaren las excepciones no tratadas (throws en Java).

Ejemplo en Python


```
try:
  fich = open(nomfich)
  # Leer fichero...
except FileNotFoundError:
  print("Fichero no existe")
except BlockingIOError:
  print("Fichero ya abierto")
except Exception as err:
  print("Error desconocido: "+err)
  raise err
finally:
  fich.close()
```

Ejemplo en Java


```
void tratarFichero(String nomFich) throws EShareViolation {
  try {
    fich = OpenFile(nomfich);
    // Leer fichero...
  } catch(EFileNotFound) {
    showMessage('Fichero no existe');
  } catch(EPermissionDenied e) do
    logFile(e.data);
    throw new EFalloFichero(..); }
  finally {
    CloseFile(fich);
  }
}
```

Subrutinas: clase *Aberrantia*

- El modelo estándar establece un único punto de entrada, uno (o varios) puntos de salida de las subrutinas.
- También establece que el *ámbito* (variables locales, parámetros) de las subrutinas sólo está activo (existe) mientras la llamada.
- Los saltos (gotos) sólo se permiten para retornar de una subrutina hacia el código del llamador.
- Pero existen otras posibilidades...

Aberrantia : Continuations

- Una *continuation* es un mecanismo que permite almacenar el estado de ejecución del programa en un momento dado (variables locales, llamadas a subrutinas, punto de ejecución) de tal forma que pueda ser invocado con posterioridad, restaurando el programa al estado en que se salvó.
- Filosóficamente una *continuation* representa el resto de la computación en un momento dado.
- Tambien se pueden contemplar como un GOTO extendido
  - Scheme: call-cc.
  - C: setjmp

Ejemplo de continuation


```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;                 // Variable continuation

void segundo(void) {
    printf("segundo\n");
    longjmp(buf,1);                 // retorna haciendo que setjmp dev. 1
}

void primero(void) {
    segundo();
    printf(“primero\n");            // no se ejecuta
}

int main() {
    if (setjmp(buf) == 0) {         // setjmp devuelve inicialmente 0
        primero();
    } else {
        printf(“principal\n");
    }
}
```

```
segundo
principal
```

Aberrantia : Coroutines

- Una corutina es una extensión del concepto de subrutina en la cuál es posible tener múltiples puntos de entrada, dependiendo de la historia de llamadas a la corutina.
- Una instrucción (tipicamente **yield**) permite retornar de la coroutina indicando que la siguiente llamada debe continuar la ejecución en ese punto.
  - Para ello una coroutina debe mantener su ámbito activo entre distintas llamadas.
  - Las corutinas son útiles para implementar iteradores, listas infinitas, tuberías, multitarea cooperativa, etc.
  - Pueden implementarse mediante continuations
  - Existen en Modula-2, Python, Ruby, Perl, C#, ...

Ejemplo de corutina


```
function enumerador(n: integer) : integer;
var i : integer;
begin
  for i := 1 to n do
    yield(i);
  enumerador := 0
end;

begin
  writeln(enumerador(3)); → 1
  writeln(enumerador(3)); → 2
  writeln(enumerador(2)); → 3
  writeln(enumerador(2)); → 0
  writeln(enumerador(2)); → 1
  writeln(enumerador(2)); → 2
end.
```

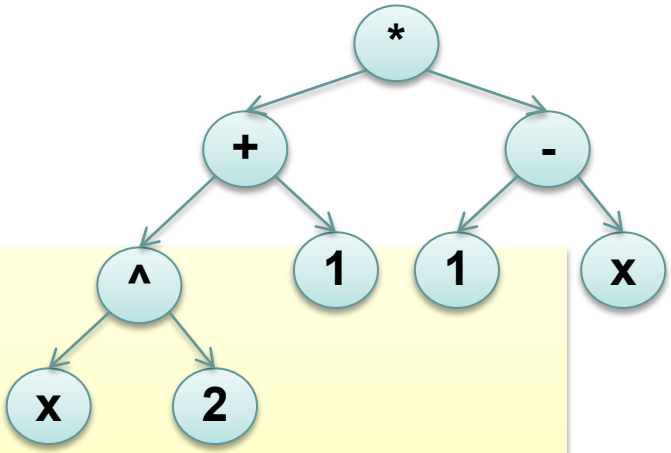
Coroutinas y Generadores

- Una utilidad de las corutinas es la de facilitar la creación de generadores: Funciones que proporcionan un flujo (stream) de datos.
- Los generadores sirven para la creación de iteradores: Objetos que mantienen información sobre el estado de un recorrido sobre una estructura.
- En Python (y otros lenguajes):
  - Existe una sintaxis para bucles que recorren secuencialmente datos: `for elem in iterador`
  - Se puede recorrer cualquier estructura que tenga definido el método **`iter()`**
  - El cuál devuelve un objeto que debe definir el método **`next()`**.
  - O bien *iterador* puede ser una coroutina:

Corutinas y Generadores

Recorrido de un arbol binario en **inorden**:

- Recorrer en inorden la rama izquierda
- Pasar por la raiz
- Recorrer en inorden la rama derecha




```
def inorden(lis):
  if len(lis) == 1: yield lis[0]
  else:
    yield '('
    for elem in inorden(lis[0]): yield elem
    yield lis[1]
    for elem in inorden(lis[2]): yield elem
    yield ')'

expr = [[['x'],'^',[2]],'+',[1]],'*',[[1],'-',['x']]]
for x in inorden(expr): print(x, end='')
(((x^2)+1)*(1-x))
```

Aberrantia : Closures

- Representan esquemas de funciones en las que aparecen variables no locales, y es posible instanciarlas (crear funciones basadas en ese esquema) con valores concretos de esas variables.
- Es necesario disponer de funciones que puedan devolver funciones.

```
function derivative(f, dx) {
  return function (x) {
    return (f(x+dx)-f(x))/dx;
  };
}
```

```
function logb(base) {
  return function (x) {
    return (ln(x)/ln(base));
  };
}
...
log2 = logb(2);
log2(16) → 4
```

1.4. Paradigma Modular

- Indica un modelo de organización de programas:
- Los programas se componen de ámbitos llamados módulos, los cuales pueden contener tipos de datos y subrutinas, y se relacionan entre sí por el uso de patrones de tipos y subrutinas.
- Establece un nuevo nivel de organización por encima del paradigma procedimental (colecciones de subrutinas).
  - Packages de Java y Python
  - Assemblies de C#
- En algunos textos el nivel de módulo se identifica con la subrutina (igual al procedimental).

Módulos: Objetivos

- Reusabilidad: El agrupar tipos de datos y subrutinas de acuerdo a su funcionalidad, para poder trasladarlo de un programa a otro.
- Organización: El poder dividir una aplicación compleja en partes lo más independientes posibles que se puedan desarrollar, probar y compilar por separado.
- Bibliotecas: La creación de bibliotecas de código que se cuya funcionalidad se pueda incorporar a cualquier aplicación.
- Diseño: La disciplina de la ingeniería de software aplica técnicas de diseño a la descomposición en módulos, las cuales intentan maximizar su cohesión y minimizar su acoplamiento.

Módulos: Propiedades

- Independencia: Los módulos son unidades de compilación separadas. Un objetivo esencial es minimizar sus dependencias respecto a otros módulos.
- Separación Interfaz - Implementación: Los módulos exponen una parte pública (interfaz), consistente en tipos (abstractos) y cabeceras de subrutinas, y una parte privada (implementación), oculta al exterior.
  - El objetivo es poder modificar la implementación sin que afecte a los usuarios del módulo (sólo pueden basarse en la interfaz)
- Espacios de nombres (namespaces): Mecanismo para evitar los problemas de enmascaramiento y poder organizar una jerarquía de módulos.

Espacios de nombres

- Enmascaramiento: Si un programa incorpora dos módulos (desarrollados por compañías distintas) con subrutinas que tengan el mismo identificador, una de ellas estará enmascarada por la otra.
- Los espacios de nombres imponen que las referencias a los elementos de los módulos incluyan el nombre (jerárquico) del módulo.

```
import audio
import multimedia.video

audio.play("ping.wav")
multimedia.video.play("300.avi")
```

2. SISTEMA DE TIPADO

2.1. Conceptos

- **Tipo de Dato**: Representa un conjunto de valores junto con las operaciones que se pueden realizar sobre ellos.
  - Parte abstracta: Define únicamente la forma de construir valores de ese tipo y los nombres de las operaciones junto con las ecuaciones que indican la forma en actúan.
  - Parte de implementación: Define la representación de los datos (el formato en que se almacenan los valores de ese tipo de dato en memoria) y los algoritmos de las operaciones (la manera en que trabajan las operaciones para esa representación concreta)
- **Tipo Abstracto de Datos (TAD)**: Definido únicamente por la parte abstracta.

Tipos de datos: Objetivos

- Determinación del espacio de almacenamiento de los valores (optimización)
- Traducción de operaciones a las instrucciones adecuadas del procesador (optimización)
- Detección de errores provocados por operaciones aplicadas a valores erroneos (seguridad)
- Síntaxis que permita la creación de tipos estructurados y o definidos por el programador (modularidad a nivel de tipado)
- Los tipos de datos son un elemento fundamental en el diseño de aplicaciones (abstracción, documentación).

2.1. Tipos de datos habituales

- Una división principal puede ser entre tipos primitivos (no se pueden descomponer) y estructurados (compuestos por partes con sentido propio).
- Respecto al sentido de la sentencia de asignación, pueden clasificarse en valores y referencias.
- Respecto al método de creación de valores pueden ser literales o dinámicos.
- Los lenguajes orientados a objetos pueden tener un tratamiento uniforme (todos los tipos de datos son clases, ej. Smalltalk) o distinguir entre tipos elementales y clases (C, C++, C#, Java).

Punteros y Referencias

- El tipo puntero representa una dirección de memoria.
- Ningún lenguaje de alto nivel permite un acceso indiscriminado a memoria, por lo tanto el uso de los punteros se restringe a referenciar elementos del programa.
- Tienen un tipo de datos asociado, y existen operadores para acceder al valor referenciado.
  - En C el operador * devuelve el valor de la posición apuntada. El operador & (**dereferencia**) sirve para obtener la dirección de una variable.
  - En C existe aritmética de punteros.
- Las referencias establecen un nivel de abstracción mayor: Se trabaja directamente con ellas, sin usar operadores de dereferencia.

Valores y Referencias (I)

- En la mayoría de lenguajes, dependiendo del tipo de dato, las variables de ese tipo pueden almacenar valores o referencias.
- En la mayoría de lenguajes los tipos primitivos suelen almacenarse como valores, y los tipos estructurados (arrays, strings, objetos) como referencias.
- Los lenguajes C, C++, C# disponen de un tipo estructurado (**struct**) que tiene **semántica de valor**.
- **Autoboxing** (Java, C#): Mecanismo para convertir automáticamente, cuando sea necesario, un tipo primitivo (semántica de valor) a un objeto (semántica de referencia).

Valores y Referencias (II)

- El tipo de almacenamiento afecta a la interpretación de las siguientes operaciones, entre otras:
  - **Asignación**: Una asignación por valor produce la creación de una copia del valor, una asignación por referencia hace que dos variables se refieran al mismo valor.
  - **Llamada a subrutinas**: En lenguajes con sintaxis uniforme de paso de parámetros (C, Java, Python, Javascript, ...) la posible modificación de los argumentos tras ser enviados a la subrutina depende de si son valores o referencias.
  - Operaciones de **comparación**: Es necesario conocer en el caso de referencias si se comparan las referencias o su contenido.

Ejemplo (C#)

```
class Punto {
    public int x, y;

    public Punto(int x, int y) {
        this.x = x; this.y = y;
    }

    public override string ToString() {
        return $"[{x}, {y}]";
    }
}

public class Program {

    static void Cambia(Punto p) { p.x = -1; }

    static void Main(string[] args) {
        Punto p1 = new Punto(1, 2);
        Punto p2 = new Punto(1, 2);
        Console.WriteLine(p1.Equals(p2));
        p2 = p1;
        p2.x = 42;
        Console.WriteLine($"{p1} | {p2}");
        Cambia(p1);
        Console.WriteLine($"{p1} | {p2}");
        Console.ReadLine();
    }
}
```

```
False
[42, 2] | [42, 2]
[-1, 2] | [-1, 2]
```

```
struct Punto {
    public int x, y;

    public Punto(int x, int y) {
        this.x = x; this.y = y;
    }

    public override string ToString() {
        return $"[{x}, {y}]";
    }
}

public class Program {

    static void Cambia(Punto p) { p.x = -1; }

    static void Main(string[] args) {
        Punto p1 = new Punto(1, 2);
        Punto p2 = new Punto(1, 2);
        Console.WriteLine(p1.Equals(p2));
        p2 = p1;
        p2.x = 42;
        Console.WriteLine($"{p1} | {p2}");
        Cambia(p1);
        Console.WriteLine($"{p1} | {p2}");
        Console.ReadLine();
    }
}
```

```
True
[1, 2] | [42, 2]
[1, 2] | [42, 2]
```

2.2. Tipado Estático/Dinámico

- Tipado estático: La comprobación de tipos se realiza en tiempo de compilación.
  - La mayoría de errores de tipado se detectan por el compilador
  - Es necesario declarar el tipo de las variables
  - Habitualmente asociado con valores literales y variables de tipo prefijado
- Tipado dinámico: La comprobación de tipos se realiza en tiempo de ejecución.
  - No es necesario declarar el tipo de las variables, y normalmente su tipo puede cambiar.
  - Asociado con valores dinámicos.
  - Puede ser dificíl detectar errores: Se deben recorrer todas los posibles caminos de ejecución.

2.3. Tipado Fuerte .. Débil

- La fortaleza del sistema de tipado tiene relación con la permisividad que el lenguaje manifiesta respecto a la conversión de valores de tipos distintos.
- Elementos de juicio:
  - Jerarquía de subtipos del lenguaje
  - Ocurrencia de conversiones automáticas (implícitas)
  - Ámbito de aplicación de las conversiones explícitas (downcasting)
  - Posibilidad de conversiones que afecten al tipo subyacente de punteros
- El tipado estático suele ser más fuerte que el dinámico, aunque no tiene porque ser una regla general.

2.4. Tipado Seguro .. Inseguro

- Un lenguaje se considera seguro si no permite operaciones o conversiones de tipos que puedan generar errores o estados no definidos.
- Típicamente está relacionado con la posibilidad de acceso fuera de rango a memoria:
  - Comprobación de índices de arrays
  - Acceso a elementos dinámicos que no existen
  - Conversión a tipos de datos más amplios
- También tiene que ver con la seguridad de las operaciones aritméticas
  - Pérdida de precisión
  - Detección de sobrepasamiento

Tipos Primitivos - Conversión

- Se suele considerar que no disminuyen la **fortaleza** conversiones implícitas que conviertan a un valor en otro de un tipo más general (en expresiones que mezclen a valores de tipos numéricos distintos).
  - Ejemplos: unsigned → signed, integer → real, etc.
- Respecto a la **seguridad**, existen dos restricciones:
  - No se deben permitir conversiones explicitas de referencias o punteros entre tipos de tamaños distintos:

```
char c;
int *i = (int *) &c;
*i = 32000;
```

  - Se debe convertir implicitamente al tipo más general que pueda representar al resultado.

```
unsigned int val,lim;
// Si valor cae por debajo de lim-10, actuar
if(lim-val > 10) { ... }
```

Arrays

- Estructura definida más por eficiencia que por utilidad (suelen ser más útiles las listas y diccionarios).
- Acceso aleatorio a elementos (directo, indexado).
- Los elementos son del mismo tipo.
- Producto cartesiano entre tipo índice y tipo valor.
- **Seguridad**: Comprobación en tiempo de ejecución de acceso con índice en el rango adecuado.
  - Responsabilidad del programador: C, C
  - Forzosa: Java, C#, Python, Javascript