

# Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Departamento de Computación - FCEyN - UBA

Solucionando problemas con una computadora

1

## IP - AED I:

### IP - AED I:

3

### IP - AED I: Régimen de aprobación

- ▶ Con nota numérica
  - ▶ Parcial individual de programación en Haskell en computadora.
  - ▶ Parcial individual de programación en Python en computadora.
  - ▶ Un TP grupal de programación en Haskell + Testing
  - ▶ Todo se aprueba con nota igual o mayor a 6.
- ▶ Criterio de aprobación de la materia
  - ▶ TP y notas en parciales mayor o igual que 8, sin recuperatorios: promoción directa, queda el promedio de notas
  - ▶ TP y notas en parciales mayor o igual que 8, con algún recuperatorio: final oral (coloquio)
  - ▶ TP y notas en parciales mayor o igual que 7: final oral (coloquio)
  - ▶ La instancia de Coloquio, sólo es válida hasta las mesas de finales de Julio y Agosto 2024. Luego de esas fechas, se deberá dar final escrito
  - ▶ TP y notas en parciales mayor o igual que 6: final escrito
  - ▶ Ambos parciales y el TP tienen instancias de recuperatorio
    - ▶ Los recuperatorios son sólo para quienes no hayan aprobado la instancia anterior (no se pueden presentar para levantar nota si aprobaron)
  - ▶ Cada nota, de cada instancia de evaluación, tiene que cumplir el criterio
    - ▶ Ej: Si el promedio de notas es 7.99: se debe rendir coloquio

4

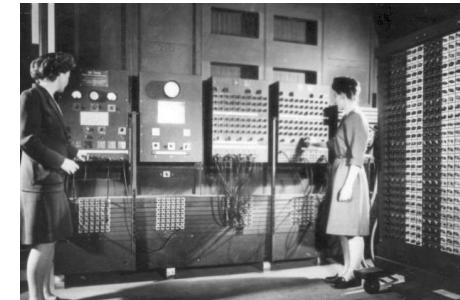
# Introducción a la Programación - AED I

**Objetivo:** Aprender a programar en **lenguajes funcionales** y en **lenguajes imperativos**.

- ▶ **Especificar** problemas.
  - ▶ Describirlos de manera tal que podemos construir y probar una solución
- ▶ Pensar **algoritmos** para resolver los problemas.
  - ▶ En esta materia nos concentramos en problemas para **tratamiento de secuencias** principalmente.
- ▶ Empezar a **Razonar** acerca de estos algoritmos y programas.
  - ▶ Veremos conceptos de testing.
- ▶ Escribir programas que implementen los algoritmos que pensamos.
  - ▶ Vamos a usar dos lenguajes de programación bien distintos para esto.

5

## ¿Qué es una computadora?



- ▶ Una **computadora** es una máquina electrónica que procesa datos automáticamente de acuerdo con un programa almacenado en memoria.
  - ▶ Es una **máquina** electrónica.
  - ▶ Su función es **procesar datos**.
  - ▶ El procesamiento se realiza en forma **automática**.
  - ▶ El procesamiento se realiza siguiendo un **programa**.
  - ▶ Este programa está **almacenado** en una memoria interna.

6

## ¿Qué es un algoritmo?

- ▶ Un **algoritmo** es la descripción de los pasos precisos para resolver un problema a partir de datos de entrada adecuados.
  1. Es la **descripción** de los pasos a realizar.
  2. Especifica una sucesión de **instrucciones primitivas**.
  3. El objetivo es resolver un **problema**.
  4. Un algoritmo típicamente trabaja a partir de **datos de entrada**.

7

## Ejemplo: Un Algoritmo

- ▶ **Problema:** Encontrar todos los números primos menores que un número natural dado  $n$
- ▶ **Algoritmo:** Criba de Eratóstenes (276 AC - 194 AC)
  - Escriba todos los números naturales desde 2 hasta a  $n$
  - Para  $i \in \mathbb{Z}$  desde 2 hasta  $\lfloor \sqrt{n} \rfloor$
  - Si  $i$  no ha sido tachado, entonces
    - Para  $j \in \mathbb{Z}$  desde  $i$  hasta  $\lfloor \frac{n}{i} \rfloor$  haga lo siguiente:
      - Si no ha sido tachado, tachar el número  $i \times j$
- ▶ **Resultado:** Los números que no han sido tachados son los números primos menores a  $n$

8

## ¿Qué es un programa?

- ▶ Un **programa** es la descripción de un algoritmo en un lenguaje de programación.
  1. Corresponde a la implementación concreta del algoritmo para ser ejecutado en una computadora.
  2. Se describe en un lenguaje de programación.

9

## Ejemplo: Un Programa (en Haskell)

Implementación de la Criba de Eratóstenes en el lenguaje de programación Haskell

```
erastotenes :: Int → [Int]
erastotenes n = erastotenes_aux [x|x ← [2..n]] 0

erastotenes_aux :: [Int] → Int → [Int]
erastotenes_aux lista n
| n == length lista-1 = lista
| otherwise = erastotenes_aux lista_filtrada (n+1)
  where lista_filtrada = [x|x ← lista ,(x `mod` lista!!n) /= 0 || x==lista !!n]
```

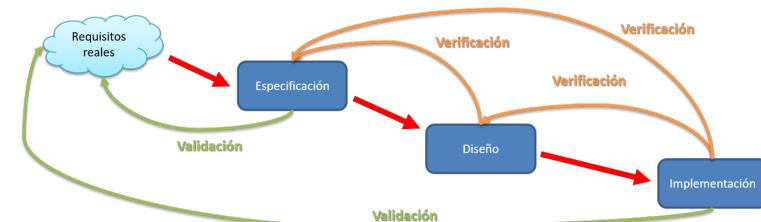
10

## Especificación, algoritmo, programa

1. **Especificación:** descripción del problema a resolver.
  - ▶ ¿Qué problema tenemos?
  - ▶ Habitualmente, dada en lenguaje formal.
  - ▶ Es un contrato que da las propiedades de los datos de entrada y las propiedades de la solución.
2. **Algoritmo:** descripción de la solución escrita para humanos.
  - ▶ ¿Cómo resolvemos el problema?
  - ▶ Puede existir sin una computadora.
3. **Programa:** descripción de la solución para ser ejecutada en una computadora.
  - ▶ También, ¿cómo resolvemos el problema?
  - ▶ Pero descripto en un lenguaje de programación.
  - ▶ Requiere una computadora para ejecutarse.

11

## Problema, especificación, algoritmo, programa



Dado un problema a resolver (de la vida real), queremos:

- ▶ Poder **describir** de una manera clara y única (especificación)
  - ▶ Esta descripción debería poder ser **validada** contra el problema real
- ▶ Poder **diseñar** una solución acorde a dicha especificación
  - ▶ Este diseño debería poder ser **verificado** con respecto a la especificación
- ▶ Poder implementar un programa acorde a dicho diseño
  - ▶ Este programa debería poder ser **verificado** con respecto a su especificación y su diseño
  - ▶ Este programa debería ser la solución al problema planteado

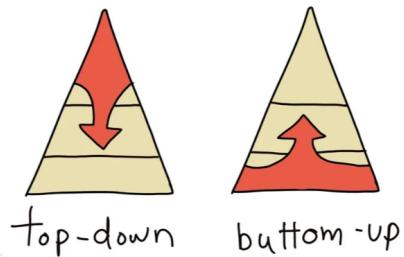
12

## También hablaremos de cómo encarar problemas...

O partir el problema en problemas más chicos...

Los conceptos de modularización y encapsulamiento siempre estarán relacionados con los principios de diseño de software. La estrategia se puede resumir en:

- ▶ Descomponer un problema grande en problemas más pequeños.
- ▶ Componerlos y obtener la solución al problema original.
- ▶ Estrategias *Top Down* versus *Bottom Up*.



13

## Diferenciaremos el QUÉ del CÓMO

- ▶ Dado un problema, será importante describirlo sin ambigüedades.
- ▶ Una buena descripción no debería condicionarse con sus posibles soluciones.
- ▶ Saber que dado un problema, hay muchas formas de describirlo y a su vez, muchas formas de solucionar... y todas pueden ser válidas!

14

## Especificación de problemas

- ▶ Una **especificación** es un contrato que define qué se debe resolver y qué propiedades debe tener la solución.
  - ▶ Define el **qué** y no el **cómo**.
- ▶ La especificación de un problema incluye un conjunto de **parámetros**: datos de entrada cuyos valores serán conocidos recién al ejecutar el programa.
- ▶ Además de cumplir un rol “contractual”, la especificación del problema es insumo para las actividades de ...
  - ▶ Testing,
  - ▶ Verificación formal de correctitud.
  - ▶ Derivación formal (construir un programa a partir de la especificación).

15

## Lenguaje naturales y lenguajes formales

### Lenguajes naturales

- ▶ Idiomas (castellano)
- ▶ Mucho poder expresivo (modos verbales –potencial, imperativo–, tiempos verbales –pasado, presente, futuro—, metáforas, etc. )
- ▶ Con un plus (conocimiento del contexto, suposiciones, etc)
- ▶ No se usan para especificar porque pueden ser ambiguos, y no tienen un cálculo formal.

### Lenguajes formales

- ▶ Sintaxis sencilla
- ▶ Limitan lo que se puede expresar
- ▶ Explicitan las suposiciones
- ▶ Relación formal entre lo escrito (sintaxis) y su significado (semántica)
- ▶ Tienen cálculo para transformar expresiones válidas en otras válidas

16

## Lenguajes formales. Ejemplos:

**Aritmética:** es un lenguaje formal para los números y sus operaciones. Tiene un cálculo asociado.

**Lógicas:** proposicional, de primer orden, modales, etc.

**Lenguajes de especificación:** sirven para describir formalmente un problema.

17

## Contratos

- ▶ Una especificación es un **contrato** entre el **programador** de una función y el **usuario** de esa función.
- ▶ **Ejemplo:** calcular la raíz cuadrada de un número real.
- ▶ ¿Cómo es la especificación (informalmente, por ahora) de este problema?
- ▶ Para hacer el cálculo, el programa debe recibir un número no negativo.
  - ▶ Obligación del usuario: no puede proveer números negativos.
  - ▶ Derecho del programador: puede suponer que el argumento recibido no es negativo.
- ▶ El resultado va a ser la raíz cuadrada del número recibido.
  - ▶ Obligación del programador: debe calcular la raíz, siempre y cuando haya recibido un número no negativo
  - ▶ Derecho del usuario: puede suponer que el resultado va a ser correcto

18

## Partes de una especificación (contrato)

1. **Encabezado**
2. **Precondiciones** o cláusulas “requiere”
  - ▶ Condición sobre los argumentos, que el programador da por cierta.
  - ▶ Especifica lo que **requiere** la función para hacer su tarea.
  - ▶ Por ejemplo: “el valor de entrada es un real no negativo”
3. **Postcondiciones** o cláusulas “asegura”
  - ▶ Condiciones sobre el resultado, que deben ser cumplidas por el programador siempre y cuando el usuario haya cumplido las precondiciones.
  - ▶ Especifica lo que la función **asegura** que se va a cumplir después de llamarla (si se cumplía la precondición).
  - ▶ Por ejemplo: “la salida es la raíz cuadrada del valor de entrada”

19

## Parámetros y tipos de datos

- ▶ La especificación de un problema incluye un conjunto de **parámetros**: datos de entrada cuyos valores serán conocidos recién al ejecutar el programa.
- ▶ Cada parámetro tiene un **tipo de datos**.
  - ▶ **Tipo de datos:** Conjunto de **valores** provisto de ciertas **operaciones** para trabajar con estos valores.
- ▶ Ejemplo 1: parámetros de tipo *fecha*
  - ▶ valores: ternas de números enteros
  - ▶ operaciones: comparación, obtener el año, ...
- ▶ Ejemplo 2: parámetros de tipo *dinero*
  - ▶ valores: números reales con dos decimales
  - ▶ operaciones: suma, resta, ...

20

## ¿Por qué escribir la especificación del problema?

- ▶ Nos ayuda a entender mejor el problema
- ▶ Nos ayuda a construir el programa
  - ▶ Derivación (Automática) de Programas
- ▶ Nos ayuda a prevenir errores en el programa
  - ▶ Testing
  - ▶ Verificación (Automática) de Programas

21

## Algoritmos y programas

- ▶ El primer paso será especificar un problema
- ▶ Luego, el objetivo será escribir un **algoritmo** que cumpla esa especificación
  - ▶ Secuencia de pasos que pueden llevarse a cabo mecánicamente
- ▶ Puede haber varios algoritmos que cumplan una misma especificación
- ▶ Una vez que se tiene el algoritmo, se escribirá el **programa**
  - ▶ Expresión formal de un algoritmo
  - ▶ Lenguajes de programación
    - ▶ Sintaxis definida
    - ▶ Semántica definida
    - ▶ Qué hace una computadora cuando recibe ese programa
    - ▶ Qué especificaciones cumple
    - ▶ Ejemplos: Haskell, Python, C, C++, C#, Java, Smalltalk, Prolog.

22

## Lenguajes de programación

- ▶ En palabras simples, es el conjunto de instrucciones a través del cual los humanos interactúan con las computadoras.
- ▶ Permiten escribir programas que son ejecutados por computadoras.

23

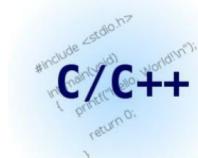
## Lenguajes de programación

No es tema de la materia... pero demos algún contexto por si se ponen a googlear...

- ▶ **Lenguaje Máquina:** son lenguajes que están expresados en lenguajes directamente inteligibles por la máquina, siendo sus instrucciones cadenas de 0 y 1.
- ▶ **Lenguaje de Bajo Nivel:** son lenguajes que dependen de una máquina (procesador) en particular (el más famoso probablemente sea Assembler)
- ▶ **Lenguaje de Alto Nivel** ([en la materia usaremos algunos de estos](#)): fueron diseñados para que las personas puedan escribir y entender más fácilmente los programas que escriben.



```
Push a      ; a → pila
Push b      ; b → pila
Load (c),R1 ; c → R1
Mult (S),R1 ; b*c → R1
Store R1,R2 ; R1 → R2
Add  (S),R1 ; a+b*c → R1
Store R1,(x) ; R1 → x
Add  #3,R2 ; 3+b*c → R2
Store R2,(y) ; R2 → y
```



```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
    return 0;
}
```

24

## Código fuente, compiladores, intérpretes...

No es tema de la materia... pero demos algún contexto por si se ponen a googlear...

- ▶ **Código Fuente:** es el programa escrito en un lenguaje de programación según sus reglas sintácticas y semánticas.
- ▶ **Compiladores e Intérpretes:** son programas *traductores* que toman un código fuente y generan otro programa en otro lenguaje, por lo general, lenguaje de máquina



25

## IDE (Integrated Development Environment)

Para escribir y ejecutar un programa alcanza con tener:

- ▶ Un editor de texto para escribir programas (Ejemplos: notepad, notepad++, gedit, etc)
- ▶ Un compilador o intérprete (según el lenguaje a utilizar), para procesar y ejecutar el programa

Pero un mundo mejor es posible...

## IDE (Integrated Development Environment)

Ventajas de utilizar algún IDE

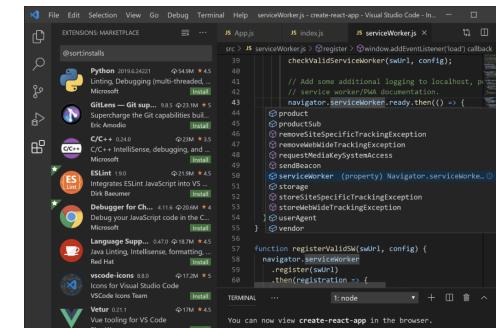
- ▶ Un editor está orientado a editar archivos mientras que un IDE está orientado a trabajar con proyectos, que tienen un conjunto de archivos.
- ▶ Integran un editor con otras herramientas útiles para los desarrolladores:
  - ▶ Permiten hacer destacado (*highlighting*) de determinadas palabras y símbolos dependiendo del lenguaje de programación.
  - ▶ Son capaces de verificar la sintaxis de los programas escritos (*linters*)
  - ▶ Generar vistas previas (*previews*) de cierto tipo de archivos (ej, archivos HTML para desarrollos web)
  - ▶ Suelen tener herramientas integradas (por ejemplo, el Android Studio tiene emuladores integrados)
  - ▶ Se pueden especializar según cada lenguaje particular
  - ▶ Permiten hacer depuración o *debugging*!

27

## IDE (Integrated Development Environment)

Algunos IDEs:

- ▶ Visual Studio (<https://visualstudio.microsoft.com/es/>)
- ▶ Eclipse (<https://www.eclipse.org/>)
- ▶ IntelliJ IDEA (<https://www.jetbrains.com/es-es/idea/>)
- ▶ Visual Code o Visual Studio Code (<https://code.visualstudio.com/>)
  - ▶ Es un editor que se “convierte” en IDE mediante *extensions*.
  - ▶ Lo utilizaremos para programar en Haskell y Python.



28

## Paradigmas

Existen diversos paradigmas de programación. Comunmente se los divide en dos grandes grupos:

- ▶ Programación Declarativa
  - ▶ Son lenguajes donde el programador le indicará a la máquina lo que quiere hacer y el resultado que desea, pero no necesariamente el cómo hacerlo.
- ▶ Programación Imperativa
  - ▶ Son lenguajes en los que el programador debe precisarle a la máquina de forma exacta el proceso que quiere realizar.

29

## Paradigmas

Cada grupo, se especializa según diferentes características

- ▶ Programación Declarativa: describe un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución.
- ▶ Paradigma Lógico: los programas están construidos únicamente por expresiones lógicas (es decir, son Verdaderas o Falsas).
- ▶ Paradigma Funcional: está basado en el modelo matemático de composición funcional. El resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produce el valor deseado.

30

## Paradigmas

Cada grupo, se especializa según diferentes características

- ▶ Programación Imperativa: describe la programación como una secuencia de instrucciones o comandos que cambian el estado de un programa.
- ▶ Paradigma Estructurado: los programas se dividen en bloques (procedimientos y funciones), que pueden o no comunicarse entre sí. Existen estructuras de control, que dirigen el flujo de ejecución: IF, GO TO, Ciclos, etc.
- ▶ Paradigma Orientado a Objetos: se basa en la idea de encapsular estado y comportamiento en objetos. Los objetos son entidades que se comunican entre sí por medio de mensajes.

31

## Paradigmas

Cada grupo, se especializa según diferentes características

- ▶ Programación Declarativa
  - ▶ Paradigma Lógico: PROLOG
  - ▶ Paradigma Funcional: LISP, GOFER, HASKELL.
- ▶ Programación Imperativa
  - ▶ Paradigma Estructurado: PASCAL, C, FORTRAN, FOX, COBOL
  - ▶ Paradigma Orientado a Objetos: SMALLTALK
- ▶ Lenguajes multiparadigma: lenguajes que soportan más de un paradigma de programación.
  - ▶ JAVA, PYTHON, .NET, PHP

32

## Paradigmas

Dado dos números, determinar si el segundo es el doble que el primero...

- ▶ Prolog:

```
% La siguiente regla es verdadera si X es el doble que Y  
es_doble(X, Y) :-  
    X is 2*Y.
```

- ▶ Haskell:

```
esDoble :: Integer -> Integer -> Bool  
esDoble x y = x == 2*y -- Verificamos si x es igual al doble de y
```

- ▶ Python:

```
def esDoble(x: int, y: int) -> bool:  
    if(x == 2*y):  
        return True  
    else:  
        return False
```

33

## Paradigmas

En la materia resolveremos (programaremos) problemas utilizando estos dos paradigmas:

- ▶ Paradigma Funcional

- ▶ Utilizaremos Haskell

- ▶ Paradigma Imperativo

- ▶ Utilizaremos Python

34

## Resolviendo problemas con una computadora

Durante el cuatrimestre, además de resolver problemas, veremos algunos aspectos sobre cómo resolverlos:

- ▶ Hablaremos de buenas prácticas
  - ▶ Utilizar nombres declarativos
  - ▶ Modularizar problemas
  - ▶ Uso de comentarios
  - ▶ y más...
- ▶ ¿De qué se trata esto?... veamos un adelanto

35

## Utilizar nombres declarativos

- ▶ Usar nombres **que revelen la intención** de los elementos nombrados. El nombre de una variable/funcióndebería decir todo lo que hay que saber sobre la variable/funcióndebería decir todo lo que hay que saber

1. Los nombres deben referirse a **conceptos** del dominio del problema.
2. Una excepción suelen ser las variables con scopes pequeños. Es habitual usar **i**, **j** y **k** para las variables de control de los ciclos.
3. Si es complicado decidirse por un nombre o un nombre no parece natural, quizás es porque esa variable o función no representa un concepto claro del problema a resolver.
4. Usar nombres **pronunciables!** No es buena idea tener una variable llamada **cdcptdc** para representar la “cantidad de cuentas por tipo de cliente”.

36

## Utilizar nombres declarativos

Ambos programas son el mismo... ¿Cuál se lee más claro?

```
int x = 0;
vector<double> y;
...
for(int i=0;i≤4;i=i+1) {
    x = x + y[i];
}

int totalAdeudado = 0;
vector<double> deudas;
...
for(int i=0;i≤conceptos;i=i+1) {
    totalAdeudado = totalAdeudado + deudas[i];
}
```

37

## Version Control Systems (CVSs)

- ▶ Permite organizar el trabajo en equipo
- ▶ Guarda un historial de versiones de los distintos archivos que se usaron
- ▶ Existen distintas aplicaciones: svn, cvs, hg, git

39

## Control de versiones



38

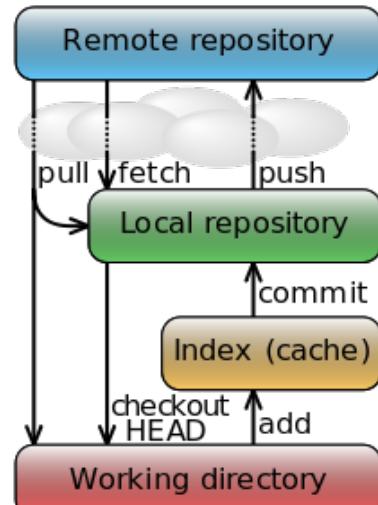
## Ejemplo: Git

- ▶ Sistema de control de versiones **distribuido**, orientado a **repositorios** y con énfasis en la eficiencia.
  1. Se tiene un servidor que permite el intercambio de los repositorios entre los usuarios.
  2. Cada usuario tiene una **copia local** del repositorio completo.
- ▶ Acciones: checkout, add, remove, commit, push, pull, status

40

## Git: Workflow

Git básico: pull, push, commit, checkout...



41

## Otros conceptos

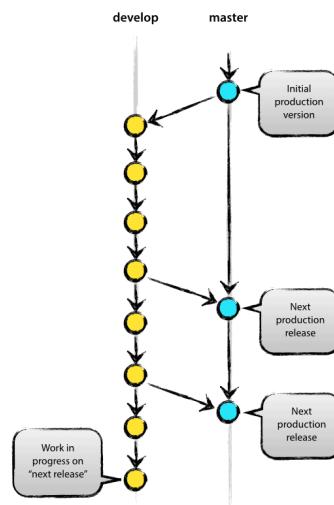
Git básico: branches y tags

- ▶ **Tag:** Nombre asignado a una versión particular, habitualmente para *releases* de versiones a usuarios.
- ▶ **Branch:** Línea paralela de desarrollo, para corregir un *bug* (error en el programa), trabajar en una nueva versión o experimentar con el código.
  - ▶ Master
  - ▶ Develop
  - ▶ Hotfixes

42

## Master/Main-develop

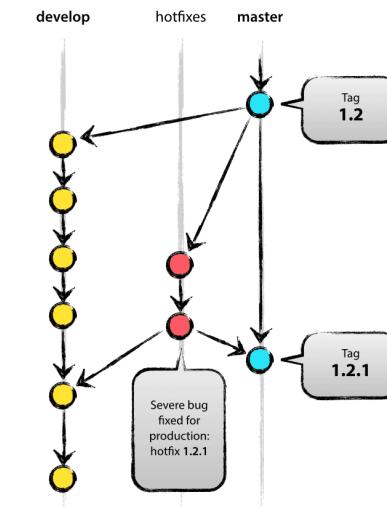
Convenciones



43

## Hotfixes

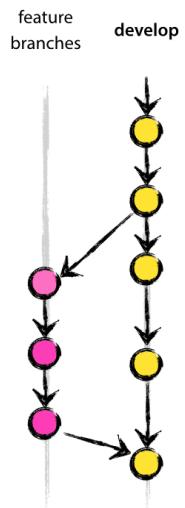
Convenciones



44

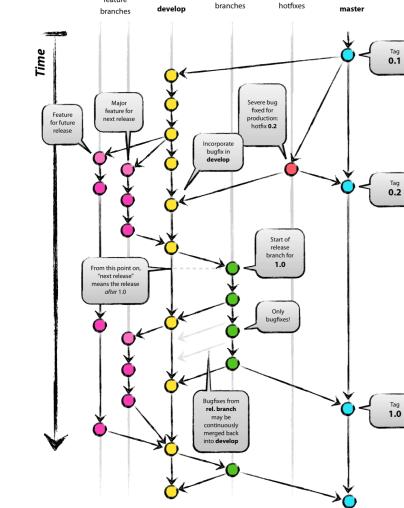
## Nuevas features

Convenciones



45

## Todo junto...



46

## Consejos

- Hacer **commits pequeños y puntuales**, con la mayor frecuencia posible.
- Mantener actualizada la copia local del repositorio, para estar sincronizados con el resto del equipo.
- Commitear los **archivos fuente**, nunca los archivos derivados!
- Manejar inmediatamente los **conflictos**.

47

## Un ejemplo

En el repositorio está toda la historia de lo que pasó con cada línea de código...



48

## Links útiles

### ► Repos hosts

- ▶ Bitbucket: <https://bitbucket.org>
- ▶ GitHub: <https://github.com>
- ▶ GitLab: <https://gitlab.com>
- ▶ GitLab Exactas: <https://git.exactas.uba.ar>

### ► Bibliografía

- ▶ **Git - la guía sencilla:**  
<http://rogerdudler.github.io/git-guide/index.es.html>
- ▶ **Pro Git book:**  
<https://git-scm.com/book/en/v2>
- ▶ **Try git:**  
<https://try.github.io>

49

## ¿Preguntas?

50

# Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Departamento de Computación - FCEyN - UBA

Lógica proposicional

1

## Habíamos visto...

**Objetivo:** Aprender a programar en lenguajes funcionales y en lenguajes imperativos.

- ▶ Especificar problemas.
  - ▶ Describirlos en un lenguaje semiformal.
- ▶ Pensar algoritmos para resolver los problemas.
  - ▶ En esta materia nos concentramos en programas para tratamiento de secuencias principalmente.
- ▶ Empezar a razonar acerca de estos algoritmos y programas.
  - ▶ Veremos conceptos de testing.

2

## Definición (Especificación) de un problema

```
problema nombre(parámetros) : tipo de dato del resultado{  
    requiere etiqueta { condiciones sobre los parámetros de entrada }  
    asegura etiqueta { condiciones sobre los parámetros de salida }  
}
```

- ▶ *nombre*: nombre que le damos al problema
  - ▶ será resuelto por una función con ese mismo nombre
- ▶ *parámetros*: lista de parámetros separada por comas, donde cada parámetro contiene:
  - ▶ Nombre del parámetro
  - ▶ Tipo de datos del parámetro
- ▶ *tipo de dato del resultado*: tipo de dato del resultado del problema (initialmente especificaremos funciones)
  - ▶ En los asegura, podremos referenciar el valor devuelto con el nombre de *res*
- ▶ *etiquetas*: son nombres **opcionales** que nos servirán para nombrar declarativamente a las condiciones de los requiere o aseguras.

3

## Definición (Especificación) de un problema

### ▶ Sobre los requiere

- ▶ Describen todas las condiciones y posibles valores o casuísticas de los parámetros de entrada.
- ▶ Puede haber más de un requiere (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
- ▶ Evitar contradicciones (un requiere no debería contradecir a otro).

### ▶ Sobre los asegura

- ▶ Describen todas las condiciones y posibles valores o casuísticas de los parámetros de salida y entrada/salida en función de los parámetros de entrada.
- ▶ Puede haber más de un asegura (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
- ▶ Evitar contradicciones (un asegura no debería contradecir a otro).

4

## Antes de continuar... hablemos de lógica proposicional

- ▶ Si bien no utilizaremos un lenguaje formal para especificar... ¿Es lo mismo decir...?
  - ▶ Mañana llueve e iré a comprar un paraguas
  - ▶ Si mañana llueve iré a comprar un paraguas
  - ▶ O mañana no llueve o no iré a comprar un paraguas
  - ▶ Comprará un paraguas por si mañana llueve
  - ▶ Si compro un paraguas, mañana llueve

5

## El abogado del diablo



- ▶ ¿Inocente o culpable?
  - ▶ Su torso está desnudo... pero... ¿y sus pies?
  - ▶ ¿Realmente estaba en el pasillo y en el ascensor al mismo tiempo?

6

## Lógica proposicional

- ▶ Es la lógica que habla sobre las proposiciones.
- ▶ Son oraciones que tienen un valor de verdad, Verdadero o Falso (aunque vamos a usar una variación).
- ▶ Sirve para poder deducir el valor de verdad de una proposición, a partir de conocer el valor de otras.

7

## Lógica proposicional - Sintaxis

- ▶ Símbolos:  
True , False ,  $\neg$  ,  $\wedge$  ,  $\vee$  ,  $\rightarrow$  ,  $\leftrightarrow$  , ( , )
- ▶ Variables proposicionales (infinitas)  
 $p$  ,  $q$  ,  $r$  , ...
- ▶ Fórmulas
  1. True y False son fórmulas
  2. Cualquier variable proposicional es una fórmula
  3. Si  $A$  es una fórmula,  $\neg A$  es una fórmula
  4. Si  $A_1, A_2, \dots, A_n$  son fórmulas,  $(A_1 \wedge A_2 \wedge \dots \wedge A_n)$  es una fórmula
  5. Si  $A_1, A_2, \dots, A_n$  son fórmulas,  $(A_1 \vee A_2 \vee \dots \vee A_n)$  es una fórmula
  6. Si  $A$  y  $B$  son fórmulas,  $(A \rightarrow B)$  es una fórmula
  7. Si  $A$  y  $B$  son fórmulas,  $(A \leftrightarrow B)$  es una fórmula

8

## Ejemplos

¿Cuáles son fórmulas?

- $p \vee q$       no
- $(p \vee q)$       sí
- $p \vee q \rightarrow r$       no
- $(p \vee q) \rightarrow r$       no
- $((p \vee q) \rightarrow r)$       sí
- $(p \rightarrow q \rightarrow r)$       no

9

## Semántica clásica

- Dos valores de verdad: "verdadero" (V) y "falso" (F).

- Interpretación:

- True siempre vale V.
- False siempre vale F.
- $\neg$  se interpreta como "no", se llama **negación**.
- $\wedge$  se interpreta como "y", se llama **conjunción**.
- $\vee$  se interpreta como "o"(no exclusivo), se llama **disyunción**.
- $\rightarrow$  se interpreta como "si... entonces", se llama **implicación**.
- $\leftrightarrow$  se interpreta como "si y solo si", se llama **doble implicación o equivalencia**.

10

## Semántica clásica: tablas de verdad

Conociendo el valor de las variables proposicionales de una fórmula, podemos calcular el valor de verdad de la fórmula.

$p$	$\neg p$
V	F
F	V

$p$	$q$	$(p \wedge q)$
V	V	V
V	F	F
F	V	F
F	F	F

$p$	$q$	$(p \vee q)$
V	V	V
V	F	V
F	V	V
F	F	F

$p$	$q$	$(p \rightarrow q)$
V	V	V
V	F	F
F	V	V
F	F	V

$p$	$q$	$(p \leftrightarrow q)$
V	V	V
V	F	F
F	V	F
F	F	V

11

## Ejemplo: tabla de verdad para $((p \wedge q) \rightarrow r)$

$p$	$q$	$r$	$(p \wedge q)$	$((p \wedge q) \rightarrow r)$
1	1	1	1	1
1	1	0	1	0
1	0	1	0	1
1	0	0	0	1
0	1	1	0	1
0	1	0	0	1
0	0	1	0	1
0	0	0	0	1

12

## Tautologías, contradicciones y contingencias

- Una fórmula es una **tautología** si siempre toma el valor *V* para valores definidos de sus variables proposicionales.

Por ejemplo,  $((p \wedge q) \rightarrow p)$  es tautología:

<i>p</i>	<i>q</i>	$(p \wedge q)$	$((p \wedge q) \rightarrow p)$
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	V

- Una fórmula es una **contradicción** si siempre toma el valor *F* para valores definidos de sus variables proposicionales.

Por ejemplo,  $(p \wedge \neg p)$  es contradicción:

<i>p</i>	$\neg p$	$(p \wedge \neg p)$
V	F	F
F	V	F

- Una fórmula es una **contingencia** cuando no es ni tautología ni contradicción.

13

## Equivalencias entre fórmulas

- Dos fórmulas *A* y *B* son **equivalentes** (y se escribe  $A \equiv B$ ) si y sólo si,  $A \leftrightarrow B$  es una tautología.

- **Teorema.** Las siguientes fórmulas son tautologías.

1. Doble negación  
 $(\neg \neg p \leftrightarrow p)$   
 $((p \wedge q) \leftrightarrow (q \wedge p))$   
 $((p \vee q) \leftrightarrow (q \vee p))$
  2. Idempotencia  
 $((p \wedge p) \leftrightarrow p)$   
 $((p \vee p) \leftrightarrow p)$   
 $((p \wedge (q \vee r)) \leftrightarrow ((p \wedge q) \vee (p \wedge r)))$   
 $((p \vee (q \wedge r)) \leftrightarrow ((p \vee q) \wedge (p \vee r)))$
  3. Asociatividad  
 $((((p \wedge q) \wedge r) \leftrightarrow (p \wedge (q \wedge r)))$   
 $((((p \vee q) \vee r) \leftrightarrow (p \vee (q \vee r)))$
  4. Comutatividad
5. Distributividad  
 $((p \wedge (q \vee r)) \leftrightarrow ((p \wedge q) \vee (p \wedge r)))$   
 $((p \vee (q \wedge r)) \leftrightarrow ((p \vee q) \wedge (p \vee r)))$
  6. Reglas de De Morgan  
 $(\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q))$   
 $(\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q))$

14

## Relación de fuerza

- Decimos que *A* es más fuerte que *B* cuando  $(A \rightarrow B)$  es tautología.

- También decimos que *A* fuerza a *B* o que *B* es más débil que *A*.

- Por ejemplo,

1. ¿ $(p \wedge q)$  es más fuerte que *p*? Sí
2. ¿ $(p \vee q)$  es más fuerte que *p*? No
3. ¿*p* es más fuerte que  $(q \rightarrow p)$ ? Sí  
Pero notemos que si *q* está indefinido y *p* es verdadero entonces  $(q \rightarrow p)$  está indefinido.
4. ¿*p* es más fuerte que *q*? No
5. ¿*p* es más fuerte que  $p$ ? Sí
6. ¿hay una fórmula más fuerte que todas? Sí, False
7. ¿hay una fórmula más débil que todas? Sí, True

15

## Expresión bien definida

- Toda expresión está **bien definida** si todas las proposiciones valen *T* o *F*.

- Sin embargo, existe la posibilidad de que haya expresiones que no estén bien definidas.

- Por ejemplo, la expresión  $x/y = 5$  no está bien definida si  $y = 0$ .

- Por esta razón, necesitamos una lógica que nos permita decir que está bien definida la siguiente expresión

$$y = 0 \vee x/y = 5$$

- Para esto, introducimos **tres** valores de verdad:

1. verdadero (*V*)
2. falso (*F*)
3. indefinido ( $\perp$ )

16

## Semántica trivaluada (secuencial)

Se llama **secuencial** porque ...

- ▶ los términos se evalúan de izquierda a derecha,
- ▶ la evaluación termina cuando se puede deducir el valor de verdad, aunque el resto esté indefinido.

Introducimos los operadores lógicos  $\wedge_L$  (y-luego, o *conditional and*, o **cand**),  $\vee_L$  (o-luego o *conditional or*, o **cor**).

$p$	$q$	$(p \wedge_L q)$
V	V	V
V	F	F
F	V	F
F	F	F
V	$\perp$	$\perp$
F	$\perp$	F
$\perp$	V	$\perp$
$\perp$	F	$\perp$
$\perp$	$\perp$	$\perp$

$p$	$q$	$(p \vee_L q)$
V	V	V
V	F	V
F	V	V
F	F	F
V	$\perp$	V
F	$\perp$	$\perp$
$\perp$	V	$\perp$
$\perp$	F	$\perp$
$\perp$	$\perp$	$\perp$

17

## Semántica trivaluada (secuencial)

¿Cuál es la tabla de verdad de  $\rightarrow_L$ ?

$p$	$q$	$(p \rightarrow_L q)$
V	V	V
V	F	F
F	V	V
F	F	V
V	$\perp$	$\perp$
F	$\perp$	V
$\perp$	V	$\perp$
$\perp$	F	$\perp$
$\perp$	$\perp$	$\perp$

18

## Entonces...

### Lógica proposicional y lógica trivaluada

- ▶ **Convención:** Dado que nuestros tipos de datos siempre tendrán como valor posible el indefinido o  $\perp$ , en general, asumiremos que estamos utilizando la lógica **trivaluada** por default.
- ▶ Es decir, salvo en los casos dónde se indique lo contrario:
  - ▶  $\wedge$  podrá ser interpretado como  $\wedge_L$  directamente
  - ▶ y así con todos los operadores vistos.

19

## Entonces... hablando de lógica proposicional

- ▶ ¿Es lo mismo decir...?
  - ▶ Mañana llueve e iré a comprar un paraguas
  - ▶ Si mañana llueve iré a comprar un paraguas
  - ▶ O mañana no llueve o no iré a comprar un paraguas
  - ▶ Compraré un paraguas por si mañana llueve
  - ▶ Si compro un paraguas, mañana llueve

20

## Entonces... hablando de lógica proposicional

- Si llamamos:
  - $a = \text{Mañana llueve}$
  - $b = \text{Iré a comprar un paraguas}$
- Mañana llueve e iré a comprar un paraguas  
Lo podríamos modelar como:  $a \wedge b$
- Si mañana llueve iré a comprar un paraguas  
Lo podríamos modelar como:  $a \rightarrow b$
- O mañana no llueve o no iré a comprar un paraguas  
Lo podríamos modelar como:  $\neg a \vee \neg b$
- Comprará un paraguas por si mañana llueve
  - ¡A veces es difícil desambiguar!
  - Por si mañana llueve es una nueva proposición
- Si compro un paraguas, mañana llueve  
Lo podríamos modelar como:  $b \rightarrow a$

21

## Práctica 1: Ejercicio 4

Determinar el valor de verdad de las siguientes proposiciones:

- a)  $(\neg a \vee b)$
- b)  $(c \vee (y \wedge x)) \vee b)$

cuando el valor de verdad de  $a, b$  y  $c$  es *verdadero*, mientras que el de  $x$  e  $y$  es *falso*.

## Práctica 1: Ejercicio 5

Determinar, utilizando tablas de verdad, si las siguientes fórmulas son tautologías, contradicciones o contingencias.

- b)  $(p \wedge \neg p)$
- d)  $((p \vee q) \rightarrow p)$
- i)  $((p \wedge (q \vee r)) \leftrightarrow ((p \wedge q) \vee (p \wedge r)))$

23

## Práctica 1: Ejercicio 6

Determinar la relación de fuerza de los siguientes pares de fórmulas:

1. True, False

$$\alpha = (p \wedge q)$$
$$\beta = (p \vee q)$$

2.  $(p \wedge q), (p \vee q)$

$p$	$q$	$\alpha$	$\beta$	$\alpha \rightarrow \beta$	$\beta \rightarrow \alpha$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	1	1

3. True, True

$p$	$q$	$\alpha$	$\alpha \rightarrow p$	$p \rightarrow \alpha$
0	0	0	1	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	1

4.  $p, (p \wedge q)$

7.  $p, q$  Ninguna es más fuerte

24

## Práctica 1: Ejercicio 7

Usando reglas de equivalencia (comutatividad, asociatividad, De Morgan, etc) determinar si los siguientes pares de fórmulas son equivalencias. Indicar en cada paso qué regla se utilizó.

2. ►  $(p \vee q) \wedge (p \vee r)$   
►  $(\neg p \rightarrow (q \wedge r))$

$$\begin{aligned} & (\neg p \rightarrow (q \wedge r)) \\ & \quad \downarrow \text{Reemplazo implicación} \\ & (p \vee (q \wedge r)) \\ & \quad \downarrow \text{Distributiva} \\ & ((p \vee q) \wedge (p \vee r)) \end{aligned}$$

25

## Práctica 1: Ejercicio 7

Usando reglas de equivalencia (comutatividad, asociatividad, De Morgan, etc) determinar si los siguientes pares de fórmulas son equivalencias. Indicar en cada paso qué regla se utilizó.

6. ►  $\neg(p \wedge (q \wedge s))$   
►  $(s \rightarrow (\neg p \vee \neg q))$

$$\begin{aligned} & \neg(p \wedge (q \wedge s)) \\ & \quad \downarrow \text{De Morgan} \\ & (\neg p \vee \neg(q \wedge s)) \\ & \quad \downarrow \text{De Morgan} \\ & (\neg p \vee \neg q \vee \neg s) \\ & \quad \downarrow \text{Comutativa} \\ & (\neg s \vee \neg p \vee \neg q) \\ & \quad \downarrow \text{Reemplazo implicación} \\ & (s \rightarrow (\neg p \vee \neg q)) \end{aligned}$$

26

## Práctica 1: Ejercicio 12

Sean las variables proposicionales  $f$ ,  $e$  y  $m$  con los siguientes significados:

- $f \equiv$  “es fin de semana”
- $e \equiv$  “Juan estudia”
- $m \equiv$  “Juan escucha música”

Escribir usando lógica proposicional las siguientes oraciones:

1. “Si es fin de semana, Juan estudia o escucha música, pero no ambas cosas”  $f \rightarrow ((e \vee m) \wedge \neg(e \wedge m))$
2. “Si no es fin de semana entonces Juan no estudia”  $\neg f \rightarrow \neg e$
3. “Cuando Juan estudia los fines de semana, lo hace escuchando música”  $(f \wedge e) \rightarrow m$

27

## Práctica 1: Ejercicio 19

Determinar los valores de verdad de las siguientes proposiciones cuando el valor de verdad de  $b$  y  $c$  es *verdadero*, el de  $a$  es *falso* y el de  $x$  e  $y$  es *indefinido*:

- a)  $(\neg x \vee_L b)$
- c)  $\neg(c \vee y)$
- g)  $(\neg c \wedge_L \neg y)$

28

## Práctica 1: Ejercicio 20

Determinar los valores de las siguientes fórmulas de Lógica Ternaria cuando el valor de verdad de  $p$  es *verdadero*, el de  $q$  es *falso* y el de  $r$  es *indefinido*:

- a)  $((9 \leq 9) \wedge p)$
- d)  $((3 > 9) \vee (r \wedge (q \wedge p)))$
- i)  $(p \wedge ((5 - 7 + 3 = 0)) \leftrightarrow (2^2 - 1 > 3)))$

29

Presentemos nuestro lenguaje de especificación

31

## Práctica 1: Ejercicio 21

Sean  $p$ ,  $q$  y  $r$  tres variables de las que se sabe que:

- $p$  y  $q$  nunca están indefinidas,
- $r$  se indefine si  $q$  es verdadera

Proponer, para cada ítem, una fórmula que nunca se indefina, utilizando siempre las tres variables. Cada fórmula debe ser verdadera si y solo si se cumple que:

- b) Ninguna es verdadera.
- d) Sólo  $p$  y  $q$  son verdaderas.

30

## Problemas y Especificaciones

Inicialmente los problemas resolveremos con una computadora serán planteados como funciones. Es decir:

- Dados ciertos datos de entrada, obtendremos un resultado
- Más adelante en la materia, extenderemos el tipo de problemas que podemos resolver...

32

## Definición (Especificación) de un problema

```
problema nombre(parámetros) : tipo de dato del resultado {  
    requiere etiqueta: { condiciones sobre los parámetros de entrada }  
    asegura etiqueta: { condiciones sobre los parámetros de salida }  
}
```

- ▶ *nombre*: nombre que le damos al problema
  - ▶ será resuelto por una función con ese mismo nombre
- ▶ *parámetros*: lista de parámetros separada por comas, donde cada parámetro contiene:
  - ▶ Nombre del parámetro
  - ▶ Tipo de datos del parámetro
- ▶ *tipo de dato del resultado*: tipo de dato del resultado del problema (initialmente especificaremos funciones)
  - ▶ En los asegura, podremos referenciar el valor devuelto con el nombre de *res*
- ▶ *etiquetas*: son nombres *opcionales* que nos servirán para nombrar declarativamente a las condiciones de los requiere o aseguras.

33

## Definición (Especificación) de un problema

### ► Sobre los requiere

- ▶ Describen todas las condiciones y posibles valores o casuísticas de los parámetros de entrada.
- ▶ Puede haber más de un requiere (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
- ▶ Evitar contradicciones (un requiere no debería contradecir a otro).

### ► Sobre los asegura

- ▶ Describen todas las condiciones y posibles valores o casuísticas de los parámetros de salida y entrada/salida en función de los parámetros de entrada.
- ▶ Puede haber más de un asegura (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
- ▶ Evitar contradicciones (un asegura no debería contradecir a otro).

34

## ¿Cómo contradicciones?

```
problema soyContradicorio(x :  $\mathbb{Z}$ ) :  $\mathbb{Z}$ {  
    requiere esMayor: { $x > 0$ }  
    requiere esMenor: { $x < 0$ }  
    asegura esElSiguiente: { $res + 1 = x$ }  
    asegura esElAnterior: { $res - 1 = x$ }  
}
```

35

## Ejemplos

```
problema raizCuadrada(x :  $\mathbb{R}$ ) :  $\mathbb{R}$ {  
    requiere: { $x \geq 0$ }  
    asegura: { $res * res = x \wedge res \geq 0$ }  
}
```

```
problema sumar(x :  $\mathbb{Z}$ , y :  $\mathbb{Z}$ ) :  $\mathbb{Z}$ {  
    requiere: {True}  
    asegura: { $res = x + y$ }  
}
```

```
problema restar(x :  $\mathbb{Z}$ , y :  $\mathbb{Z}$ ) :  $\mathbb{Z}$ {  
    requiere: {True}  
    asegura: { $res = x - y$ }  
}
```

```
problema cualquieramayor(x :  $\mathbb{Z}$ ) :  $\mathbb{Z}$ {  
    requiere: {True}  
    asegura: { $res > x$ }  
}
```

36

## ¿Por qué nuestro lenguaje será semiformal?: Ejemplos

```
problema raizCuadrada(x : ℝ) : ℝ {  
    requiere: {x debe ser mayor o igual que 0}  
    asegura: {res debe ser mayor o igual que 0}  
    asegura: {res elevado al cuadrado será x}  
}
```

```
problema sumar(x : ℤ, y : ℤ) : ℤ {  
    requiere: {-}  
    asegura: {res es la suma de x e y}  
}
```

```
problema restar(x : ℤ, y : ℤ) : ℤ {  
    requiere: {Siempre cumplen}  
    asegura: {res es la resta de x menos y}  
}
```

```
problema cualquiermayor(x : ℤ) : ℤ {  
    requiere: {Vale para cualquier valor posible de x}  
    asegura: {res debe tener cualquier valor mayor a x}  
}
```

37

## El contrato

- ▶ **Contrato:** *El programador escribe un programa P tal que si el usuario suministra datos que hacen verdadera la precondición, entonces P termina en una cantidad finita de pasos retornando un valor que hace verdadera la postcondición.*
- ▶ El programa P es **correcto** para la especificación dada por la precondición y la postcondición exactamente cuando se cumple el contrato.
- ▶ Si el usuario no cumple la precondición y P se cuelga o no cumple la poscondición...
  - ▶ ¿El usuario tiene derecho a quejarse?
  - ▶ ¿Se cumple el contrato?
- ▶ Si el usuario cumple la precondición y P se cuelga o no cumple la poscondición...
  - ▶ ¿El usuario tiene derecho a quejarse?
  - ▶ ¿Se cumple el contrato?

38

## Interpretando una especificación

- ▶ problema raizCuadrada(x : ℝ) : ℝ {  
 requiere: {x debe ser mayor o igual que 0}  
 asegura: {res debe ser mayor o igual que 0}  
 asegura: {res elevado al cuadrado será x}  
}
- ▶ ¿Qué significa esta especificación?
- ▶ Se especifica que si el programa raizCuadrada se comienza a ejecutar en un estado que cumple  $x \geq 0$ , entonces el programa **termina** y el estado final cumple  $res * res = x$  y  $res \geq 0$ .

39

## Otro ejemplo

Dados dos enteros **dividendo** y **divisor**, obtener el cociente entero entre ellos.

```
problema cociente(dividendo : ℤ, divisor : ℤ) : ℤ {  
    requiere: {divisor > 0}  
    asegura: {res * divisor ≤ dividendo}  
    asegura: {(res + 1) * divisor > dividendo}  
}
```

Qué sucede si ejecutamos con ...

- ▶ dividendo = 1 y divisor = 0?
- ▶ dividendo = -4 y divisor = -2, y obtenemos res = 2?
- ▶ dividendo = -4 y divisor = -2, y obtenemos res = 0?
- ▶ dividendo = 4 y divisor = -2, y el programa no termina?

40

## Tipos de datos

- ▶ Un **tipo de datos** es un **conjunto de valores** (el conjunto base del tipo) provisto de una serie de **operaciones** que involucran a esos valores.
- ▶ Para hablar de un elemento de un tipo  $T$  en nuestro lenguaje, escribimos un **término** o **expresión**
  - ▶ Variable de tipo  $T$  (ejemplos:  $x, y, z$ , etc)
  - ▶ Constante de tipo  $T$  (ejemplos:  $1, -1, \frac{1}{5}, 'a'$ , etc)
  - ▶ Función (operación) aplicada a otros términos (del tipo  $T$  o de otro tipo)
- ▶ Todos los tipos tienen un elemento distinguido:  $\perp$  o **Indef**

41

## Tipos de datos de nuestro lenguaje de especificación

- ▶ Básicos
  - ▶ Enteros ( $\mathbb{Z}$ )
  - ▶ Reales ( $\mathbb{R}$ )
  - ▶ Booleanos (Bool)
  - ▶ Caracteres (Char)
- ▶ Enumerados
- ▶ Uplas
- ▶ Secuencias

42

## Tipo $\mathbb{Z}$ (números enteros)

- ▶ Su **conjunto base** son los números enteros.
- ▶ Constantes:  $0 ; 1 ; -1 ; 2 ; -2 ; \dots$
- ▶ Operaciones aritméticas:
  - ▶  $a + b$  (suma);  $a - b$  (resta);  $\text{abs}(a)$  (valor absoluto)
  - ▶  $a * b$  (multiplicación);  $a \text{ div } b$  (división entera);
  - ▶  $a \text{ mod } b$  (resto de dividir  $a$  por  $b$ ),  $a^b$  o  $\text{pot}(a,b)$  (potencia)
  - ▶  $a / b$  (división, da un valor de  $\mathbb{R}$ )
- ▶ Fórmulas que comparan términos de tipo  $\mathbb{Z}$ :
  - ▶  $a < b$  (menor)
  - ▶  $a \leq b$  o  $a \leq b$  (menor o igual)
  - ▶  $a > b$  (mayor)
  - ▶  $a \geq b$  o  $a \geq b$  (mayor o igual)
  - ▶  $a = b$  (iguales)
  - ▶  $a \neq b$  (distintos)

43

## Tipo $\mathbb{R}$ (números reales)

- ▶ Su conjunto base son los números reales.
- ▶ Constantes:  $0 ; 1 ; -7 ; 81 ; 7,4552 ; \pi \dots$
- ▶ Operaciones aritméticas:
  - ▶ Suma, resta y producto (pero no div y mod)
  - ▶  $a/b$  (división)
  - ▶  $\log_b(a)$  (logaritmo)
  - ▶ Funciones trigonométricas
- ▶ Fórmulas que comparan términos de tipo  $\mathbb{R}$ :
  - ▶  $a < b$  (menor)
  - ▶  $a \leq b$  o  $a \leq b$  (menor o igual)
  - ▶  $a > b$  (mayor)
  - ▶  $a \geq b$  o  $a \geq b$  (mayor o igual)
  - ▶  $a = b$  (iguales)
  - ▶  $a \neq b$  (distintos)

44

## Tipo Bool (valor de verdad)

- ▶ Su conjunto base es  $\mathbb{B} = \{\text{true}, \text{false}\}$ .
- ▶ Conectivos lógicos:  $!$ ,  $\&\&$ ,  $\|$ , con la semántica bi-valuada estándar.
- ▶ Fórmulas que comparan términos de tipo Bool:
  - ▶  $a = b$
  - ▶  $a \neq b$  (se puede escribir  $a \neq b$ )

45

## Tipo Char (caracteres)

- ▶ Sus elementos son las letras, dígitos y símbolos.
- ▶ Constantes:  $'a'$ ,  $'b'$ ,  $'c'$ , ...,  $'z'$ , ...,  $'A'$ ,  $'B'$ ,  $'C'$ , ...,  $'Z'$ , ...,  $'0'$ ,  $'1'$ ,  $'2'$ , ...,  $'9'$  (en el orden dado por el estándar ASCII).
- ▶ Función  $\text{ord}$ , que numera los caracteres, con las siguientes propiedades:
  - ▶  $\text{ord}'a' + 1 = \text{ord}'b'$
  - ▶  $\text{ord}'A' + 1 = \text{ord}'B'$
  - ▶  $\text{ord}'1' + 1 = \text{ord}'2'$
- ▶ Función  $\text{char}$ , de modo tal que si  $c$  es cualquier char entonces  $\text{char}(\text{ord}(c)) = c$ .
- ▶ Las comparaciones entre caracteres son comparaciones entre sus órdenes, de modo tal que  $a < b$  es equivalente a  $\text{ord}(a) < \text{ord}(b)$ .

46

## Tipos enumerados

- ▶ Cantidad finita de elementos.  
Cada uno, denotado por una constante.  

```
enum Nombre { constantes }
```
- ▶ *Nombre* (del tipo): tiene que ser nuevo.
- ▶ *Constantes*: nombres nuevos separados por comas.
- ▶ Convención: todos en mayúsculas.
- ▶  $\text{ord}(a)$  da la posición del elemento en la definición (empezando de 0).
- ▶ Inversa: se usa el nombre del tipo funciona como inversa de  $\text{ord}$ .

47

## Ejemplo de tipo enumerado

Definimos el tipo Día así:

```
enum Día {  
    LUN, MAR, MIER, JUE, VIE, SAB, DOM  
}
```

Valen:

- ▶  $\text{ord}(\text{LUN}) = 0$
- ▶  $\text{Día}(2) = \text{MIE}$
- ▶  $\text{JUE} < \text{VIE}$

48

# Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Departamento de Computación - FCEyN - UBA

Introducción a la especificación de problemas

1

## Retomando: Tipos de datos

- ▶ Un **tipo de datos** es un **conjunto de valores** (el conjunto base del tipo) provisto de una serie de **operaciones** que involucran a esos valores.
- ▶ Para hablar de un elemento de un tipo  $T$  en nuestro lenguaje, escribimos un **término** o **expresión**
  - ▶ Variable de tipo  $T$  (ejemplos:  $x, y, z$ , etc)
  - ▶ Constante de tipo  $T$  (ejemplos:  $1, -1, \frac{1}{5}, 'a'$ , etc)
  - ▶ Función (operación) aplicada a otros términos (del tipo  $T$  o de otro tipo)
- ▶ Todos los tipos tienen un elemento distinguido:  $\perp$  o **Indef**

3

## Definición (Especificación) de un problema

- ```
problema nombre(parámetros) : tipo de dato del resultado {  
    requiere etiqueta: { condiciones sobre los parámetros de entrada }  
    asegura etiqueta: { condiciones sobre los parámetros de salida }  
}
```
- ▶ **nombre**: nombre que le damos al problema
    - ▶ será resuelto por una función con ese mismo nombre
  - ▶ **parámetros**: lista de parámetros separada por comas, donde cada parámetro contiene:
    - ▶ Nombre del parámetro
    - ▶ Tipo de datos del parámetro
  - ▶ **tipo de dato del resultado**: tipo de dato del resultado del problema (initialmente especificaremos funciones)
    - ▶ En los asegura, podremos referenciar el valor devuelto con el nombre de **res**
  - ▶ **etiquetas**: son nombres **opcionales** que nos servirán para nombrar declarativamente a las condiciones de los requiere o aseguras.

2

## Tipos de datos de nuestro lenguaje de especificación

- ▶ Básicos
  - ▶ Enteros ( $\mathbb{Z}$ )
  - ▶ Reales ( $\mathbb{R}$ )
  - ▶ Booleanos (Bool)
  - ▶ Caracteres (Char)
- ▶ Enumerados
- ▶ Uplas
- ▶ Secuencias

4

## Tipo upla (o tupla)

- ▶ Una estructura de datos es una forma particular de organizar la información.
- ▶ Uplas, de dos o más elementos, cada uno de cualquier tipo.
- ▶  $T_0 \times T_1 \times \dots \times T_k$ : Tipo de las  $k$ -uplas de elementos de tipos  $T_0, T_1, \dots, T_k$ , respectivamente, donde  $k$  es fijo.
- ▶ Ejemplos:
  - ▶  $\mathbb{Z} \times \mathbb{Z}$  son los pares ordenados de enteros.
  - ▶  $\mathbb{Z} \times \text{Char} \times \text{Bool}$  son las triples ordenadas con un entero, luego un carácter y luego un valor booleano.
- ▶  $n$ ésimo:  $(a_0, \dots, a_k)_m$  es el valor  $a_m$  en caso de que  $0 \leq m \leq k$ . Si no, está indefinido.
- ▶ Ejemplos:
  - ▶  $(7, 5)_0 = 7$
  - ▶  $('a', \text{DOM}, 78)_2 = 78$

5

## Secuencias

- ▶ **Secuencia:** Varios elementos del mismo tipo  $T$ , posiblemente repetidos, ubicados en un cierto orden.
- ▶  $\text{seq}\langle T \rangle$  es el tipo de las secuencias cuyos elementos son de tipo  $T$ .
- ▶  $T$  es un tipo arbitrario.
  - ▶ Hay secuencias de  $\mathbb{Z}$ , de  $\text{Bool}$ , de  $\text{Días}$ , de 5-uplas;
  - ▶ también hay secuencias de secuencias de  $T$ ;
  - ▶ etcétera.

6

## Secuencias. Notación

- ▶ Una forma de escribir un elemento de tipo  $\text{seq}\langle T \rangle$  es escribir términos de tipo  $T$  separados por comas, entre  $\langle \dots \rangle$ .
  - ▶  $\langle 1, 2, 3, 4, 1, 0 \rangle$  es una secuencia de  $\mathbb{Z}$ .
  - ▶  $\langle 1, 1 + 1, 3, 2 * 2, 5 \bmod 2, 0 \rangle$  es otra secuencia de  $\mathbb{Z}$  (igual a la anterior).
- ▶ La **secuencia vacía** se escribe  $\langle \rangle$ , cualquiera sea el tipo de los elementos de la secuencia.
- ▶ Se puede formar secuencias de elementos de cualquier tipo.
  - ▶ Como  $\text{seq}\langle \mathbb{Z} \rangle$  es un tipo, podemos armar secuencias de  $\text{seq}\langle \mathbb{Z} \rangle$  (secuencias de secuencias de  $\mathbb{Z}$ , o sea  $\text{seq}\langle \text{seq}\langle \mathbb{Z} \rangle \rangle$ ).
    - ▶  $\langle \langle 12, 13 \rangle, \langle -3, 9, 0 \rangle, \langle 5 \rangle, \langle \rangle, \langle \rangle, \langle 3 \rangle \rangle$  es un elemento de tipo  $\text{seq}\langle \text{seq}\langle \mathbb{Z} \rangle \rangle$ .

7

## Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ( $\text{seq}\langle \mathbb{Z} \rangle$ , etc...)

- ▶  $\langle 1, 2, 3, 4, 5 \rangle$ ? Bien Formada. Tipa como  $\text{seq}\langle \mathbb{Z} \rangle$  y  $\text{seq}\langle \mathbb{R} \rangle$
- ▶  $\langle 1, \text{true}, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea ( $\text{Bool}$  y  $\mathbb{Z}$ )
- ▶  $\langle 'a', 2, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea ( $\text{Char}$  y  $\mathbb{Z}$ )
- ▶  $\langle 'H', 'o', 'l', 'a' \rangle$ ? Bien Formada. Tipa como  $\text{seq}\langle \text{Char} \rangle$
- ▶  $\langle \text{true}, \text{false}, \text{true}, \text{true} \rangle$ ? Bien Formada. Tipa como  $\text{seq}\langle \text{Bool} \rangle$
- ▶  $\langle \frac{2}{5}, \pi, e \rangle$ ? Bien Formada. Tipa como  $\text{seq}\langle \mathbb{R} \rangle$
- ▶  $\langle \rangle$ ? Bien formada. Tipa como cualquier secuencia  $\text{seq}\langle X \rangle$  donde  $X$  es un tipo válido.
- ▶  $\langle \langle \rangle \rangle$ ? Bien formada. Tipa como cualquier secuencia  $\text{seq}\langle \text{seq}\langle X \rangle \rangle$  donde  $X$  es un tipo válido.

8

## Funciones sobre secuencias

### Longitud

- Longitud:  $\text{length}(a : \text{seq}\langle T \rangle) : \mathbb{Z}$ 
  - ▶ Representa la longitud de la secuencia  $a$ .
  - ▶ Notación:  $\text{length}(a)$  se puede escribir como  $|a|$  o como  $a.\text{length}$ .
- Ejemplos:
  - ▶  $|\langle \rangle| = 0$
  - ▶  $|\langle 'H', 'o', 'l', 'a' \rangle| = 4$
  - ▶  $|\langle 1, 1, 2 \rangle| = 3$

9

## Funciones con secuencias

### Pertenece

- Pertenece:  $\text{pertenece}(x : T, s : \text{seq}\langle T \rangle) : \text{Bool}$ 
  - ▶ Es **true** sí y solo sí  $x$  es elemento de  $s$ .
  - ▶ Notación:  $\text{pertenece}(x, s)$  se puede escribir como  $x \in s$ .
- Ejemplos:
  - ▶  $(1, MAR) \in \langle (1, LUN), (2, MAR), (3, JUE), (1, MAR) \rangle ? \text{true}$
  - ▶  $(1, MAR) \in \langle (1, LUN), (2, MAR), (3, JUE), (3, MAR) \rangle ? \text{false}$

11

## Funciones con secuencias

### I-ésimo elemento

- Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se define.
- Ejemplos:
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[0] = 'H'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[1] = 'o'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[2] = 'l'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[3] = 'a'$
  - ▶  $\langle 1, 1, 1, 1 \rangle[0] = 1$
  - ▶  $\langle \rangle[0] = \perp$  (Indefinido)
  - ▶  $\langle 1, 1, 1, 1 \rangle[7] = \perp$  (Indefinido)

10

## Funciones con secuencias

### Igualdad

Dos secuencias  $s_0$  y  $s_1$  (notación  $s_0 = s_1$ ) son iguales si y sólo si

- Tienen la misma cantidad de elementos
- Dada una posición, el elemento contenido en la secuencia  $s_0$  es igual al elemento contenido en la secuencia  $s_1$ .

### Ejemplos:

- $\langle 1, 2, 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle ? \text{Sí}$
- $\langle \rangle = \langle \rangle ? \text{Sí}$
- $\langle 4, 4, 4 \rangle = \langle 4, 4, 4 \rangle ? \text{Sí}$
- $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 3, 4 \rangle ? \text{No}$
- $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 4, 5, 6 \rangle ? \text{No}$
- $\langle 1, 2, 3, 5, 4 \rangle = \langle 1, 2, 3, 4, 5 \rangle ? \text{No}$

12

## Funciones con secuencias

Cabeza o Head

- ▶ Cabeza:  $head(a : seq(T)) : T$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es el primer elemento de la secuencia  $a$ .
  - ▶ Es equivalente a la expresión  $a[0]$ .
  - ▶ Si no vale  $|a| > 0$  se indefine.
- ▶ Ejemplos:
  - ▶  $head('H', 'o', 'l', 'a') = 'H'$
  - ▶  $head(\langle 1, 1, 1 \rangle) = 1$
  - ▶  $head(\langle \rangle) = \perp$  (Indefinido)

13

## Funciones con secuencias

Cola o Tail

- ▶ Cola:  $tail(a : seq(T)) : seq(T)$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es la secuencia resultante de eliminar su primer elemento.
  - ▶ Si no vale  $|a| > 0$  se indefine.
- ▶ Ejemplos:
  - ▶  $tail('H', 'o', 'l', 'a') = \langle 'o', 'l', 'a' \rangle$
  - ▶  $tail(\langle 1, 1, 1 \rangle) = \langle 1, 1, 1 \rangle$
  - ▶  $tail(\langle \rangle) = \perp$  (Indefinido)
  - ▶  $tail(\langle 6 \rangle) = \langle \rangle$

14

## Funciones con secuencias

Agregar al principio o addFirst

- ▶ Agregar cabeza:  $addFirst(t : T, a : seq(T)) : seq(T)$ 
  - ▶ Es una secuencia con los elementos de  $a$ , agregándole  $t$  como primer elemento.
  - ▶ Es una función que no se indefine
- ▶ Ejemplos:
  - ▶  $addFirst('x', 'H', 'o', 'l', 'a') = \langle 'x', 'H', 'o', 'l', 'a' \rangle$
  - ▶  $addFirst(5, \langle 1, 1, 1 \rangle) = \langle 5, 1, 1, 1, 1 \rangle$
  - ▶  $addFirst(1, \langle \rangle) = \langle 1 \rangle$

15

## Funciones con secuencias

Concatenación o concat

- ▶ Concatenación:  $concat(a : seq(T), b : seq(T)) : seq(T)$ 
  - ▶ Es una secuencia con los elementos de  $a$ , seguidos de los de  $b$ .
  - ▶ Notación:  $concat(a, b)$  se puede escribir  $a ++ b$ .
- ▶ Ejemplos:
  - ▶  $concat('H', 'o', 'l', 'a', 'H', 'o', 'l', 'a') = \langle 'H', 'o', 'l', 'a', 'H', 'o', 'l', 'a' \rangle$
  - ▶  $concat(\langle 1, 2 \rangle, \langle 3, 4 \rangle) = \langle 1, 2, 3, 4 \rangle$
  - ▶  $concat(\langle \rangle, \langle \rangle) = \langle \rangle$
  - ▶  $concat(\langle 2, 3 \rangle, \langle \rangle) = \langle 2, 3 \rangle$
  - ▶  $concat(\langle \rangle, \langle 5, 7 \rangle) = \langle 5, 7 \rangle$

16

## Funciones con secuencias

Subsecuencia o subseq

- ▶ Subsecuencia:  $\text{subseq}(a : \text{seq}\langle T \rangle, d, h : \mathbb{Z}) : \text{seq}\langle T \rangle$ 
  - ▶ Es una sublista de  $a$  en las posiciones entre  $d$  (inclusive) y  $h$  (exclusive).
  - ▶ Cuando  $0 \leq d = h \leq |a|$ , retorna la secuencia vacía.
  - ▶ Cuando no se cumple  $0 \leq d \leq h \leq |a|$ , se **define!**
- ▶ Ejemplos:
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 1) = \langle 'H' \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 4) = \langle 'H', 'o', 'l', 'a' \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 2, 2) = \langle \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, -1, 3) = \perp$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 10) = \perp$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 3, 1) = \perp$

17

## Funciones con secuencias

- ▶ Cambiar una posición:  $\text{setAt}(a : \text{seq}\langle T \rangle, i : \mathbb{Z}, val : T) : \text{seq}\langle T \rangle$ 
  - ▶ Requiere  $0 \leq i < |a|$
  - ▶ Es una secuencia igual a  $a$ , pero con valor  $val$  en la posición  $i$ .
- ▶ Ejemplos:
  - ▶  $\text{setAt}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 'X') = \langle 'X', 'o', 'l', 'a' \rangle$
  - ▶  $\text{setAt}(\langle 'H', 'o', 'l', 'a' \rangle, 3, 'A') = \langle 'H', 'o', 'l', 'A' \rangle$
  - ▶  $\text{setAt}(\langle \rangle, 0, 5) = \perp$  (Indefinido)

18

## Operaciones sobre secuencias

- ▶  $\text{length}(a : \text{seq}\langle T \rangle) : \mathbb{Z}$  (notación  $|a|$ )
- ▶  $\text{pertenece}(x : T, s : \text{seq}\langle T \rangle) : \text{Bool}$  (notación  $x \in s$ )
- ▶ indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$
- ▶ igualdad:  $\text{seq}\langle T \rangle = \text{seq}\langle T \rangle$
- ▶  $\text{head}(a : \text{seq}\langle T \rangle) : T$
- ▶  $\text{tail}(a : \text{seq}\langle T \rangle) : \text{seq}\langle T \rangle$
- ▶  $\text{addFirst}(t : T, a : \text{seq}\langle T \rangle) : \text{seq}\langle T \rangle$
- ▶  $\text{concat}(a : \text{seq}\langle T \rangle, b : \text{seq}\langle T \rangle) : \text{seq}\langle T \rangle$  (notación  $a++b$ )
- ▶  $\text{subseq}(a : \text{seq}\langle T \rangle, d, h : \mathbb{Z}) : \langle T \rangle$
- ▶  $\text{setAt}(a : \text{seq}\langle T \rangle, i : \mathbb{Z}, val : T) : \text{seq}\langle T \rangle$

19

## Definición (Especificación) de un problema

- ▶ **Sobre los requiere**
  - ▶ Describen todas las condiciones y posibles valores o casuísticas de los parámetros de entrada.
  - ▶ Puede haber más de un requiere (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
  - ▶ Evitar contradicciones (un requiere no debería contradecir a otro).
- ▶ **Sobre los asegura**
  - ▶ Describen todas las condiciones y posibles valores o casuísticas de los parámetros de salida y entrada/salida en función de los parámetros de entrada.
  - ▶ Puede haber más de un asegura (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
  - ▶ Evitar contradicciones (un asegura no debería contradecir a otro).

20

## Problemas comunes de las especificaciones

- ▶ ¿Qué sucede si específico de menos?
- ▶ ¿Qué sucede si específico de más?

21

## Sub-especificación

- ▶ Consiste en dar una **precondición más restrictiva** de lo realmente necesario, o bien una **postcondición más débil** de la que se necesita.
- ▶ Deja afuera datos de entrada o ignora condiciones necesarias para la salida (permite soluciones no deseadas).
- ▶ Ejemplo:

```
problema distinto(x : Z) : Z{  
    requiere: {x > 0}  
    asegura: {res ≠ x}  
}
```

... en vez de:

```
problema distinto(x : Z) : Z{  
    requiere: {True}  
    asegura: {res ≠ x}  
}
```

23

## Sobre-especificación

- ▶ Consiste en dar una **postcondición más restrictiva** de la que se necesita, o bien dar una **precondición más laxa**.
- ▶ Limita los posibles algoritmos que resuelven el problema, porque impone más condiciones para la salida, o amplía los datos de entrada.
- ▶ Ejemplo:  

```
problema distinto(x : Z) : Z {  
    requiere: {True}  
    asegura: {res = x + 1}  
}
```
- ▶ ... en lugar de:  

```
problema distinto(x : Z) : Z {  
    requiere: {True}  
    asegura: {res ≠ x}  
}
```

22

## Modularización

Partiendo un problema en problemas mas chicos

Dadas dos secuencias, queremos saber si uno es una permutación<sup>1</sup> de la otra secuencia:  
¿Cuándo será una secuencia permutación de la otra?

- ▶ Tienen los mismos elementos
- ▶ Cada elemento aparece la misma cantidad de veces en ambas secuencias

```
problema esPermutacion(s1, s2 : seq(T)) : Bool {  
    asegura: {res = true ⇔ para cada elemento es cierto que tiene la misma  
              cantidad de apariciones en s1 y s2 }  
}
```

Pero... falta algo...

<sup>1</sup>mismos elementos y misma cantidad por cada elemento, en un orden potencialmente distinto

24

## Modularización

Partiendo un problema en problemas mas chicos

Ahora, tenemos que especificar el problema *cantidadDeApariciones*

¿Cómo podemos saber la cantidad de apariciones de un elemento en una lista?

- ▶ Podríamos sumar 1 por cada posición donde el elemento en dicha posición es el que buscamos!
- ▶ Las operaciones de Sumatorias y Productorias también podemos usarlos

```
problema cantidadDeApariciones(s : seq(T), e : T) : Z {  
    asegura {res = la cantidad de veces que el elemento e aparece en la lista s}  
}
```

25

## Recapitulando

Partiendo un problema en problemas mas chicos

Dadas dos secuencias, queremos saber si uno es una una permutación<sup>1</sup> de la otra secuencia:

```
problema esPermutacion(s1,s2 : seq(T)) : Bool {  
    asegura: {res = true  $\leftrightarrow$  (para cada elemento e de T, se cumple que  
    (cantidadDeApariciones(s1,e) = cantidadDeApariciones(s2,e)))}  
}
```

Donde...

```
problema cantidadDeApariciones(s : seq(T), e : T) : Z {  
    asegura {res = la cantidad de veces que el elemento e aparece en la lista s}  
}
```

Y así podemos modularizar y descomponer nuestro problemas, partiendo los en problemas más chicos. Y también los podemos reutilizar!

<sup>1</sup>mismos elementos y misma cantidad por cada elemento, en un orden potencialmente distinto

26

## Modularización

O partir el problema en problemas más chicos...

Los conceptos de modularización y encapsulamiento siempre estarán relacionados con los principios de diseño de software. La estrategia se puede resumir en:

- ▶ Descomponer un problema grande en problemas más pequeños (y sencillos)
- ▶ Componerlos y obtener la solución al problema original

Esto favorece muchos aspectos de calidad como:

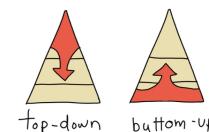
- ▶ La reutilización (una función auxiliar puede ser utilizada en muchos contextos)
- ▶ Es más fácil probar algo chico que algo grande (si cada parte cumple su función correctamente, es más probable que todas juntas también lo haga)
- ▶ La declaratividad (es más fácil entender al ojo humano)

27

## Modularización

Top Down versus Bottom Up

También es aplicable a la especificación de problemas:



```
problema esPermutacion(s1,s2 : seq(T)) : Bool {  
    asegura: {res = true  $\leftrightarrow$  (para cada elemento e de T, se cumple que  
    (cantidadDeApariciones(s1,e) = cantidadDeApariciones(s2,e)))}  
}
```

```
problema cantidadDeApariciones(s : seq(T), e : T) : Z {  
    asegura {res = la cantidad de veces que el elemento e aparece en la lista s}  
}
```

¿Lo encaramos Top Down o Bottom Up?

28

## Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¡Piedra, Papel y Tijera?... ¿Cómo podríamos especificarlo?

- ▶ ¿Cómo lo podemos modelar?
- ▶ ¿Qué tipo de datos podríamos utilizar?

Opciones:

- ▶ Números?
- ▶ Letras?
- ▶ Un enumerado?

29

## Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¡Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}
```

- ▶ ¿Qué problemas tenemos que resolver?

## Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¡Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}
```

- ▶ ¿Qué problemas tenemos que resolver?
  - ▶ En cada jugada... cada jugador elige jugar con: Piedra, Papel o Tijera
    - ▶ Si ambos jugadores eligen lo mismo, empatan
    - ▶ Piedra le gana a la Tijera
    - ▶ Tijera le gana al Papel
    - ▶ Papel le gana a la Piedra
  - ▶ Una partida es al mejor de 3 jugadas...

31

## Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¡Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}
```

```
problema determinarGanadorJugada(j1 : PPT, j2 : PPT) : Z {  
    requiere: {...}  
    asegura: {...}  
}
```

30

32

## Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¡Piedra Papel y Tijera!... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}  
  
problema determinarGanadorJugada(j1 : PPT, j2 : PPT) : Z {  
    asegura: {res = 1  $\leftrightarrow$  esGanador(j1, j2)}  
    asegura: {res = 2  $\leftrightarrow$  esGanador(j2, j1)}  
    asegura: {res = 0  $\leftrightarrow$  j1 es igual j2}  
}  
  
problema esGanador(j1 : PPT, j2 : PPT) : Bool {  
    asegura: {res = true  $\leftrightarrow$  ((j1 = PIEDRA  $\wedge$  j2 = TIJERA)  $\vee$  (j1 =  
    TIJERA  $\wedge$  j2 = PAPEL)  $\vee$  (j1 = PAPEL  $\wedge$  j2 = PIEDRA))}  
}
```

33

## Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Ya tenemos una jugada... ¿cómo sería una partida?

```
problema determinarGanadorPartida(jugadas : seq((PPTxPPT))) : Z {  
    requiere: {|jugadas| = 3}  
    asegura: {res = 1  $\leftrightarrow$  cantidadJugadasGanadas1(jugadas) >  
    cantidadJugadasGanadas2(jugadas)}  
    asegura: {res = 2  $\leftrightarrow$  cantidadJugadasGanadas1(jugadas) <  
    cantidadJugadasGanadas2(jugadas)}  
    asegura: {res = 0  $\leftrightarrow$  cantidadJugadasGanadas1(jugadas) =  
    cantidadJugadasGanadas2(jugadas)}  
}  
  
problema cantidadJugadasGanadas1(jugadas : seq((PPTxPPT))) : Z {  
    asegura: {res es la cantidad de jugadas j de la lista jugadas tales que  
    determinarGanadorJugada(j0, j1) = 1}  
}  
  
problema cantidadJugadasGanadas2(jugadas : seq((PPTxPPT))) : Z {  
    asegura: {res es la cantidad de jugadas j de la lista jugadas tales que  
    determinarGanadorJugada(j0, j1) = 2}  
}
```

34

## Siguiente paso: Algoritmos y programas

- ▶ Hasta ahora estudiamos lógica y aprendimos a **especificar** problemas
- ▶ El objetivo es ahora escribir un **algoritmo** que cumpla esa especificación
  - ▶ Secuencia de pasos que pueden llevarse a cabo mecánicamente
- ▶ Puede haber varios algoritmos que cumplan una misma especificación
- ▶ Una vez que se tiene el algoritmo, se escribe el **programa** que implementa el algoritmo
  - ▶ Expresión formal de un algoritmo
  - ▶ Lenguajes de programación
    - ▶ sintaxis definida
    - ▶ semántica definida
    - ▶ qué hace una computadora cuando recibe ese programa
    - ▶ qué especificaciones cumple
    - ▶ ejemplos: Haskell, C, C++, C#, Python, Java, Smalltalk, Prolog, etc.
- ▶ A partir de un algoritmo van a existir múltiples programas que implementan dicho algoritmo.

35

## Paradigmas de Programación

- ▶ Existen distintos paradigmas de programación
  - ▶ Formas de pensar un algoritmo que cumpla una especificación
  - ▶ Cada uno tiene asociado un conjunto de lenguajes
  - ▶ Nos llevan a encarar la programación según ese paradigma
- ▶ Haskell pertenece al paradigma de programación funcional
  - ▶ programa = colección de funciones
    - ▶ Transforman datos de entrada en un resultado
  - ▶ Los lenguajes funcionales nos dan herramientas para explicarle a la computadora cómo computar esas funciones

36

## Programación funcional

- Un **programa** en un lenguaje funcional es un **conjunto de ecuaciones orientadas** que definen una o más funciones.

Por ejemplo:

```
doble x = x + x
```

- La **ejecución** de un programa en este caso corresponde a la **evaluación de una expresión**, habitualmente solicitada desde la consola del entorno de programación.

```
Prelude> doble 10  
20
```

- La expresión se evalúa usando las ecuaciones definidas en el programa, hasta llegar a un resultado.
- Las ecuaciones orientadas junto con el mecanismo de reducción describen **algoritmos**.

37

## Ecuaciones

Para determinar el valor de la aplicación de una función se reemplaza cada expresión por otra, según las ecuaciones.

- Este proceso puede no terminar, aún con ecuaciones bien definidas.

- Por ejemplo, consideremos la expresión:

```
doble (1 + 1)
```

Si reemplazamos  $1 + 1$  por **doble 1** obtenemos **doble (doble 1)**

Y ahora podemos reemplazar **doble 1** por  $1 + 1$

Volvimos a empezar...

```
doble (1 + 1) ~~ doble (doble 1) ~~ doble (1 + 1) ~~ ...
```

38

## Ecuaciones orientadas

- Lado **izquierdo**: expresión a definir
- Lado **derecho**: definición
- Cálculo del valor de una expresión : reemplazamos las subexpresiones que sean lado izquierdo de una ecuación por su lado derecho

Ejemplo: **doble x = x + x**  
 $doble (1 + 1) ~~ (1 + 1) + (1 + 1) ~~ 2 + (1 + 1) ~~ 2 + 2 ~~ 4$

También podría ser:

$doble (1 + 1) ~~ doble 2 ~~ 2 + 2 ~~ 4$

Más adelante veremos cómo funciona Haskell en particular.

39

## Transparencia referencial

Es la propiedad de un lenguaje que garantiza que el valor de una expresión depende exclusivamente de sus subexpresiones.

Por lo tanto,

- Cada expresión del lenguaje representa siempre el mismo valor en cualquier lugar de un programa
- Es una propiedad muy importante en el paradigma de la programación funcional.
  - En otros paradigmas el significado de una expresión depende del contexto
- Es muy útil para verificar correctitud (demostrar que se cumple la especificación)
  - Podemos usar propiedades ya probadas para subexpresiones
  - El valor no depende de la historia
  - Valen en cualquier contexto

40

## Formación de expresiones

### ► Expresiones atómicas

- ▶ También se llaman **formas normales**
- ▶ Son las más simples, no se puede **reducir** más.
- ▶ Son la forma más intuitiva de representar un valor
- ▶ Ejemplos:
  - ▶ 2
  - ▶ False
  - ▶ (3, True)
- ▶ Es común llamarlas "valores" aunque no son un valor, *denotan* un valor, como las demás expresiones

### ► Expresiones compuestas

- ▶ Se construyen combinando expresiones atómicas con operaciones
- ▶ Ejemplos:
  - ▶ 1+1
  - ▶ 1==2
  - ▶ (4-1, True || False)

41

## Formación de expresiones

### ► Algunas cadenas de símbolos no forman expresiones

#### ▶ por problemas sintácticos:

- ▶ ++1-
- ▶ (True
- ▶ ('a',)

#### ▶ o por error de tipos:

- ▶ 2 + False
- ▶ 2 || 'a'
- ▶ 4 \* 'b'

### ► Para saber si una expresión está bien formada, aplicamos

- ▶ Reglas sintácticas
- ▶ Reglas de asignación o inferencia de tipos (algoritmo de Hindley-Milner)

### ► En Haskell toda expresión denota un valor, y ese valor pertenece a un tipo de datos y no se puede usar como si fuera de otro tipo distinto.

- ▶ Haskell es un lenguaje **fuertemente tipado**

42

## ¿Cómo ejecuta Haskell?

¿Qué sucede en Haskell cuando escribo una expresión? ¿Cómo se transforma esa expresión en un resultado?

### ► Dado el siguiente programa:

```
resta x y = x - y
```

```
suma x y = x + y
```

```
negar x = -x
```

### ► ¿Qué sucede al evaluar la expresión suma (resta 2 (negar 42)) 4

43

## Reducción

suma (resta 2 (negar 42)) 4

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
2. La subexpresión a reemplazar es alguna **instancia** del lado izquierdo de alguna ecuación orientada del programa, y se la llama **radical** o **redex** (*reducible expression*).
  - ▶ Buscamos un redex: suma (resta 2 (negar 42)) 4
3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
  - ▶ resta x y = x - y
  - ▶ x ← 2
  - ▶ y ← (negar 42)
4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.
  - ▶ suma (resta 2 (negar 42)) 4 ~~> suma (2 - (negar 42)) 4
5. Si la expresión resultante aún puede reducirse, volvemos al paso 1, sino llegamos a una expresión atómica (forma normal) y ese es el resultado del cálculo.

suma (2 - (negar 42)) 4 ~~> suma (2 - (- 42)) 4suma (2 - (- 42)) 4 ~~> suma (44)  
4suma (44) 4 ~~> 44 + 4 ~~> 48

44

## Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
suma (3+4) (suc (2*3))
~~ (3+4) + (suc (2*3))
~~ 7 + (suc (2*3))
~~ 7 + ((2*3) + 1)
~~ 7 + (6 + 1)
~~ 7 + 7
~~ 14
```

Otros lenguajes de programación (C, C++, Pascal, Java) tienen un orden de **evaluación eager** (ansioso): primero se evalúan los argumentos y después la función.

45

## Definiciones de funciones por casos

Podemos usar **guardas** para definir funciones por casos:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

```
f n | n == 0 = 1
     | n /= 0 = 0
```

Palabra clave “si no”.

```
f n | n == 0 = 1
     | otherwise = 0
```

47

## Indefinición

► Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** ( $\perp$ ).

► ¿Cómo podemos clasificar las funciones?

► Funciones **totales**: nunca se indefinen.  
`suc x = x + 1`

► Funciones **parciales**: hay argumentos para los cuales se indefinen.  
`division x y = div x y`

¿Qué pasa al reducir las siguientes expresiones en Haskell?

- `(division 1 1 ==0) && (division 1 0 ==1)`
- `(division 1 1 ==1) && (division 1 0 ==1)`
- `(division 1 0 ==1) && (division 1 1 ==1)`

¿Y si hicéramos una evaluación eager o ansiosa?

46

## La función signo

$$\text{signo}(n) = \begin{cases} 1 & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ -1 & \text{si } n < 0 \end{cases}$$

```
signo n | n > 0 = 1
         | n == 0 = 0
         | n < 0 = -1
```

```
signo n | n > 0 = 1
         | n == 0 = 0
         | otherwise = -1
```

48

## La función máximo

```
maximo x y | x ≥ y = x  
| otherwise = y
```

49

## ¿Qué hacen las siguientes funciones?

```
f1 n | n ≥ 3 = 5
```

```
f2 n | n ≥ 3 = 5  
| n ≤ 1 = 8
```

```
f3 n | n ≥ 3 = 5  
| n == 2 = undefined  
| otherwise = 8
```

50

## ¿Qué hacen las siguientes funciones?

```
f4 n | n ≥ 3 = 5  
| n ≤ 9 = 7
```

```
f5 n | n ≤ 9 = 7  
| n ≥ 3 = 5
```

**Prestar atención al orden de las guardas.** ¡Cuando las condiciones se solapan, el orden de las guardas cambia el comportamiento de la función!

51

## Otra posibilidad usando *pattern matching*

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

```
f n | n == 0 = 1  
| n /= 0 = 0
```

También se puede hacer:

```
f 0 = 1  
f n = 0
```

52

## Otra posibilidad usando *pattern matching*

$$\text{signo}(n) = \begin{cases} 1 & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ -1 & \text{si } n < 0 \end{cases}$$

```
signo n | n > 0 = 1
          | n == 0 = 0
          | n < 0 = -1
```

También se puede hacer:

```
signo 0 = 0
signo n | n > 0 = 1
          | otherwise = -1
```

53

## Un ejemplo con especificación

Dados tres números  $a$ ,  $b$  y  $c$ , calcular la cantidad de soluciones reales de la ecuación cuadrática:  $aX^2 + bX + c = 0$ .

```
problema cantidadDeSoluciones(a : Z, b : Z, c : Z) : Z {
    requiere: {a ≠ 0}
    asegura: {res = 2 ↔ discriminante(a, b, c) > 0}
    asegura: {res = 1 ↔ discriminante(a, b, c) = 0}
    asegura: {res = 0 ↔ discriminante(a, b, c) < 0}
}

problema discriminante(a : Z, b : Z, c : Z) : Z {
    requiere: {a ≠ 0}
    asegura: {res = b^2 - 4 * a * c}
}

cantidadDeSoluciones a b c | b^2 - 4*a*c > 0 = 2
                           | b^2 - 4*a*c == 0 = 1
                           | otherwise = 0
```

Otra posibilidad:

```
cantidadDeSoluciones a b c | discriminante > 0 = 2
                           | discriminante == 0 = 1
                           | otherwise = 0
where discriminante = b^2 - 4*a*c
```

54

## Tipos de datos

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

### Ejemplos:

1.  $\text{Int} = (\mathbb{Z}, \{+, -, *, \text{div}, \text{mod}\})$  es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
  2.  $\text{Float} = (\mathbb{Q}, \{+, -, *, /\})$  es el tipo de datos que representa a los racionales, con la aritmética de **punto flotante**.
  3.  $\text{Char} = (\{\text{'a'}, \text{'A'}, \text{'1'}, \text{'?'}, \{\text{ord}, \text{chr}, \text{isUpper}, \text{toUpper}\}\})$  es el tipo de datos que representan los caracteres.
  4.  $\text{Bool} = (\{\text{True}, \text{False}\}, \{\&&, \|\|, \text{not}\})$  representa a los valores lógicos.
- Podemos declarar explícitamente el tipo de datos del *dominio* y *codominio* de las funciones. A esto lo llamamos dar la **signatura** de la función.
- No es estrictamente necesario hacerlo (Haskell puede inferir el tipo), pero suele ser una buena práctica (y **¡nosotros lo vamos a pedir!**).

55

## Aplicación de funciones

En programación funcional (como en matemática) las funciones son elementos (valores).

Notación  $f :: T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \dots \rightarrow T_n$

- Una función es un valor
- la operación básica que podemos realizar con ese valor es la **aplicación**
  - Aplicar la función a un elemento para obtener un resultado
- Sintácticamente, la aplicación se escribe como una yuxtaposición (la función seguida de su parámetro).
- Por ejemplo: sea  $f :: T_1 \rightarrow T_2$ , y  $e$  de tipo  $T_1$  entonces  $f \ e$  es una expresión de tipo  $T_2$ .  
Sea  $doble :: \text{Int} \rightarrow \text{Int}$ , entonces  $doble \ 2$  representa un número entero.

56

## Ejemplos de funciones con la signatura

```
maximo :: Int → Int → Int
maximo x y | x ≥ y = x
            | otherwise = y

maximoRac :: Float → Float → Float
maximoRac x y | x ≥ y = x
               | otherwise = y

esMayorA9 :: Int → Bool
esMayorA9 n | n > 9 = True
            | otherwise = False

esPar :: Int → Bool
esPar n | mod n 2 == 0 = True
        | otherwise = False

esPar2 :: Int → Bool
esPar2 n = mod n 2 == 0

esImpar :: Int → Bool
esImpar n = not (esPar n)
```

57

## Otro ejemplo más raro:

```
funcionRara :: Float → Float → Bool → Bool
funcionRara x y z = (x ≥ y) || z
```

Otras posibilidades, usando *pattern matching*:

```
funcionRara :: Float → Float → Bool → Bool
funcionRara x y True = True
funcionRara x y False = x ≥ y
```

```
funcionRara :: Float → Float → Bool → Bool
funcionRara _ _ True = True
funcionRara x y False = x ≥ y
```

## Nueva familia de tipos: Tuplas

### Tuplas

- Dados tipos  $A_1, \dots, A_k$ , el **tipo  $k$ -upla** ( $A_1, \dots, A_k$ ) es el conjunto de las  $k$ -uplas  $(v_1, \dots, v_k)$  donde  $v_i$  es de tipo  $A_i$

```
(1, 2)          :: (Int, Int)
(1.1, 3.2, 5.0) :: (Float, Float, Float)
(True, (1, 2))  :: (Bool, (Int, Int))
(True, 1, 2)    :: (Bool, Int, Int)
```

- En Haskell hay infinitos tipos de tuplas

### Funciones de acceso a los valores de un par en Prelude

- **fst** ::  $(a, b) \rightarrow a$       Ejemplo:  $\text{fst} (1 + 4, 2) \rightsquigarrow 5$
- **snd** ::  $(a, b) \rightarrow b$       Ejemplo:  $\text{snd} (1, (2, 3)) \rightsquigarrow (2, 3)$

Ejemplo: suma de vectores en  $\mathbb{R}^2$

```
suma :: (Float, Float) → (Float, Float) → (Float, Float)
suma v w = ((fst v) + (fst w), (snd v) + (snd w))
```

59

58

## Introducción a la Programación Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Polimorfismo, Currificación y Recursión sobre enteros

- ▶ Se llama polimorfismo a una función que puede aplicarse a distintos tipos de datos (sin redefinirla).
- ▶ se usa cuando el comportamiento de la función no depende paramétricamente del tipo de sus argumentos
- ▶ lo vimos en el lenguaje de especificación con las funciones genéricas.
- ▶ En Haskell los polimorfismos se escriben usando **variables de tipo** y conviven con el tipado fuerte.
- ▶ Ejemplo de una función polimórfica: la función identidad.

1

## Variables de tipos

¿Qué tipo tienen las siguientes funciones?

```
identidad x = x
primero x y = x
segundo x y = y
constante5 x y z = 5
```

### Variables de tipo

- ▶ Son parámetros que se escriben en la signatura usando variables minúsculas
- ▶ En lugar de valores, denotan tipos
- ▶ Cuando se invoca la función se usa como argumento el tipo del valor

## Variables de tipo (cont.)

### Funciones con variables de tipo

```
identidad :: t -> t
identidad x = x

primero :: tx -> ty -> tx
primero x y = x

segundo :: tx -> ty -> ty
segundo x y = y

constante5 :: tx -> ty -> tz -> Int
constante5 x y z = 5

mismoTipo :: t -> t -> Bool
mismoTipo x y = True
```

Si dos argumentos deben tener el mismo tipo, se debe usar la misma variable de tipo

- ▶ Luego, primero `True 5 :: Bool`, pero `mismoTipo 1 True 0` no tipa

3

4

# Clases de tipos

¿Qué tipo tienen las siguientes funciones?

```
triple x = 3*x

maximo x y | x >= y = x
             | otherwise = y

distintos x y = x /= y
```

## Clases de tipos

- ▶ Conjunto de tipos a los que se le pueden aplicar ciertas funciones
- ▶ Un tipo puede pertenecer a distintas clases

Los **Float** son números (**Num**), con orden (**Ord**), de punto flotante (**Floating**), etc.

## En este curso

- ▶ No vamos a evaluar el uso de clases de tipos, pero ...
- ▶ ...saber la mecánica permite comprender los mensajes del compilador de Haskell (GHCi)

# Clases de tipos (cont)

Las clases de tipos se describen como restricciones sobre variables de tipos

```
triple :: (Num t) => t -> t
triple x = 3*x

maximo :: (Ord t) => t -> t -> t
maximo x y | x >= y = x
             | otherwise = y

distintos :: (Eq t) => t -> t -> Bool
distintos x y = x /= y

— Cantidad de raíces de la ecuación: ax^2 + bx + c
cantidadDeSoluciones :: (Num t, Ord t) => t -> t -> t -> Int
cantidadDeSoluciones a b c | discriminante > 0 = 2
                           | discriminante == 0 = 1
                           | otherwise = 0
                           where discriminante = b^2 - 4*a*c

pepe :: (Floating t, Eq t, Num u, Eq u) => t -> t -> u -> Bool
pepe x y z = sqrt (x + y) == x && 3*z == 0
```

(**Floating** t, **Eq** t, **Num** u, **Eq** u) => ... significa que:

- ▶ la variable t tiene que ser de un tipo que pertenezca a **Floating** y **Eq**
- ▶ la variable u tiene que ser de un tipo que pertenezca a **Num** y **Eq**

# Clases de tipos (cont)

## Clase de tipos

- ▶ **Conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**

## Algunas clases:

1. **Integral** := ({ **Int**, **Integer**, ... }, { **mod**, **div**, ... })
2. **Fractional** := ({ **Float**, **Double**, ... }, { **(/)**, ... })
3. **Floating** := ({ **Float**, **Double**, ... }, { **sqrt**, **sin**, **cos**, **tan**, ... })
4. **Num** := ({ **Int**, **Integer**, **Float**, **Double**, ... }, { **(+)**, **(\*)**, **abs**, ... })
5. **Ord** := ({ **Bool**, **Int**, **Integer**, **Float**, **Double**, ... }, { **(<=)**, **compare** })
6. **Eq** := ({ **Bool**, **Int**, **Integer**, **Float**, **Double**, ... }, { **(==)**, **(/=)** })

5

# Ejercitación conjunta

Averiguar el tipo asignado por Haskell a las siguientes funciones

```
f1 x y z = x**y + z <= x+y**z

f2 x y = (sqrt x) / (sqrt y)

f3 x y = div (sqrt x) (sqrt y)

f4 x y z | x == y = z
           | x ** y == y = x
           | otherwise = y

f5 x y z | x == y = z
           | x ** y == y = z
           | otherwise = z
```

6

¿Qué error ocurre cuando ejecutamos f4 5 5 **True**? ¿Tiene sentido?

¿Y si ejecutamos f5 5 5 **True**? ¿Qué cambió?

8

# Nueva familia de tipos: Tuplas

## Tuplas

- Dados tipos  $A_1, \dots, A_k$ , el **tipo  $k$ -upla** ( $A_1, \dots, A_k$ ) es el conjunto de las  $k$ -uplas  $(v_1, \dots, v_k)$  donde  $v_i$  es de tipo  $A_i$

```
(1, 2)      :: (Int, Int)
(1.1, 3.2, 5.0) :: (Float, Float, Float)
(True, (1, 2)) :: (Bool, (Int, Int))
(True, 1, 2)   :: (Bool, Int, Int)
```

- En Haskell hay infinitos tipos de tuplas

### Funciones de acceso a los valores de un par en Prelude

- **fst** ::  $(a, b) \rightarrow a$       Ejemplo: `fst (1 + 4, 2) ~\~ 5`
- **snd** ::  $(a, b) \rightarrow b$       Ejemplo: `snd (1, (2, 3)) ~\~ (2, 3)`

Ejemplo: suma de vectores en  $\mathbb{R}^2$

```
suma :: (Float, Float) -> (Float, Float) -> (Float, Float)
suma v w = ((fst v) + (fst w), (snd v) + (snd w))
```

Podemos usar *pattern matching* para acceder a los valores de una tupla

```
suma (vx, vy) (wx, wy) = (vx + wx, vy + wy)
```

## Pattern matching sobre tuplas

Podemos usar *pattern matching* sobre constructores de tuplas y números

```
esOrigen :: (Float, Float) -> Bool
esOrigen (0, 0) = True
esOrigen (_, _) = False

angulo0 :: (Float, Float) -> Bool
angulo0 (_, 0) = True
angulo0 (_, _) = False

{-
No podemos usar dos veces la misma variable
angulo45 :: (Float, Float) -> Bool
angulo45 (x,x) = True
angulo45 (_,_) = False
-}
angulo45 :: (Float, Float) -> Bool
angulo45 (x,y) = x == y
```

```
patternMatching :: (Float, (Bool, Int), (Bool, (Int, Float))) -> (Float, (Int, Float))
patternMatching (f1, (True, _), (_, (0, f2))) = (f1, (1, f2))
patternMatching (_, _, (_, (0, f))) = (f, (0, f))
```

9

10

## Parámetros vs. tuplas

¿Conviene tener dos parámetros escalares o un parámetro dupla?

```
suma :: (Float, Float) -> (Float, Float) -> (Float, Float)
suma (vx, vy) (wx, wy) = (vx + wx, vy + wy)

— normaVectorial2 x y es la norma de (x,y)
normaVectorial2 :: Float -> Float -> Float
normaVectorial2 x y = sqrt (x^2 + y^2)

— normaVectorial1 (x,y) es la norma de (x,y)
normaVectorial1 :: (Float, Float) -> Float
normaVectorial1 (x,y) = sqrt (x^2 + y^2)

normalSuma :: (Float, Float) -> (Float, Float) -> Float
normalSuma v1 v2 = normaVectorial1 (suma v1 v2)

norma2Suma :: (Float, Float) -> (Float, Float) -> Float
norma2Suma v1 v2 = normaVectorial2 (fst s) (snd s)
    where s = suma v1 v2
```

## Curificación

- Diferencia entre **promedio1** y **promedio2**

- **promedio1** ::  $(\text{Float}, \text{Float}) \rightarrow \text{Float}$   
`promedio1 (x,y) = (x+y)/2`
- **promedio2** ::  $\text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$   
`promedio2 x y = (x+y)/2`

11

12

## Curificación

- ▶ Diferencia entre `promedio1` y `promedio2`
  - ▶ `promedio1 :: (Float, Float) -> Float`  
`promedio1 (x,y) = (x+y)/2`
  - ▶ `promedio2 :: Float -> Float -> Float`  
`promedio2 x y = (x+y)/2`
- ▶ solo cambia el tipo de datos de la función
  - ▶ `promedio1` recibe un solo parámetro (una dupla)
  - ▶ `promedio2` recibe dos `Float` separados por un espacio
  - ▶ para declararla, separamos los tipos de los parámetros con una flecha
  - ▶ tiene motivos teóricos y prácticos (que no veremos ahora)
- ▶ la notación se llama **curificación** en honor al matemático Haskell B. Curry
- ▶ para nosotros, alcanza con ver que evita el uso de varios signos de puntuación (comas y paréntesis)
  - ▶ `promedio1 (promedio1 (2, 3), promedio1 (1, 2))`
  - ▶ `promedio2 (promedio2 2 3) (promedio2 1 2)`

## Funciones binarias: notación prefija vs. infija

### Funciones binarias

- ▶ Notación prefija: función antes de los argumentos (e.g., `suma x y`)
- ▶ Notación infija: función entre argumentos (e.g. `x + y`, `5 * 3`, etc)
- ▶ La notación infija se permite para funciones cuyos nombres son operadores
- ▶ El nombre real de una función definido por un operador `•` es `(•)`
- ▶ Se puede usar el nombre real con notación prefija, e.g. `(+) 2 3`
- ▶ Haskell permite definir nuevas funciones con símbolos, e.g., `(**)` (no hacerlo!)
- ▶ Una función binaria `f` puede ser usada de forma infija escribiendo '`f`'

### Ejemplos:

```
(>=) :: Ord a => a -> a -> Bool  
(>=) 5 3 —evalua a True  
(==) :: Eq a => a -> a -> Bool  
(==) 3 4 —evalua a False  
(^) :: (Num a, Int b) => a -> b -> a  
(^) 2 5 —evalua 32.0  
mod :: (Integral a) => a -> a -> a  
5 `mod` 3 —evalua 2  
div :: (Integral a) => a -> a -> a  
5 `div` 3 —evalua 1
```

12

13

## Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.

## Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número  $n \in \mathbb{N}_0$ ?

14

14

## Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas” .
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número  $n \in \mathbb{N}_0$ ?

$$n! = \prod_{k=1}^n k$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n - 1)! & \text{si } n > 0 \end{cases}$$

## Recursión

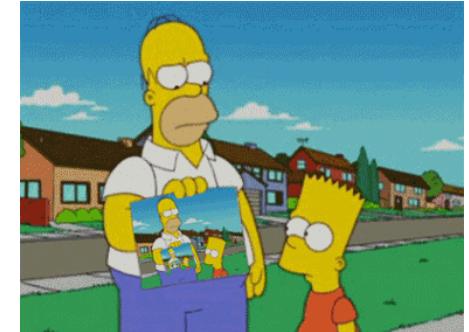
- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas” .
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número  $n \in \mathbb{N}_0$ ?

$$n! = \prod_{k=1}^n k$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n - 1)! & \text{si } n > 0 \end{cases}$$

¡La segunda definición de factorial involucra a esta misma función del lado derecho!

```
factorial :: Int -> Int
factorial n
| n == 0 = 1
| n > 0 = n * factorial (n-1)
```



14

14

## Recursión y reducción

¿Podemos definirla usando otherwise?

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
            | otherwise = n * factorial (n-1)
```

15

15

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
            | otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
            | otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

15

15

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
            | otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión factorial 3?

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
            | otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión factorial 3?

```
factorial 3
```

15

15

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
| otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión factorial 3?

factorial 3 ~~~ 3 \* factorial 2

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
| otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión factorial 3?

factorial 3 ~~~ 3 \* factorial 2 ~~~ 3 \* 2 \* factorial 1

15

15

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
| otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión factorial 3?

factorial 3 ~~~ 3 \* factorial 2 ~~~ 3 \* 2 \* factorial 1 ~~~  
~~~ 6 \* factorial 1

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
| otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión factorial 3?

factorial 3 ~~~ 3 \* factorial 2 ~~~ 3 \* 2 \* factorial 1 ~~~  
~~~ 6 \* factorial 1 ~~~ 6 \* 1 \* factorial 0

15

15

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
| otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión factorial 3?

```
factorial 3 ~~ 3 * factorial 2 ~~ 3 * 2 * factorial 1 ~~
~~ 6 * factorial 1 ~~ 6 * 1 * factorial 0 ~~ 6 * factorial 0 ~~
~~ 6 * 1
```

## Recursión y reducción

¿Podemos definirla usando otherwise?

```
factorial :: Int -> Int
factorial n | n == 0 = 1
| otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión factorial 3?

```
factorial 3 ~~ 3 * factorial 2 ~~ 3 * 2 * factorial 1 ~~
~~ 6 * factorial 1 ~~ 6 * 1 * factorial 0 ~~ 6 * factorial 0 ~~
~~ 6 * 1 ~~ 6
```

15

15

## Asegurarse de llegar a un caso base

Veamos este programa recursivo para determinar si un entero positivo es par:

```
esPar :: Int -> Bool
esPar n | n==0 = True
| otherwise = esPar (n-2)
```

¿Qué problema tiene esta función?

## Asegurarse de llegar a un caso base

Veamos este programa recursivo para determinar si un entero positivo es par:

```
esPar :: Int -> Bool
esPar n | n==0 = True
| otherwise = esPar (n-2)
```

¿Qué problema tiene esta función?

¿Cómo se arregla?

16

16

## Asegurarse de llegar a un caso base

Veamos este programa recursivo para determinar si un entero positivo es par:

```
esPar :: Int -> Bool
esPar n | n==0 = True
| otherwise = esPar (n-2)
```

¿Qué problema tiene esta función?

¿Cómo se arregla?

```
esPar :: Int -> Bool
esPar n | n==0 = True
| n==1 = False
| otherwise = esPar (n-2)
```

```
esPar :: Int -> Bool
esPar n | n==0 = True
| otherwise = not (esPar (n-1))
```

## ¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
  - ▶ en el paso recursivo, **suponiendo** que tenemos el resultado para el caso anterior, ¿qué falta para poder obtener el resultado que quiero? En este caso, suponemos ya calculado factorial (n-1) y lo combinamos multiplicándolo por n para lograr obtener factorial n.

16

## ¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
  - ▶ en el paso recursivo, **suponiendo** que tenemos el resultado para el caso anterior, ¿qué falta para poder obtener el resultado que quiero? En este caso, suponemos ya calculado factorial (n-1) y lo combinamos multiplicándolo por n para lograr obtener factorial n.
  - ▶ además, identificamos el o los **casos base**. En el ejemplo de factorial, definimos como casos base la función sobre 0:  
`factorial n | n == 0 = 1`

## ¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
  - ▶ en el paso recursivo, **suponiendo** que tenemos el resultado para el caso anterior, ¿qué falta para poder obtener el resultado que quiero? En este caso, suponemos ya calculado factorial (n-1) y lo combinamos multiplicándolo por n para lograr obtener factorial n.
  - ▶ además, identificamos el o los **casos base**. En el ejemplo de factorial, definimos como casos base la función sobre 0:  
`factorial n | n == 0 = 1`
- ▶ Propiedades de una definición recursiva:
  - ▶ las **llamadas recursivas** tienen que “acercarse” a un caso base.
  - ▶ tiene que tener uno o más **casos base** que dependerán del tipo de llamado recursivo. Un caso base, es aquella expresión que no tiene paso recursivo.

17

17

## ¿Cómo pensar recursivamente?

- Casos bases: identificar el o los casos bases.
- Casos recursivos: **suponiendo que la llamada recursiva es correcta**, ¿qué tengo que hacer para completar la solución?

## ¿Cómo pensar recursivamente?

- Casos bases: identificar el o los casos bases.
- Casos recursivos: **suponiendo que la llamada recursiva es correcta**, ¿qué tengo que hacer para completar la solución?

Otro Ejemplo:

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
| n == 1 = 1
| n > 1 = ... sumaLosPrimerosNImpares (n-1) ...
```

- Verificar que ( $n==1$ ) es el caso base, está bien definido y no hay otros.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos!

## ¿Cómo pensar recursivamente?

- Casos bases: identificar el o los casos bases.
- Casos recursivos: **suponiendo que la llamada recursiva es correcta**, ¿qué tengo que hacer para completar la solución?

Otro Ejemplo:

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
| n == 1 = 1
| n > 1 = ... sumaLosPrimerosNImpares (n-1) ...
```

- Verificar que ( $n==1$ ) es el caso base, está bien definido y no hay otros.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos!

Con el paso anterior resuelto: ¿Qué falta para que el nuevo paso esté resuelto?

18

## ¿Cómo pensar recursivamente?

- Casos bases: identificar el o los casos bases.
- Casos recursivos: **suponiendo que la llamada recursiva es correcta**, ¿qué tengo que hacer para completar la solución?

Otro Ejemplo:

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
| n == 1 = 1
| n > 1 = ... sumaLosPrimerosNImpares (n-1) ...
```

- Verificar que ( $n==1$ ) es el caso base, está bien definido y no hay otros.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos!

Con el paso anterior resuelto: ¿Qué falta para que el nuevo paso esté resuelto?

```
| n > 1 = n_esimoImpar + sumaLosPrimerosNImpares (n-1)
```

Cambiamos el problema: ahora sólo falta definir `n_esimoImpar`.

18

18

## ¿Cómo pensar recursivamente?

- ▶ Casos bases: identificar el o los casos bases.
- ▶ Casos recursivos: **suponiendo que la llamada recursiva es correcta**, ¿qué tengo que hacer para completar la solución?

Otro Ejemplo:

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
| n == 1 = 1
| n > 1 = ... sumaLosPrimerosNImpares (n-1) ...
```

- ▶ Verificar que ( $n==1$ ) es el caso base, está bien definido y no hay otros.
- ▶ Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos!

Con el paso anterior resuelto: ¿Qué falta para que el nuevo paso esté resuelto?

```
| n > 1 = n_esimoImpar + sumaLosPrimerosNImpares (n-1)
```

Cambiamos el problema: ahora sólo falta definir `n_esimoImpar`.

```
| n > 1 = n_esimoImpar + sumaLosPrimerosNImpares (n-1)
where n_esimoImpar = 2*n - 1
```

## Inducción vs. Recursión

- ▶ Probar por inducción  
 $P(n) : \sum_{i=1}^n (2i - 1) = n^2$
- ▶ Implementar una función recursiva para  $f(n) = \sum_{i=1}^n (2i - 1)$

## Inducción vs. Recursión

- ▶ Probar por inducción  
 $P(n) : \sum_{i=1}^n (2i - 1) = n^2$
- ▶ Vale para  $n = 1$ :  $\sum_{i=1}^1 (2i - 1) = 1^2$
- ▶ Implementar una función recursiva para  $f(n) = \sum_{i=1}^n (2i - 1)$
- ▶ Caso base en Haskell: `f 1 = 1`

## Inducción vs. Recursión

- ▶ Probar por inducción  
 $P(n) : \sum_{i=1}^n (2i - 1) = n^2$
- ▶ Vale para  $n = 1$ :  $\sum_{i=1}^1 (2i - 1) = 1^2$
- ▶ Supongo que vale  $P(n)$ , quiero probar  $P(n + 1)$
- ▶ Implementar una función recursiva para  $f(n) = \sum_{i=1}^n (2i - 1)$
- ▶ Caso base en Haskell: `f 1 = 1`
- ▶ Supongo que ya sé calcular  $f(n - 1)$ , quiero calcular  $f(n)$

## Inducción vs. Recursión

- ▶ Probar por inducción  
 $P(n) : \sum_{i=1}^n (2i - 1) = n^2$
- ▶ Vale para  $n = 1$  :  $\sum_{i=1}^1 (2i - 1) = 1^2$
- ▶ Supongo que vale  $P(n)$ , quiero probar  $P(n + 1)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^n (2i - 1)$  y  $\sum_{i=1}^{n+1} (2i - 1)$ ?

$$\sum_{i=1}^{n+1} (2i - 1) = \left( \sum_{i=1}^n (2i - 1) \right) + 2n + 1$$

- ▶ Implementar una función recursiva para  $f(n) = \sum_{i=1}^n (2i - 1)$
- ▶ Caso base en Haskell: `f 1 = 1`
- ▶ Supongo que ya sé calcular  $f(n - 1)$ , quiero calcular  $f(n)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^{n-1} (2i - 1)$  y  $\sum_{i=1}^n (2i - 1)$ ?

$$\sum_{i=1}^n (2i - 1) = \left( \sum_{i=1}^{n-1} (2i - 1) \right) + 2n - 1$$

## Inducción vs. Recursión

- ▶ Probar por inducción  
 $P(n) : \sum_{i=1}^n (2i - 1) = n^2$
- ▶ Vale para  $n = 1$  :  $\sum_{i=1}^1 (2i - 1) = 1^2$
- ▶ Supongo que vale  $P(n)$ , quiero probar  $P(n + 1)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^n (2i - 1)$  y  $\sum_{i=1}^{n+1} (2i - 1)$ ?

$$\sum_{i=1}^{n+1} (2i - 1) = \left( \sum_{i=1}^n (2i - 1) \right) + 2n + 1$$

- ▶ Uso la Hipótesis Inductiva  $P(n)$ :

$$\sum_{i=1}^{n+1} (2i - 1) = n^2 + 2n + 1 = (n + 1)^2$$

- ▶ ¡Pero cómo?! ¡Estoy usando lo que quiero probar?!

- ▶ Implementar una función recursiva para  $f(n) = \sum_{i=1}^n (2i - 1)$
- ▶ Caso base en Haskell: `f 1 = 1`
- ▶ Supongo que ya sé calcular  $f(n - 1)$ , quiero calcular  $f(n)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^{n-1} (2i - 1)$  y  $\sum_{i=1}^n (2i - 1)$ ?

$$\sum_{i=1}^n (2i - 1) = \left( \sum_{i=1}^{n-1} (2i - 1) \right) + 2n - 1$$

- ▶ Uso la función que sé calcular:  
 $f(n) = f(n - 1) + 2n - 1$

En Haskell: `f n = f (n-1) + 2*n - 1`

- ▶ ¡Pero cómo?! ¡Estoy usando la función que quiero definir?!

## Inducción vs. Recursión

- ▶ Probar por inducción  
 $P(n) : \sum_{i=1}^n (2i - 1) = n^2$
- ▶ Vale para  $n = 1$  :  $\sum_{i=1}^1 (2i - 1) = 1^2$
- ▶ Supongo que vale  $P(n)$ , quiero probar  $P(n + 1)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^n (2i - 1)$  y  $\sum_{i=1}^{n+1} (2i - 1)$ ?

$$\sum_{i=1}^{n+1} (2i - 1) = \left( \sum_{i=1}^n (2i - 1) \right) + 2n + 1$$

- ▶ Uso la Hipótesis Inductiva  $P(n)$ :

$$\sum_{i=1}^{n+1} (2i - 1) = n^2 + 2n + 1 = (n + 1)^2$$

- ▶ Implementar una función recursiva para  $f(n) = \sum_{i=1}^n (2i - 1)$
- ▶ Caso base en Haskell: `f 1 = 1`
- ▶ Supongo que ya sé calcular  $f(n - 1)$ , quiero calcular  $f(n)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^{n-1} (2i - 1)$  y  $\sum_{i=1}^n (2i - 1)$ ?

$$\sum_{i=1}^n (2i - 1) = \left( \sum_{i=1}^{n-1} (2i - 1) \right) + 2n - 1$$

- ▶ Uso la función que sé calcular:  
 $f(n) = f(n - 1) + 2n - 1$

En Haskell: `f n = f (n-1) + 2*n - 1`

19

## Inducción vs. Recursión

- ▶ Probar por inducción  
 $P(n) : \sum_{i=1}^n (2i - 1) = n^2$
- ▶ Vale para  $n = 1$  :  $\sum_{i=1}^1 (2i - 1) = 1^2$
- ▶ Supongo que vale  $P(n)$ , quiero probar  $P(n + 1)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^n (2i - 1)$  y  $\sum_{i=1}^{n+1} (2i - 1)$ ?

$$\sum_{i=1}^{n+1} (2i - 1) = \left( \sum_{i=1}^n (2i - 1) \right) + 2n + 1$$

- ▶ Uso la Hipótesis Inductiva  $P(n)$ :

$$\sum_{i=1}^{n+1} (2i - 1) = n^2 + 2n + 1 = (n + 1)^2$$

- ▶ ¡Pero cómo?! ¡Estoy usando lo que quiero probar?!

- ▶ Implementar una función recursiva para  $f(n) = \sum_{i=1}^n (2i - 1)$
- ▶ Caso base en Haskell: `f 1 = 1`
- ▶ Supongo que ya sé calcular  $f(n - 1)$ , quiero calcular  $f(n)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^{n-1} (2i - 1)$  y  $\sum_{i=1}^n (2i - 1)$ ?

$$\sum_{i=1}^n (2i - 1) = \left( \sum_{i=1}^{n-1} (2i - 1) \right) + 2n - 1$$

- ▶ Uso la función que sé calcular:  
 $f(n) = f(n - 1) + 2n - 1$

En Haskell: `f n = f (n-1) + 2*n - 1`

- ▶ ¡Pero cómo?! ¡Estoy usando la función que quiero definir?!

- ▶ Probar por inducción  
 $P(n) : \sum_{i=1}^n (2i - 1) = n^2$
- ▶ Vale para  $n = 1$  :  $\sum_{i=1}^1 (2i - 1) = 1^2$
- ▶ Supongo que vale  $P(n)$ , quiero probar  $P(n + 1)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^n (2i - 1)$  y  $\sum_{i=1}^{n+1} (2i - 1)$ ?

$$\sum_{i=1}^{n+1} (2i - 1) = \left( \sum_{i=1}^n (2i - 1) \right) + 2n + 1$$

- ▶ Uso la Hipótesis Inductiva  $P(n)$ :

$$\sum_{i=1}^{n+1} (2i - 1) = n^2 + 2n + 1 = (n + 1)^2$$

- ▶ ¡Pero cómo?! ¡Estoy usando lo que quiero probar?!

- ▶ Ah, claro... vale  $P(1)$  y  $P(n) \Rightarrow P(n + 1)$ , entonces ¡vale para todo  $n$ !

- ▶ Implementar una función recursiva para  $f(n) = \sum_{i=1}^n (2i - 1)$
- ▶ Caso base en Haskell: `f 1 = 1`
- ▶ Supongo que ya sé calcular  $f(n - 1)$ , quiero calcular  $f(n)$
- ▶ ¿Qué relación hay entre  $\sum_{i=1}^{n-1} (2i - 1)$  y  $\sum_{i=1}^n (2i - 1)$ ?

$$\sum_{i=1}^n (2i - 1) = \left( \sum_{i=1}^{n-1} (2i - 1) \right) + 2n - 1$$

- ▶ Uso la función que sé calcular:  
 $f(n) = f(n - 1) + 2n - 1$

En Haskell: `f n = f (n-1) + 2*n - 1`

- ▶ ¡Pero cómo?! ¡Estoy usando la función que quiero definir?!

- ▶ Ah, claro... está definido  $f(1)$  y con  $f(n - 1)$  sé obtener  $f(n)$ , entonces ¡puedo calcular  $f$  para todo  $n$ !

19

19

## Generalización de funciones

### ¿Una fácil?.. o no tanto

- Implementar una función `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores de un número entero positivo.

problema `sumaDivisores(n : Z) : Z {`  
    requiere:  $\{n > 0\}$   
    asegura:  $\{res = \sum_{i=1}^n \text{if } (n \bmod i = 0) \text{ then } i \text{ else } 0 \text{ fi}\}$   
`}`

20

## Generalización de funciones

### ¿Una fácil?.. o no tanto

- Implementar una función `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores de un número entero positivo.

problema `sumaDivisores(n : Z) : Z {`  
    requiere:  $\{n > 0\}$   
    asegura:  $\{res = \sum_{i=1}^n \text{if } (n \bmod i = 0) \text{ then } i \text{ else } 0 \text{ fi}\}$   
`}`

20

## Generalización de funciones

### ¿Una fácil?.. o no tanto

- Implementar una función `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores de un número entero positivo.

problema `sumaDivisores(n : Z) : Z {`  
    requiere:  $\{n > 0\}$   
    asegura:  $\{res = \sum_{i=1}^n \text{if } (n \bmod i = 0) \text{ then } i \text{ else } 0 \text{ fi}\}$   
`}`

20

## Generalización de funciones

### ¿Una fácil?.. o no tanto

- Implementar una función `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores de un número entero positivo.

problema `sumaDivisores(n : Z) : Z {`  
    requiere:  $\{n > 0\}$   
    asegura:  $\{res = \sum_{i=1}^n \text{if } (n \bmod i = 0) \text{ then } i \text{ else } 0 \text{ fi}\}$   
`}`

**Pregunta clave:** ¿alcanza con hacer recursión sobre  $n$ ?

20

20

## Generalización de funciones

### ¿Una fácil?.. o no tanto

- Implementar una función `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores de un número entero positivo.

```
problema sumaDivisores(n : Z) : Z {  
    requiere: {n > 0}  
    asegura: {res =  $\sum_{i=1}^n$  if (n mod i = 0) then i else 0 fi}  
}
```

**Pregunta clave:** ¿alcanza con hacer recursión sobre  $n$ ?

No hay ninguna relación sencilla entre `sumaDivisores n` y `sumaDivisores (n-k)` (para ningún  $k$  particular).

## Generalización de funciones

### ¿Una fácil?.. o no tanto

- Implementar una función `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores de un número entero positivo.

```
problema sumaDivisores(n : Z) : Z {  
    requiere: {n > 0}  
    asegura: {res =  $\sum_{i=1}^n$  if (n mod i = 0) then i else 0 fi}  
}
```

**Pregunta clave:** ¿alcanza con hacer recursión sobre  $n$ ?

No hay ninguna relación sencilla entre `sumaDivisores n` y `sumaDivisores (n-k)` (para ningún  $k$  particular).

¿Qué sucede si definimos primero una función **más general** que devuelve la suma de los divisores de un número hasta cierto punto?

```
sumaDivisoresHasta :: Integer -> Integer -> Integer
```

20

20

## Generalización de funciones

### ¿Una fácil?.. o no tanto

- Implementar una función `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores de un número entero positivo.

```
problema sumaDivisores(n : Z) : Z {  
    requiere: {n > 0}  
    asegura: {res =  $\sum_{i=1}^n$  if (n mod i = 0) then i else 0 fi}  
}
```

**Pregunta clave:** ¿alcanza con hacer recursión sobre  $n$ ?

No hay ninguna relación sencilla entre `sumaDivisores n` y `sumaDivisores (n-k)` (para ningún  $k$  particular).

¿Qué sucede si definimos primero una función **más general** que devuelve la suma de los divisores de un número hasta cierto punto?

```
sumaDivisoresHasta :: Integer -> Integer -> Integer
```

Ahora **sí** existe una relación sencilla entre `sumaDivisoresHasta n k` y `sumaDivisoresHasta n (k-1)`. ¿Por qué?

## Generalización de funciones

Veamos cómo sería la especificación:

```
problema sumaDivisoresHasta(n : Z, k : Z) : Z {  
    requiere: {(n > 0)  $\wedge$  (k > 0)}  
    asegura: {res =  $\sum_{i=1}^k$  if (n mod i = 0) then i else 0 fi}  
}
```

20

21

## Generalización de funciones

Veamos cómo sería la especificación:

```
problema sumaDivisoresHasta(n : Z, k : Z) : Z {  
    requiere: {(n > 0) ∧ (k > 0)}  
    asegura: {res =  $\sum_{i=1}^k$  if (n mod i = 0) then i else 0 fi}  
}
```

Ahora podemos definir esta función en Haskell recursivamente

```
sumaDivisoresHasta :: Integer -> Integer -> Integer  
sumaDivisoresHasta n 1 = 1  
sumaDivisoresHasta n i | (mod n i == 0) = i +  
    sumaDivisoresHasta n (i-1)  
    | otherwise = sumaDivisoresHasta n (i-1)
```

## Generalización de funciones

Veamos cómo sería la especificación:

```
problema sumaDivisoresHasta(n : Z, k : Z) : Z {  
    requiere: {(n > 0) ∧ (k > 0)}  
    asegura: {res =  $\sum_{i=1}^k$  if (n mod i = 0) then i else 0 fi}  
}
```

Ahora podemos definir esta función en Haskell recursivamente

```
sumaDivisoresHasta :: Integer -> Integer -> Integer  
sumaDivisoresHasta n 1 = 1  
sumaDivisoresHasta n i | (mod n i == 0) = i +  
    sumaDivisoresHasta n (i-1)  
    | otherwise = sumaDivisoresHasta n (i-1)
```

¿Y por último, cómo definimos SumaDivisores utilizando lo anterior?

21

21

## Generalización de funciones

Veamos cómo sería la especificación:

```
problema sumaDivisoresHasta(n : Z, k : Z) : Z {  
    requiere: {(n > 0) ∧ (k > 0)}  
    asegura: {res =  $\sum_{i=1}^k$  if (n mod i = 0) then i else 0 fi}  
}
```

Ahora podemos definir esta función en Haskell recursivamente

```
sumaDivisoresHasta :: Integer -> Integer -> Integer  
sumaDivisoresHasta n 1 = 1  
sumaDivisoresHasta n i | (mod n i == 0) = i +  
    sumaDivisoresHasta n (i-1)  
    | otherwise = sumaDivisoresHasta n (i-1)
```

¿Y por último, cómo definimos SumaDivisores utilizando lo anterior?

```
sumaDivisores :: Integer -> Integer  
sumaDivisores n = sumaDivisoresHasta n n
```

## Generalización de funciones

Veamos cómo sería la especificación:

```
problema sumaDivisoresHasta(n : Z, k : Z) : Z {  
    requiere: {(n > 0) ∧ (k > 0)}  
    asegura: {res =  $\sum_{i=1}^k$  if (n mod i = 0) then i else 0 fi}  
}
```

Ahora podemos definir esta función en Haskell recursivamente

```
sumaDivisoresHasta :: Integer -> Integer -> Integer  
sumaDivisoresHasta n 1 = 1  
sumaDivisoresHasta n i | (mod n i == 0) = i +  
    sumaDivisoresHasta n (i-1)  
    | otherwise = sumaDivisoresHasta n (i-1)
```

¿Y por último, cómo definimos SumaDivisores utilizando lo anterior?

```
sumaDivisores :: Integer -> Integer  
sumaDivisores n = sumaDivisoresHasta n n
```

Entonces, *SumaDivisores*, ¿es una función recursiva?

21

21

## Recursión en más de un parámetro

Implementar la siguiente función:

$$f(n, m) = \sum_{i=1}^n \sum_{j=1}^m i^j$$

## Recursión en más de un parámetro

Implementar la siguiente función:

$$f(n, m) = \sum_{i=1}^n \sum_{j=1}^m i^j$$

Veamos primero la especificación:

```
problema sumatoriaDoble(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res = ∑_{i=1}^n ∑_{j=1}^m i^j}  
}
```

22

22

## Recursión en más de un parámetro

Implementar la siguiente función:

$$f(n, m) = \sum_{i=1}^n \sum_{j=1}^m i^j$$

Veamos primero la especificación:

```
problema sumatoriaDoble(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res = ∑_{i=1}^n ∑_{j=1}^m i^j}  
}
```

## Recursión en más de un parámetro

Implementar la siguiente función:

$$f(n, m) = \sum_{i=1}^n \sum_{j=1}^m i^j$$

Veamos primero la especificación:

```
problema sumatoriaDoble(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res = ∑_{i=1}^n ∑_{j=1}^m i^j}  
}
```

**Pregunta clave:** ¿alcanza con hacer recursión sobre  $n$ ?

22

22

## Recursión en más de un parámetro

Implementar la siguiente función:

$$f(n, m) = \sum_{i=1}^n \sum_{j=1}^m i^j$$

Veamos primero la especificación:

```
problema sumatoriaDoble(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res = ∑_{i=1}^n ∑_{j=1}^m i^j}  
}
```

**Pregunta clave:** ¿alcanza con hacer recursión sobre  $n$ ?

¿Qué sucede si definimos primero una función **más específica** que devuelve la sumatoria interna?

```
sumatoriaInterna :: Integer -> Integer -> Integer
```

## Recursión en más de un parámetro

Implementar la siguiente función:

$$f(n, m) = \sum_{i=1}^n \sum_{j=1}^m i^j$$

Veamos primero la especificación:

```
problema sumatoriaDoble(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res = ∑_{i=1}^n ∑_{j=1}^m i^j}  
}
```

**Pregunta clave:** ¿alcanza con hacer recursión sobre  $n$ ?

¿Qué sucede si definimos primero una función **más específica** que devuelve la sumatoria interna?

```
sumatoriaInterna :: Integer -> Integer -> Integer
```

Ahora parece más sencillo definir `sumatoriaDoble n m` utilizando `sumatoriaInterna n m`. ¿Cómo lo hacemos?

22

22

## Recursión en más de un parámetro

Veamos cómo sería la especificación:

```
problema sumatorialInterna(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res = ∑_{j=1}^m n^j}  
}
```

## Recursión en más de un parámetro

Veamos cómo sería la especificación:

```
problema sumatorialInterna(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res = ∑_{j=1}^m n^j}  
}
```

Ahora podemos definir esta función en Haskell recursivamente

```
sumatorialInterna :: Integer -> Integer -> Integer  
sumatorialInterna _ 0 = 0  
sumatorialInterna n j = n^j + sumatorialInterna n (j-1)
```

23

23

## Recursión en más de un parámetro

Veamos cómo sería la especificación:

```
problema sumatorialInterna(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res =  $\sum_{j=1}^m n^j$ }  
}
```

Ahora podemos definir esta función en Haskell recursivamente

```
sumatorialInterna :: Integer -> Integer -> Integer  
sumatorialInterna _ 0 = 0  
sumatorialInterna n j = n^j + sumatorialInterna n (j-1)
```

¿Y por último, cómo definimos sumatoriaDoble utilizando lo anterior?

## Recursión en más de un parámetro

Veamos cómo sería la especificación:

```
problema sumatorialInterna(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res =  $\sum_{j=1}^m n^j$ }  
}
```

Ahora podemos definir esta función en Haskell recursivamente

```
sumatorialInterna :: Integer -> Integer -> Integer  
sumatorialInterna _ 0 = 0  
sumatorialInterna n j = n^j + sumatorialInterna n (j-1)
```

¿Y por último, cómo definimos sumatoriaDoble utilizando lo anterior?

```
sumatoriaDoble :: Integer -> Integer -> Integer  
sumatoriaDoble 0 _ = 0  
sumatoriaDoble n m = sumatoriaDoble (n-1) m + sumatorialInterna n m
```

23

23

## Recursión en más de un parámetro

Veamos cómo sería la especificación:

```
problema sumatorialInterna(n : Z, m : Z) : Z {  
    requiere: {(n > 0) ∧ (m > 0)}  
    asegura: {res =  $\sum_{j=1}^m n^j$ }  
}
```

Ahora podemos definir esta función en Haskell recursivamente

```
sumatorialInterna :: Integer -> Integer -> Integer  
sumatorialInterna _ 0 = 0  
sumatorialInterna n j = n^j + sumatorialInterna n (j-1)
```

¿Y por último, cómo definimos sumatoriaDoble utilizando lo anterior?

```
sumatoriaDoble :: Integer -> Integer -> Integer  
sumatoriaDoble 0 _ = 0  
sumatoriaDoble n m = sumatoriaDoble (n-1) m + sumatorialInterna n m
```

Entonces, sumatoriaDoble, ¿cuántas recursiones involucra?

## Práctica 3: Ejercicio 6

Especificar e implementar la función `sumaDigitos :: Integer -> Integer` que calcula la suma de dígitos de un número natural. Para esta función pueden utilizar `div` y `mod`.

23

24

## Práctica 3: Ejercicio 7

Implementar la función `todosDigitosIguales :: Integer -> Bool` que determina si todos los dígitos de un número natural son iguales, es decir:

```
problema todosDigitosIguales(n :  $\mathbb{Z}$ ) : bool{  
    requiere:  $\{(n > 0)\}$   
    asegura:  $\{\text{res} \leftrightarrow \text{todos los dígitos de } n \text{ son iguales}\}$   
}
```

# Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Departamento de Computación - FCEyN - UBA

Listas. Recursión sobre listas

1

## Variables de tipos

¿Qué tipo tienen las siguientes funciones?

```
identidad x = x  
  
primero x y = x  
  
segundo x y = y  
  
constante5 x y z = 5
```

### Variables de tipo

- Son parámetros que se escriben en la firma usando variables minúsculas
- En lugar de valores, denotan tipos
- Cuando se invoca la función se usa como argumento el tipo del valor

3

## Polimorfismo

Repasando...

- Se llama polimorfismo a una función que puede aplicarse a distintos tipos de datos (sin redefinirla).
- se usa cuando el comportamiento de la función no depende paramétricamente del tipo de sus argumentos
- lo vimos en el lenguaje de especificación con las funciones genéricas.
- En Haskell los polimorfismos se escriben usando **variables de tipo** y conviven con el tipado fuerte.
- Ejemplo de una función polimórfica: la función identidad.

2

## Variables de tipo (cont.)

### Funciones con variables de tipo

```
identidad :: t → t  
identidad x = x  
  
primero :: tx → ty → tx  
primero x y = x  
  
segundo :: tx → ty → ty  
segundo x y = y  
  
constante5 :: tx → ty → tz → Int  
constante5 x y z = 5  
  
mismoTipo :: t → t → Bool  
mismoTipo x y = True
```

Si dos argumentos deben tener el mismo tipo, se debe usar la misma variable de tipo

- Luego, `primero True 5 :: Bool`, pero `mismoTipo 1 True 0` no tipa

4

## Especificación de un problema: Extensión

Variables de tipo

- ▶ Vamos a querer describir funciones polimórficas con nuestro lenguaje de especificación
- ▶ Veamos cómo podemos hacerlo...

5

## Especificación de un problema: Extensión

Variables de tipo

- ▶ El símbolo o nombre (letra) de la variable de tipo no se corresponde con ninguno de los tipos de datos conocidos. Es una representación genérica.
- ▶ Cada ocurrencia de una variable de tipo, **siempre** representa al mismo tipo de datos.

```
problema segundo(x : U, y : T) : T{  
    asegura devuelveElSegundo: {res = y}  
}
```

```
problema cantidadDeApariciones(s : seq(T), e : T) : Z {  
    asegura: {res =  $\sum_{i=0}^{|s|-1}$  (if s[i] = e then 1 else 0 fi)}  
}
```

7

## Especificación de un problema: Extensión

Variables de tipo

```
problema nombre(parámetros) : tipo de dato del resultado {  
    requiere etiqueta: { condiciones sobre los parámetros de entrada }  
    asegura etiqueta: { condiciones sobre los parámetros de salida }  
}  
▶ nombre: nombre que le damos al problema  
    ▶ será resuelto por una función con ese mismo nombre  
▶ parámetros: lista de parámetros separada por comas, donde cada parámetro contiene:  
    ▶ Nombre del parámetro  
    ▶ Tipo de datos del parámetro o una variable de tipo  
▶ tipo de dato del resultado: tipo de dato del resultado del problema (initialmente especificaremos funciones) o una variable de tipo  
    ▶ En los asegura, podremos referenciar el valor devuelto con el nombre de res  
▶ etiquetas: son nombres opcionales que nos servirán para nombrar declarativamente a las condiciones de los requiere o aseguras.
```

6

## Especificación de un problema: Extensión

Variables de tipo con restricciones

- ▶ Se puede restringir los posibles tipos de una variable de tipo mediante un requiere

```
problema suma(x : T, y : T) : T{  
    requiere: {T ∈ [N, Z, R]}  
    asegura: {res = x + y}  
}
```

8

## Pensemos en listas: Motivación

### Algunas operaciones

- ▶ `maximo :: Int → Int → Int`
- ▶ `maximo3 :: Int → Int → Int → Int`
- ▶ `maximo4 :: Int → Int → Int → Int → Int`
- ⋮
- ▶ `maximoN :: Int → Int → ⋯ → Int`

### Pregunta

¿Hay alguna manera de definir funciones que nos permitan trabajar con cantidades arbitrarias de elementos?

Más concretamente, ¿podemos definir una función máximo que funcione por igual para 2, 10 o una cantidad  $N$  de elementos?

Respuesta: ¡Sí!, usando **listas**.

9

## Un nuevo tipo: Listas

### Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`

10

## Un nuevo tipo: Listas

### Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`
- ▶ `[1, 2, 3, 4] :: [Int]`
- ▶ `[div 10 5, div 2 2] :: [Int]`
- ▶ `[[1], [2,3], [], [1,1000,2,0]] :: [[Int]]`

11

## Un nuevo tipo: Listas

### Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`
- ▶ `[1, 2, 3, 4] :: [Int]`
- ▶ `[div 10 5, div 2 2] :: [Int]`
- ▶ `[[1], [2,3], [], [1,1000,2,0]] :: [[Int]]`
- ▶ `[1, True]`
- ▶ `[(1,2), (3,4), (5,2)]`

¿Cuál es el tipo de esta lista?

12

## Operaciones

Algunas operaciones que nos brinda el Preludio de Haskell

- ▶ `head :: [a] → a`
- ▶ `tail :: [a] → [a]`
- ▶ `(:) :: a → [a] → [a]`

Tipar y evaluar las siguientes expresiones

- ▶ `head [(1,2), (3,4), (5,2)]`
- ▶ `tail [1,2,3,4,4,3,2,1]`
- ▶ `[1,2] : []`
- ▶ `head []`
- ▶ `head [1,2,3] : [4,5]`
- ▶ `head ([1,2,3] : [4,5])`
- ▶ `head ([1,2,3] : [4,5] : [])`

13

## Recursión sobre listas

¿Se puede pensar recursivamente en listas? ¿Cómo?

Implementar las siguientes funciones (en el pizarrón)

1. `longitud :: [Int] → Int`  
que indica cuántos elementos tiene una lista.
2. `sumatoria :: [Int] → Int`  
que indica la suma de los elementos de una lista.
3. `pertenece :: Int → [Int] → Bool`  
que indica si un elemento aparece en la lista. Por ejemplo:  
`pertenece 9 [] ~> False`  
`pertenece 9 [1,2,3] ~> False`  
`pertenece 9 [1,2,9,9,-1,0] ~> True`

Idea: Pensar cómo combinar el resultado de la función sobre la cola de la lista con el primer elemento. Recordar:

- ▶ `head [1, 2, 3] ~> 1`
- ▶ `tail [1, 2, 3] ~> [2, 3]`

15

## Creando listas

Formas rápidas para crear listas

Prueben las siguientes expresiones en GHCI

- ▶ `[1..100]`
- ▶ `[1,3..100]`
- ▶ `[100..1]`
- ▶ `[1..]`

### Ejercicio

- ▶ Escribir una expresión que denote la lista estrictamente decreciente de enteros que comienza con el número 1 y termina con el número -100.
- ▶ Escribir una expresión que denote la lista estrictamente creciente de enteros entre -20 y 20 que son congruentes a 1 módulo 4.

14

## Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (Bool, Int, tuplas). ¿Se puede hacer *pattern matching* en listas?

¿Cuál es la verdadera forma de las listas?

Las listas tienen dos “pintas”:

- ▶ `[]` (lista vacía)
- ▶ `algo : lista` (lista no vacía)

Escribir la función `longitud :: [Int] → Int` usando *pattern matching*

```
longitud [] = 0
longitud (_:xs) = 1 + longitud xs
```

Escribir la función `sumatoria :: [Int] → Int` usando *pattern matching*

```
sumatoria [] = 0
sumatoria (x:xs) = sumatoria xs + x
```

Ejercicio: volver a implementar la función `pertenece` utilizando *pattern matching*.

16

## Guía 5: Ejercicio 4

- ▶ `sacarBlancosRepetidos :: [Char] →[Char]`, que reemplaza cada subsecuencia de blancos contiguos de la primera lista por un solo blanco en la segunda lista.
- ▶ `contarPalabras :: [Char] →Integer`, que dada una lista de caracteres devuelve la cantidad de palabras que tiene.
- ▶ `palabras :: [Char] →[[Char]]`, que dada una lista arma una nueva lista con las palabras de la lista original.
- ▶ `palabraMasLarga :: [Char] →[Char]`, que dada una lista de caracteres devuelve su palabra más larga.
- ▶ `aplanar :: [[Char]] →[Char]`, que a partir de una lista de palabras arma una lista de caracteres concatenándolas.

17

# Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Departamento de Computación - FCEyN - UBA

Introducción a Validación & Verificación

1

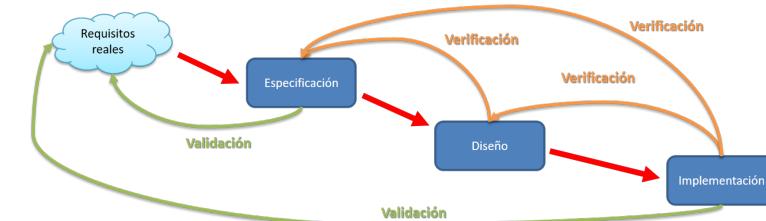
## Validación y Verificación

Según wikipedia...

En el contexto de la ingeniería de software, verificación y validación (V&V) es el proceso de comprobar que un sistema de software cumple con sus especificaciones y que cumple su propósito previsto. También puede ser denominado como el control de la **calidad del software**.

3

## Problema, especificación, algoritmo, programa



Dado un problema a resolver (de la vida real), queremos:

- ▶ Poder **describir** de una manera clara y única (especificación)
  - ▶ Esta descripción debería poder ser **validada** contra el problema real
- ▶ Poder **diseñar** una solución acorde a dicha especificación
  - ▶ Este diseño debería poder ser **verificado** con respecto a la especificación
- ▶ Poder implementar un programa acorde a dicho diseño
  - ▶ Este programa debería poder ser **verificado** con respecto a su especificación y su diseño
  - ▶ Este programa debería ser la solución al problema planteado

2

## Calidad en Software

Uno de los objetivos principales en el desarrollo de software es obtener productos de alta calidad

| Generalmente, se mide en atributos de calidad... |                                   |
|--------------------------------------------------|-----------------------------------|
| Confiabilidad                                    | Usabilidad                        |
| Corrección                                       | Robustez                          |
| Facilidad de Mantenimiento                       | Seguridad (en datos, acceso, ...) |
| Reusabilidad                                     | Funcionalidad                     |
| Verificabilidad + Claridad                       | Interoperabilidad                 |
| Etc..                                            |                                   |

4

## Asegurar la calidad vs Controlar la calidad

Una vez definidos los requerimientos de calidad, tengo que tener en cuenta que:

- ▶ Las calidad **no puede inyectarse al final**
- ▶ La calidad del producto depende de tareas realizadas durante **todo el proceso**
- ▶ Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos

5

## Nociones básicas

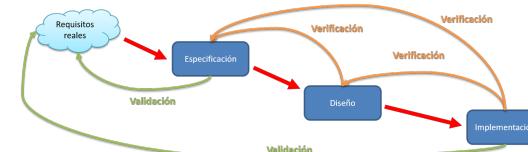
- ▶ Falla
  - ▶ Diferencia entre los resultados esperados y reales
- ▶ Defecto
  - ▶ Desperfecto en algún componente del sistema (en el texto del programa, una especificación, un diseño, etc), que origina una o más fallas
- ▶ Error (también llamado Bug)
  - ▶ Equivocación humana
    - ▶ Un **error** lleva a uno o más **defectos**, que están presentes en un producto de software
    - ▶ Un **defecto** lleva a cero, una o más **fallas**
    - ▶ Una **falla** es la manifestación del **defecto**

7

## Validación y Verificación

Son procesos que ayudan a mostrar que el software cubre las expectativas para las cuales fue construido: contribuyen a garantizar calidad.

- ▶ Validación
  - ▶ ¿Estamos haciendo el producto correcto?
  - ▶ El software debería hacer lo que el usuario requiere de él.
- ▶ Verificación
  - ▶ ¿Estamos haciendo el producto correctamente?
  - ▶ El software debería realizar lo que su especificación indica.



6

## El proceso de V&V

- ▶ V&V debería aplicarse en cada instancia del proceso de desarrollo
  - ▶ En rigor no sólo el código debe ser sometido a actividades de V&V sino también todos los subproductos generados durante el desarrollo del software
- ▶ Objetivos principales
  - ▶ Descubrir **defectos** en el sistema
  - ▶ Asegurar que el software **respete su especificación**
  - ▶ Determinar si satisface las **necesidades** de sus usuarios

8

## Metas de la V&V

- ▶ La verificación y la validación deberían **establecer la confianza** de que el software es adecuado a su propósito
- ▶ Esto **NO** significa que esté completamente **libre de defectos**
- ▶ Sino que debe ser lo **suficientemente bueno** para su uso previsto y el tipo de uso determinará el grado de confianza que se necesita

9

## Verificación estática y dinámica

- ▶ Una forma de realizar tareas de V&V es a través de análisis (de programas, modelos, especificaciones, documentos, etc.). En particular para el *código*, tenemos análisis estático y análisis dinámico
- ▶ **Dinámica:** trata con *ejecutar* y observar el *comportamiento* de un producto
- ▶ **Estática:** trata con el *análisis* de una *representación estática* del sistema para descubrir problemas

10

## Verificación estática y dinámica

### Técnicas de Verificación Estática

- Inspecciones, Revisiones
- Análisis de reglas sintácticas sobre código
- Análisis Data Flow sobre código
- Model checking
- Prueba de Teoremas
- Entre otras...

### Técnicas de Verificación Dinámica

- Testing
- Run-Time Monitoring. (pérdida de memoria, performance)
- Run-Time Verification
- Entre otras...

11

## Recap: ¿Por qué escribir la especificación del problema?

- ▶ Nos ayuda a entender mejor el problema
- ▶ Nos ayuda a construir el programa
  - ▶ Derivación (Automática) de Programas
- ▶ Nos ayuda a prevenir errores en el programa
  - ▶ Testing
  - ▶ Verificación (Automática) de Programas

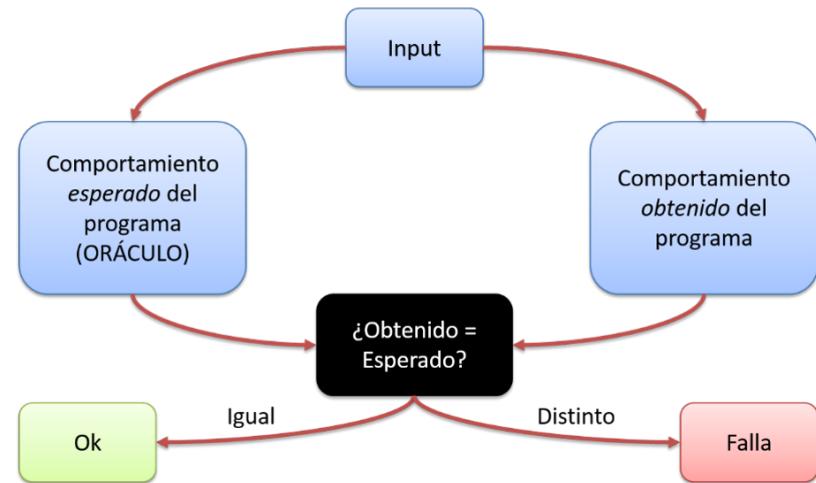
12

## ¿Qué es hacer testing?

- ▶ Es el proceso de ejecutar un producto para ...
  - ▶ Verificar que satisface los requerimientos (en nuestro caso, la **especificación**)
  - ▶ Identificar diferencias entre el comportamiento **real** y el comportamiento **esperado** (IEEE Standard for Software Test Documentation, 1983).
- ▶ Objetivo: encontrar defectos en el software.
- ▶ Representa entre el 30 % al 50 % del costo de un software confiable.

13

## ¿Cómo se hace testing?



14

## Niveles de Test

- ▶ Test de Sistema
  - ▶ Comprende todo el sistema. Por lo general constituye el test de aceptación.
- ▶ Test de Integración
  - ▶ Test orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto
  - ▶ Testeamos la interacción, la comunicación entre partes
- ▶ Test de Unidad
  - ▶ Se realiza sobre una unidad de código pequeña, claramente definida.
    - ▶ ¿Qué es una unidad? Depende...



15

## Test Input, Test Case y Test Suite

- ▶ **Programa bajo test:** Es el programa que queremos saber si funciona bien o no.
- ▶ **Test Input** (o dato de prueba): Es una asignación concreta de valores a los parámetros de entrada para ejecutar el programa bajo test.
- ▶ **Test Case:** Caso de Test (o caso de prueba). Es un programa que ejecuta el programa bajo test usando un dato de test, y chequea (automáticamente) si se cumple la condición de aceptación sobre la salida del programa bajo test.
- ▶ **Test Suite:** Es un conjunto de casos de Test (o de conjunto de casos de prueba).

16

## Hagamos Testing

- ▶ ¿Cuál es el programa de test?
  - ▶ Es la implementación de una **especificación**.
- ▶ ¿Entre qué datos de prueba puedo elegir?
  - ▶ Aquellos que cumplen la **precondición (requieres)** en la **especificación**
- ▶ ¿Qué condición de aceptación tengo que checar?
  - ▶ La condición que me indica la **postcondición (aseguras)** en la **especificación**.
- ▶ ¿Qué pasa si el dato de prueba no satisface la precondición de la especificación?
  - ▶ Entonces no tenemos ninguna condición de aceptación

17

## Hagamos Testing

¿Cómo testeamos un programa que resuelva el siguiente problema?  
problema  $valorAbsoluto(n : \mathbb{Z}) : \mathbb{Z}\{$

**requiere:** { *True* }  
    **asegura:** { *res* =  $\|n\|$  }  
}

- ▶ Probar *valorAbsoluto* con 0, chequear que *result*=0
- ▶ Probar *valorAbsoluto* con -1, chequear que *result*=1
- ▶ Probar *valorAbsoluto* con 1, chequear que *result*=1
- ▶ Probar *valorAbsoluto* con -2, chequear que *result*=2
- ▶ Probar *valorAbsoluto* con 2, chequear que *result*=2
- ▶ ...etc.
- ▶ ¿Cuántas entradas tengo que probar?

18

## Probando (Testeando) programas

- ▶ Si los enteros se representan con 32 bits, necesitaríamos probar  $2^{32}$  datos de test.
- ▶ Necesito escribir un test suite de 4,294,967,296 test cases.
- ▶ Incluso si lo escribo automáticamente, cada test tarda 1 milisegundo, necesitaríamos 1193,04 horas (49 días) para ejecutar el test suite.
- ▶ Cuanto más complicada la entrada (ej: secuencias), más tiempo lleva hacer testing.
- ▶ La mayoría de las veces, el testing exhaustivo **no es práctico**.

19

## Limitaciones del testing

- ▶ Al no ser exhaustivo, el testing NO puede probar (demostrar) que el software funciona correctamente.

*"El testing puede demostrar la presencia de errores nunca su ausencia"* (Dijkstra)



- ▶ Una de las mayores dificultades es encontrar un conjunto de tests adecuado:
  - ▶ **Suficientemente grande** para abarcar el dominio y maximizar la probabilidad de encontrar errores.
  - ▶ **Suficientemente pequeño** para poder ejecutar el proceso con cada elemento del conjunto y minimizar el costo del testing.

20

## ¿Con qué datos probar?

- ▶ **Intuición:** hay inputs que son “parecidos entre sí” (por el tratamiento que reciben)
- ▶ Entonces probar el programa con uno de estos inputs, ¿equivalearía a probarlo con cualquier otro de estos parecidos entre sí?
- ▶ Esto es la base de la mayor parte de las técnicas
- ▶ ¿Cómo definimos cuándo dos inputs son “parecidos”?
  - ▶ Si únicamente disponemos de la especificación, nos valemos de nuestra *experiencia*

21

## Hagamos Testing

¿Cómo testeamos un programa que resuelva el siguiente problema?  
problema `valorAbsoluto(inn : Z) : Z{`

```
    requiere: {True}
    asegura: {res = ||n||}
}
```

Ejemplo:

- ▶ Probar `valorAbsoluto` con 0, chequear que `result=0`
- ▶ Probar `valorAbsoluto` con un valor negativo `x`, chequear que `result=-x`
- ▶ Probar `valorAbsoluto` con un valor positivo `x`, chequear que `result=x`

22

## Ejemplo: `valorAbsoluto`

- ▶ Programa a testear:
  - ▶ `valorAbsoluto :: Int -> Int`
- ▶ Test Suite:
  - ▶ Test Case #1 (cero):
    - ▶ Entrada: (`x = 0`)
    - ▶ Salida Esperada: `result = 0`
  - ▶ Test Case #2 (positivos):
    - ▶ Entrada: (`x = 1`)
    - ▶ Salida Esperada: `result = 1`
  - ▶ Test Case #3 (negativos):
    - ▶ Entrada: (`x = -5`)
    - ▶ Salida Esperada: `result = 5`

23

## Retomando... ¿Qué casos de test elegir?

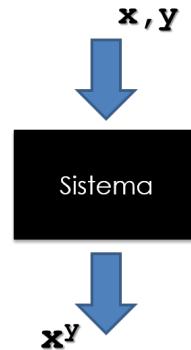
1. No hay un algoritmo que proponga casos tales que encuentren todos los errores en cualquier programa.
2. Ninguna técnica puede ser efectiva para detectar todos los errores en un programa arbitrario
3. En ese contexto, veremos dos tipos de criterios para seleccionar datos de test:
  - ▶ **Test de Caja Negra:** los casos de test se generan analizando la especificación sin considerar la implementación.
  - ▶ **Test de Caja Blanca:** los casos de test se generan analizando la implementación para determinar los casos de test.

24

## Criterios de caja negra o funcionales

- ▶ Los datos de test se derivan a partir de la descripción del programa sin conocer su implementación.

```
problema fastexp(x : Z, y : Z) : Z{  
    requiere: {(0 ≤ x ∧ 0 ≤ y)}  
    asegura: {res = xy}  
}
```

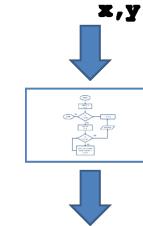


25

## Criterios de caja blanca o estructurales

- ▶ Los datos de test se derivan a partir de la estructura interna del programa.

```
def fastexp(x: int, y: int) → int:  
    z: int = 1  
    while(y != 0):  
        if(esImpar(y)):  
            z = z * x  
            y = y - 1  
  
        x = x * x  
        y = y / 2  
  
    return z
```



¿Qué pasa si y es potencia de 2?  
¿Qué pasa si y = 2n - 1?

## ¿Se puede automatizar el testing?

El diseño de casos de test, no sólo permite organizar u optimizar el trabajo de la persona que ejecutará los casos buscando fallas: existen herramientas que permiten *programar* estos casos de pruebas.

- ▶ Vimos que el nivel más básico del testing se denomina TEST UNITARIO
- ▶ La gran mayoría de los lenguajes de programación tienen herramientas que permiten programar casos de prueba.
- ▶ En el caso de Haskell: HUnit

27

## HUnit

```
import Test.HUnit  
  
— La función que queremos probar  
valorAbsoluto :: Int → Int  
valorAbsoluto x | x ≥ 0 = div 1 x  
               | otherwise = -x  
  
— Las pruebas unitarias  
testSuiteValorAbs = test [  
    "casoPositivo" ~: (valorAbsoluto 1) ~?= 1 ,  
    "casoNegativo" ~: (valorAbsoluto (-5)) ~?= 5 ,  
    "casoCero" ~: (valorAbsoluto 0) ~?= 0  
]  
  
— Corre todas las pruebas  
correrTests = runTestTT testSuiteValorAbs
```

28

## Método de Partición de Categorías

- ▶ Consiste en una técnica que permite generar casos de prueba de una manera metódica.
- ▶ Es aplicable a especificaciones formales, semiformales e inclusive, informales.

El método se puede resumir en los siguientes pasos:

1. Listar todos los problemas que queremos testear
2. Elegir uno en particular
3. Identificar sus **parámetros** o las relaciones entre ellos que condicionan su comportamiento. Los llamaremos genéricamente **factores**.
4. Determinar las características relevantes (categorías) de cada factor.
5. Determinar elecciones (choices) para cada característica de cada factor.
6. Clasificar las elecciones: errores, únicos, restricciones, etc
7. Armado de casos, combinando las distintas elecciones determinadas para cada categoría, y detallando el resultado esperado en cada caso.
8. Volver al paso 2 hasta completar todas las unidades funcionales.

29

## Método de Partición de Categorías

Paso 2: Elegir una unidad funcional

Lo ideal es llegar a testear todas las unidades funcionales. Un buen criterio, es empezar por aquellas que *son utilizadas* por otras.

En este caso:

- ▶ **esPermutacion**
- ▶ **cantidadDeApariciones**

31

## Método de Partición de Categorías

Paso 1: Descomponer la solución informática en unidades funcionales

Consiste en enumerar todas las operaciones, funciones, funcionalidades, problemas que se probarán.

```
problema esPermutacion(s1,s2 : seq(T)) : Bool {  
    asegura: {res = true  $\leftrightarrow$   $(\forall e : T)(cantidadDeApariciones(s1,e) = cantidadDeApariciones(s2,e))$ }  
}
```

```
problema cantidadDeApariciones(s : seq(T), e : T) : Z {  
    requiere: {e  $\in$  s}  
    requiere: {|s| > 0}  
    asegura: {res =  $\sum_{i=0}^{|s|-1} (\text{if } s[i] = e \text{ then 1 else 0 fi})$ }  
}
```

En este caso:

- ▶ **esPermutacion**
- ▶ **cantidadDeApariciones**

30

## Método de Partición de Categorías

Paso 3: Identificar factores

Esto pueden ser los parámetros del problema a testear (si el sistema es más complejo... podrían ser otros factores).

```
problema cantidadDeApariciones(s : seq(T), e : T) : Z {  
    requiere: {|s| > 0}  
    asegura: {res =  $\sum_{i=0}^{|s|-1} (\text{if } s[i] = e \text{ then 1 else 0 fi})$ }  
}
```

En este caso:

- ▶ Tomamos  $T$  como  $\mathbb{Z}$  (porque vamos a buscar datos concretos para probar)
- ▶  $s : \text{seq}(\mathbb{Z})$
- ▶  $e : \mathbb{Z}$

32

## Método de Partición de Categorías

### Paso 4: Determinar categorías

Las categorías son distintas características de cada factor, o características que relacionan diferentes factores, y que tienen influencia en los resultados. Son el resultado del análisis de toda la información disponible sobre la funcionalidad a testear.

En nuestro ejemplo, para cada parámetro podemos determinar las siguientes características:

- ▶  $s : \text{seq}(\mathbb{Z})$ 
  - ▶ ¿Tiene elementos?
- ▶  $e : \mathbb{Z}$ 
  - ▶ En este caso, para  $e$  no se distingue ninguna característica interesante
- ▶ Relación entre  $s$  y  $e$  (esta relación puede ser interesante)
  - ▶ ¿Pertenece  $e$  a  $s$ ?

33

## Método de Partición de Categorías

### Paso 5: Determinar elecciones

Se trata de buscar los conjuntos de valores donde se espera un comportamiento similar. Se basa en las especificaciones, la experiencia, el conocimiento de errores.

En nuestro ejemplo, para cada categoría, determinamos sus elecciones o choices:

- ▶  $s : \text{seq}(\mathbb{Z})$ 
  - ▶ ¿Tiene elementos?
    - ▶ Si
    - ▶ No
- ▶  $e : \mathbb{Z}$ 
  - ▶ Relación entre  $s$  y  $e$ 
    - ▶ ¿Pertenece  $e$  a  $s$ ?
      - ▶ Si
      - ▶ No

34

## Método de Partición de Categorías

### Paso 6: Clasificar las elecciones

Se trata de identificar algunas propiedades o restricciones de las elecciones en el marco de la unidad funcional.

Las clasificaciones más comunes son:

- ▶ Error: Se clasificarán como error aquellas elecciones que por sí mismas determinen que como resultado de la ejecución el sistema debe detectar un error o que no está definido su comportamiento.
- ▶ Otros posibles valores: Único, Restricción, etc. (más adelante ampliaremos).

En nuestro ejemplo, para cada elecciones o choices, analizamos si debemos clasificarlo especialmente:

- ▶  $s : \text{seq}(\mathbb{Z})$ 
  - ▶ ¿Tiene elementos?
    - ▶ Si
    - ▶ No [ERROR]
- ▶  $e : \mathbb{Z}$
- ▶ Relación entre  $s$  y  $e$ 
  - ▶ ¿Pertenece  $e$  a  $s$ ?
    - ▶ Si
    - ▶ No

35

## Método de Partición de Categorías

### Paso 7: Armar los casos de test

Finalmente, se combinarán las distintas elecciones de las categorías consideradas generando los distintos casos de test. Cada combinación es un caso de test, el cual deberá tener identificado con claridad su resultado esperado.

En nuestro ejemplo, deberemos combinar:

- ▶  $\{s \text{ tiene elementos?}\}$ : Si
- ▶  $\{s \text{ tiene elementos?}\}$ : No
- ▶  $\{e \text{ pertenece a } s\}$ : Si
- ▶  $\{e \text{ pertenece a } s\}$ : No

Tenemos 4 elecciones posibles a combinar

- ▶ ¿Nos interesan todas las combinaciones?
- ▶ ¿Cuántos casos de test tenemos?
- ▶ Las elecciones marcadas como ERROR y UNICO, suelen no ser combinables (recortando así la cantidad de combinaciones a realizar).
- ▶ Las elecciones RESTRICCION, condicionan las combinaciones posibles.
- ▶ Tip: estas elecciones son las primeras a considerar en el armado de casos.

36

## Método de Partición de Categorías

### Paso 7: Armar los casos de test

En este caso, sólo nos quedan 3 casos interesantes (nos ahorraremos 1)

- ▶ Por cada caso, debemos describir su resultado esperado: es importante indicar si el resultado será un posible resultado correcto u esperable o un error o comportamiento indefinido.
- ▶ Recordar que los casos de prueba definido serán una herramienta que eventualmente otra persona pueda ejecutar los test: eligiendo datos concretos y comparando el resultado obtenido con el esperado.

| Característica                          | ¿s tiene elementos? | ¿e pertenece a s? | Resultado esperado                                        | Comentario                                           |
|-----------------------------------------|---------------------|-------------------|-----------------------------------------------------------|------------------------------------------------------|
| Caso 1: S sin elementos                 | No                  | -                 | ERROR: no está especificado que sucede en este caso.      | Como la elección No es un ERROR, no importa el valor |
| Caso 2: E no pertenece                  | Si                  | No                | OK: 0                                                     |                                                      |
| Caso 3: S tiene elementos y e Pertenece | Si                  | Si                | OK: el resultado obtenido debe ser igual a la cantidad de |                                                      |

- ▶ La tabla es una representación gráfica y práctica de los casos.
- ▶ Suele ocurrir, que las primeras columnas son siempre de aquellas elecciones que tienen errores, únicos o restricciones entre sus posibles valores: porque descartan casos hacia la derecha.

37

## Método de Partición de Categorías

### Paso 6: Clasificar las elecciones - Volviendo un paso atrás

Se trata de identificar algunas propiedades o restricciones de las elecciones en el marco de la unidad funcional.

Las clasificaciones más comunes son:

- ▶ Único: Son aquellas elecciones que no necesitan combinarse con ninguna otra elección para determinar el resultado esperado.

```
problema siElPrimeroEsCinco(x : Z, y : Z) : Z {  
    asegura: {x = 5 → res = 5}  
    asegura: {x ≠ 5 → res = x + y}  
}
```

- ▶ Factor: x
  - ▶ Característica: valor
    - ▶ Elección: Igual a 5 [ÚNICO]
    - ▶ Elección: Distinto que 5
- ▶ Factor: y

38

## Método de Partición de Categorías

### Paso 6: Clasificar las elecciones - Volviendo un paso atrás

Se trata de identificar algunas propiedades o restricciones de las elecciones en el marco de la unidad funcional.

Las clasificaciones más comunes son:

- ▶ Único: Son aquellas elecciones que no necesitan combinarse con ninguna otra elección para determinar el resultado esperado.

```
problema siElPrimeroEsCincoHaceOtraCosa(x : Z, y : Z, z : Z) : Z {  
    requiere: {x = 5 → z ≠ 0}  
    asegura: {x = 5 → res = x + y/z}  
    asegura: {x ≠ 5 → res = x + y}  
}
```

- ▶ Factor: x
  - ▶ Característica: valor
    - ▶ Elección: Igual a 5 [RESTRICCIÓN]
    - ▶ Elección: Distinto que 5
- ▶ Factor: y
- ▶ Factor: z
  - ▶ Característica: valor
    - ▶ Elección: Igual a 0
    - ▶ Elección: Distinto de 0 Sólo si x ≠ 5 por RESTRICCIÓN

39

## Método de Partición de Categorías

### Si hay otra funcionalidad a testear → Paso 2: Elegir una unidad funcional

Si probamos `esPermutacion` luego de haber testeado `cantidadDeApariciones`, podemos asumir que `cantidadDeApariciones` funciona correctamente.

En este caso:

- ▶ `esPermutacion`
- ▶ `cantidadDeApariciones`

Y repetimos el proceso

40

## HUnit

```
import Test.HUnit

— La función que queremos probar
cantidadDeApariciones :: [Int] → Int → Int
cantidadDeApariciones [] _ = 0
cantidadDeApariciones (x:xs) e
| x == e = 1 + cantidadDeApariciones xs e
| otherwise = cantidadDeApariciones xs e

— Las pruebas unitarias
testSuiteCantidadApariciones = test [
    "eNoPertenece" ~: (cantidadDeApariciones [1,2,3] 4) ~?= 0,
    "ePertenece" ~: (cantidadDeApariciones [1,2,(-2),4,(-2),1,(-2)] (-2)) ~?= 3
]

— Correr todas las pruebas
correrTests = runTestTT testSuiteCantidadApariciones
```

41

## Otro ejemplo

Diseñar los casos de test de caja negra utilizando el método de partición por categorías para el siguiente problema:

```
problema multiplosDeN(n : ℤ, s : seq(ℤ)) : seq(ℤ) {
    requiere: {No hay elementos repetidos en s}
    asegura: {res contiene los elementos de s múltiplos de n, respetando el orden }
}
```

42

## Método de Partición de Categorías

Paso 1: Descomponer la solución informática en unidades funcionales

Consiste en enumerar todas las operaciones, funciones, funcionalidades, problemas que se probarán.

**En nuestro caso, este paso ya está listo:**

```
problema multiplosDeN(n : ℤ, s : seq(ℤ)) : seq(ℤ) {
    requiere: {No hay elementos repetidos en s}
    asegura: {res contiene los elementos de s múltiplos de n, respetando el orden }
}
```

43

## Método de Partición de Categorías

Paso 2: Elegir una unidad funcional

**Este paso también ya lo tenemos listo:**

```
problema multiplosDeN(n : ℤ, s : seq(ℤ)) : seq(ℤ) {
    requiere: {No hay elementos repetidos en s}
    asegura: {res contiene los elementos de s múltiplos de n, respetando el orden }
}
```

44

## Método de Partición de Categorías

### Paso 3: Identificar factores

Estos son los parámetros del problema a testear.

```
problema multiplosDeN(n : Z, s : seq(Z)) : seq(Z) {  
    requiere: {No hay elementos repetidos en s}
```

```
asegura: {res contiene los elementos de s múltiplos de n, respetando el orden }  
}
```

En este caso:

- ▶  $n : \mathbb{Z}$
- ▶  $s : \text{seq}(\mathbb{Z})$

45

## Método de Partición de Categorías

### Paso 5: Determinar elecciones

Se trata de buscar los conjuntos de valores donde se espera un comportamiento similar. Debería ser una partición sin dejar valores afuera.

En nuestro ejemplo, para cada categoría, determinamos sus elecciones o choices:

- ▶  $n : \mathbb{Z}$ 
  - ▶ valor
    - ▶  $< 0$
    - ▶  $= 0$
    - ▶  $> 0$
- ▶  $s : \text{seq}(\mathbb{Z})$ 
  - ▶ ¿Tiene elementos? ¿Tiene elementos repetidos?
    - ▶ No tiene elementos
    - ▶ Sí tiene elementos, pero no repetidos
    - ▶ Sí tiene elementos repetidos
- ▶ Relación entre  $n$  y  $s$ 
  - ▶ Cantidad de múltiplos de  $n$  en  $s$ 
    - ▶ 0
    - ▶ 1
    - ▶  $> 1$

47

## Método de Partición de Categorías

### Paso 4: Determinar categorías

Las categorías son distintas características de cada factor, o características que relacionan diferentes factores, y que tienen influencia en los resultados. Son el resultado del análisis de toda la información disponible sobre la funcionalidad a testear.

En nuestro ejemplo, para cada parámetro podemos determinar las siguientes características:

- ▶  $n : \mathbb{Z}$ 
  - ▶ valor
- ▶  $s : \text{seq}(\mathbb{Z})$ 
  - ▶ ¿Tiene elementos? ¿Tiene elementos repetidos?
- ▶ Relación entre  $n$  y  $s$ 
  - ▶ Cantidad de múltiplos de  $n$  en  $s$

46

## Método de Partición de Categorías

### Paso 6: Clasificar las elecciones

Se trata de identificar algunas propiedades o restricciones de las elecciones en el marco de la unidad funcional.

Las clasificaciones más comunes son:

- ▶ **Error:** Se clasificarán como error aquellas elecciones que por sí mismas determinen que como resultado de la ejecución el sistema debe detectar un error o que no está definido su comportamiento.
- ▶ **Único:** Nos libra de realizar todas las combinaciones con esta elección
- ▶ **Restricción:** Nos permite indicar una condición que se debe cumplir para combinar con esta elección

48

## Método de Partición de Categorías

### Paso 6: Clasificar las elecciones

- ▶ ¿Tenemos casos de ERROR?

Sí. El requerimiento del problema exige que  $s$  no tenga elementos repetidos.

- ▶ ¿Tenemos casos de ÚNICO?

Sí. Cuando la lista  $s$  es vacía.

- ▶ ¿Tenemos casos de RESTRICCIÓN?

Sí. Cuando  $\text{valor}$  es 0 no puede haber más de un múltiplo.

- ▶ ¿Nos interesan todas las combinaciones?

No.

49

## Método de Partición de Categorías

### Paso 7: Armar los casos de test

- ▶ Por cada caso, debemos describir su **resultado esperado**: es importante indicar si el resultado será un posible resultado correcto u esperable o un error o comportamiento indefinido.
- ▶ Los casos de prueba definidos serán una herramienta para que eventualmente otra persona pueda ejecutar los test: eligiendo datos concretos y comparando el resultado obtenido con el esperado.

51

## Método de Partición de Categorías

### Paso 6: Clasificar las elecciones

- ▶  $n : \mathbb{Z}$

- ▶ valor

- ▶  $< 0$
- ▶  $= 0$  [RESTRICCIÓN]
- ▶  $> 0$

- ▶  $s : \text{seq}(\mathbb{Z})$

- ▶ ¿Tiene elementos? ¿Tiene elementos repetidos?

- ▶ No tiene elementos [ÚNICO]
- ▶ Sí tiene elementos, pero no repetidos
- ▶ Sí tiene elementos repetidos [ERROR]

- ▶ Relación entre  $n$  y  $s$

- ▶ Cantidad de múltiplos de  $n$  en  $s$

- ▶ 0
- ▶ 1
- ▶  $> 1$  Sólo si  $n \neq 0$  por RESTRICCIÓN

¿Cuántos casos nos quedaron? 1 (error) + 1 (único) +  $1*1*2$  (restricción)  
 $+ 2*1*3 = 10$  casos

50

## Método de Partición de Categorías

### Paso 7: Armar los casos de test

Analizamos cada caso Caso 1:  $s$  tiene elementos repetidos Caso 2:  $s$  vacía Caso 3:  $\text{valor} = 0$ , cantidad de múltiplos = 0 Caso 4:  $\text{valor} = 0$ , cantidad de múltiplos = 1 Caso 5:  $\text{valor} < 0$ , cantidad de múltiplos = 0 Caso 6:  $\text{valor} < 0$ , cantidad de múltiplos = 1 Caso 7:  $\text{valor} < 0$ , cantidad de múltiplos > 1 Caso 8:  $\text{valor} > 0$ , cantidad de múltiplos = 0 Caso 9:  $\text{valor} < 0$ , cantidad de múltiplos = 1 Caso 10:  $\text{valor} < 0$ , cantidad de múltiplos > 1

| caso | descripción         | ¿tiene elem?<br>¿repetidos? | valor | cant. de<br>múlt | resultado<br>esperado |
|------|---------------------|-----------------------------|-------|------------------|-----------------------|
| 1    | repetidos           | sí (repe)                   | -     | -                | no especificado       |
| 2    | lista vacía         | no                          | -     | -                | lista vacía           |
| 3    | valor 0, múlt 0     | sí                          | 0     | 0                | lista vacía           |
| 4    | valor 0, múlt 1     | sí                          | 0     | 1                | [0]                   |
| 5    | valor < 0, múlt 0   | sí                          | < 0   | 0                | lista vacía           |
| 6    | valor < 0, múlt 1   | sí                          | < 0   | 1                | lista con el múlt     |
| 7    | valor < 0, múlt > 1 | sí                          | < 0   | > 1              | lista con los múlt    |
| 8    | valor > 0, múlt 0   | sí                          | > 0   | 0                | lista vacía           |
| 9    | valor > 0, múlt 1   | sí                          | > 0   | 1                | lista con el múlt     |
| 10   | valor > 0, múlt > 1 | sí                          | > 0   | > 1              | lista con los múlt    |

## HUnit

```
module MultiplosDeN where

— La función que queremos probar
multiplosDeN :: Int → [Int] → [Int]
multiplosDeN _ [] = []
multiplosDeN n (x:xs)
| n == 0 && x == 0 = [0]
| n /= 0 && mod x n == 0 = x : pasoRecursivo
| otherwise = pasoRecursivo
where pasoRecursivo = multiplosDeN n xs
```

53

## HUnit

```
import MultiplosDeN
import Test.HUnit

— Las pruebas unitarias
testSuiteMultiplosDeN = test [
    "lista vacia" ~: (multiplosDeN 4 []) ~?= [],
    "valor 0, mult 0" ~: (multiplosDeN 0 [1,3]) ~?= [],
    "valor 0, mult 1" ~: (multiplosDeN 0 [-1,0,9]) ~?= [0],
    "valor < 0, mult 0" ~: (multiplosDeN (-3) [20,13,-4]) ~?= [],
    "valor < 0, mult 1" ~: (multiplosDeN (-8) [9,-16,7]) ~?= [-16],
    "valor < 0, mult > 1" ~: (multiplosDeN (-7) [0,-14,15]) ~?= [0,-14],
    "valor > 0, mult 0" ~: (multiplosDeN 5 [4,-7,9]) ~?= [],
    "valor > 0, mult 1" ~: (multiplosDeN 7 [7,8,-9]) ~?= [7],
    "valor > 0, mult > 1" ~: (multiplosDeN 11 [-22,10,33]) ~?= [-22,33]
]

— Correr todas las pruebas
correrTests = runTestTT testSuiteMultiplosDeN
```

54

## Método de Partición de Categorías

Comentarios finales

- ▶ Este es sólo un método más (de otros tantos que existen) para encarar el problema de generar casos de prueba.
- ▶ No se evaluará su uso de manera rigurosa en la materia:
  - ▶ La intención es que cuenten con alguna herramienta en caso de que se encuentren frente a la situación de no saber cómo testear sus problemas.
  - ▶ No existe una única forma de generar casos de prueba!
  - ▶ Lo importante, es que sus casos de prueba abarquen, en la medida de lo posible, todas las casuísticas posibles con respecto a los parámetros de entrada.

55

# Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Departamento de Computación - FCEyN - UBA

Introducción a la Programación Imperativa

1

## Algoritmos y programas

Repasando, retomando, continuando...

- ▶ Primero aprendimos a especificar problemas.
- ▶ El objetivo luego fue escribir un **algoritmo** que cumpla esa especificación:
  - ▶ Secuencia de pasos que pueden llevarse a cabo mecánicamente.
- ▶ Puede haber varios algoritmos que cumplan una misma especificación.
- ▶ Una vez que se tiene el algoritmo, se escribe el **programa**:
  - ▶ Expresión formal de un algoritmo.
  - ▶ Lenguajes de programación:
    - ▶ Sintaxis definida.
    - ▶ Semántica definida.
    - ▶ Qué hace una computadora cuando recibe ese programa.
    - ▶ Qué especificaciones cumple.
    - ▶ Ejemplos: Haskell, C, C++, C#, Java, Smalltalk, Prolog, etc.

2

## Paradigmas

Repasando, retomando, continuando...

- ▶ Paradigmas de programación:
  - ▶ Formas de pensar un algoritmo que cumpla una especificación.
  - ▶ Cada uno tiene asociado un conjunto de lenguajes.
  - ▶ Nos llevan a encarar la programación según ese paradigma.
- ▶ Próximo paso: Programación imperativa.

3

## Programación imperativa (diferencias con funcional)

- ▶ Los programas no necesariamente son funciones
  - ▶ Ahora pueden *devolver* más de un valor
  - ▶ Hay nuevas formas de pasar argumentos
- ▶ Nuevo concepto de **variables**
  - ▶ Posiciones de memoria
  - ▶ Cambian explícitamente de valor a lo largo de la ejecución de un programa
  - ▶ Pérdida de la transparencia referencial

4

## Programación imperativa (diferencias con funcional)

- ▶ Nueva operación: la asignación
  - ▶ Cambiar el valor de una variable
- ▶ Las funciones no pertenecen a un tipo de datos
- ▶ Distinto mecanismo de repetición
  - ▶ En lugar de la recursión usamos la **iteración**
- ▶ Nuevo tipo de datos: el **arreglo**
  - ▶ Secuencia de valores de un tipo (como las listas)
  - ▶ Longitud prefijada
  - ▶ Acceso directo a una posición (en las listas, hay que acceder primero a las anteriores)
  - ▶ (y también habrá listas, y muchos otros tipos más... como en cualquier lenguaje)

5

## Lenguaje Python

- ▶ Vamos a usarlo para la programación imperativa (también soporta parte del paradigma de objetos, y parte del paradigma funcional)
- ▶ Vamos a usar un subconjunto (como hicimos con Haskell)
  - ▶ No objetos, no memoria dinámica, etc.
  - ▶ Sí vamos a usar la notación de clases, para definir tipos de datos
- ▶ Es un lenguaje interpretado
- ▶ Tiene tipado dinámico:
  - ▶ Una variable puede tomar valores de distintos tipos
  - ▶ *Nosotros lo vamos a pensar con tipado estático (con fines didácticos)*
    - ▶ Declararemos el tipo de cada variable en tiempo de diseño
- ▶ Es fuertemente tipado:
  - ▶ Dado el valor de una variable de un tipo concreto, no se puede usar como si fuera de otro tipo distinto a menos que se haga una conversión explícita de tipos (casting). Es decir, no se permiten violaciones de los tipos de datos.

6

## Programa Python

- ▶ Colección de tipos y funciones.
- ▶ Definición de función:

```
def nombreFunción (parámetros) -> tipoResultado
    bloqueInstrucciones
```
- ▶ Su evaluación consiste en ejecutar una por una las instrucciones del bloque.
- ▶ El orden entre las instrucciones es importante:
  - ▶ Siempre de arriba hacia abajo.

7

## Programa Python

- ▶ Ejemplo
- ```
problema suma2(x : Z, y : Z) : Z{
    asegura: res = x + y
}

def suma2 (x: int, y: int) -> int:
    res: int = x + y
    return res
```

8

## Programa Python

Aclaración por aquello de tipado dinámico versus tipado estático

- ▶ Tipado dinámico: Una variable puede tomar valores de distintos tipos
- ▶ Tipado estático: La comprobación de tipificación se realiza durante la compilación (y no durante la ejecución)

Python es un lenguaje de tipado dinámico, por lo tanto no es necesario declarar los tipos de sus variables.

```
def suma2 (x, y):  
    res = x + y  
    return res
```

- ▶ En la materia lo pediremos con fines didácticos.
- ▶ Existen implementaciones de Python con tipado estático
  - ▶ <https://mypy.readthedocs.io/en/stable/>

```
def suma2 (x: int, y: int) -> int:  
    res: int = x + y  
    return res
```

9

## Instrucciones

- ▶ Asignación
- ▶ Condicional (if ... else ...)
- ▶ Ciclos (while ...)
- ▶ Procedimientos  
(funciones que no devuelven valores pero modifican sus argumentos)
- ▶ Retorno de control (con un valor, return)

11

## Variables en imperativo

- ▶ Nombre asociado a un espacio de memoria
- ▶ La variable puede tomar distintos valores a lo largo de la ejecución
- ▶ En Python se **declaran** dando su nombre (y opcionalmente su tipo)  
`x: int; // x es una variable de tipo int`  
`c: char; // c es una variable de tipo char`
- ▶ Programación imperativa
  - ▶ Conjunto de variables
  - ▶ Instrucciones que van cambiando sus valores
  - ▶ Los valores finales, deberían resolver el problema

10

## La asignación

- ▶ Es la operación fundamental para modificar el valor de una variable.
  - ▶ Sintaxis: `variable = expresión;`
- ▶ Es una operación asimétrica
  - ▶ Lado izquierdo: debe ir una variable u otra expresión que represente una posición de memoria.
  - ▶ Lado derecho: una expresión del mismo tipo que la variable
    - ▶ constante,
    - ▶ variable o
    - ▶ función aplicada a argumentos.
- ▶ Efecto de la asignación:
  1. Se evalúa la expresión de la derecha y se obtiene un valor.
  2. Ese valor se copia en el espacio de memoria de la variable.
  3. **El resto de la memoria no cambia.**

12

## La asignación

- ▶ Ejemplos:

```
x = 0  
y = x  
x = x+x  
x = suma2(z+1,3)  
x = x*x + 2*y + z
```

- ▶ No son asignaciones:

```
3 = x  
doble (x) = y  
8*x = 8
```

13

## La instrucción *return*

- ▶ Termina la ejecución de una función.
- ▶ Retorna el control a su invocador.
- ▶ Devuelve el valor de la expresión como resultado.

```
problema suma2(x : Z, y : Z) : Z{  
    asegura: res = x + y  
}
```

```
def suma2 (x: int, y: int) -> int:  
    res: int = x + y  
    return res  
  
def suma2 (x: int, y: int) -> int:  
    return x + y
```

14

## Transformación de estados

- ▶ Llamamos **estado** de un programa a los valores de todas sus variables en un punto de su ejecución:
  - ▶ Antes de ejecutar la primera instrucción.
  - ▶ Entre dos instrucciones.
  - ▶ Despues de ejecutar la última instrucción.
- ▶ Veremos la ejecución de un programa como una **sucesión de estados**.
- ▶ La asignación es la instrucción que transforma estados.
- ▶ El resto de las instrucciones son de control: modifican el flujo de ejecución es decir, el orden de ejecución de las instrucciones.

15

## Ejemplo de transformación de estados

```
def ejemplo() -> int:  
    x: int = 0  
    x = x + 3  
    x = 2 * x  
    return x
```

Ejemplo de transformación de estados:

```
x = 0  
    //Estado 1 x == 0  
x = x + 3  
    //Estado 2 x == 3  
x = 2 * x  
    //Estado 3 x == 6
```

16

## Ejemplo de transformación de estados

- ▶ Podemos pensar que cada instrucción define un nuevo estado.
- ▶ A cada estado se le puede dar un nombre, que representará el conjunto de valores de las variables entre dos instrucciones de un programa.

```
instrucción
// estado nombre_estado
otra instrucción
```

Luego de nombrar un estado, podemos referirnos al valor de una variable en dicho estado.

```
nombre_variable@nombre_estado
```

17

## Los argumentos de entrada de las funciones

Para indicar que una función recibe argumentos de entrada usamos variables.

Estas variables toman valor cuando el llamador invoca a la función.

En los lenguajes imperativos, en general, existen dos tipos de pasajes de parámetros:

- ▶ **Pasaje por valor:** se crea una copia local de la variable dentro de la función.
- ▶ **Pasaje por referencia:** se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera.

Hay lenguajes de programación imperativa que se toman en serio que los argumentos de entrada son exactamente eso: **de entrada**.

Sin embargo existen otros lenguajes donde es posible escribir programas que reciben un argumento de entrada en una variable, y luego pueden modificar la variable a gusto.

La mayoría manejan ambos conceptos.

19

## Ejecución simbólica

```
def suc(x: int) -> int:
    //estado a;
    x = x + 2
    //estado b
    //vale x == x@a+2;
    «En el estado b, x vale lo que valía en el estado a más 2»
    x = x - 1
    //estado c
    //vale x == x@b-1;
    «En el estado c, x vale lo que valía en el estado b menos 1»
    return x
```

- ▶ De esta manera, mediante la transformación de estados, podremos realizar una ejecución simbólica del programa, declarando cuánto vale cada variable, en cada estado del programa, en función de los valores anteriores.
- ▶ Algunas técnicas de verificación estática utilizan estos recursos.

18

## Valor & Referencia

Nota: el manejo de memoria no es parte del temario de la materia

- ▶ No está dentro del alcance de la materia hablar sobre temas relacionados a manejo de memoria (temas que son abordados más adelante en la carrera).
- ▶ Veamos un modelo **muy simplificado** para entender la diferencia entre valor y referencia.

Pensemos la memoria como una grilla de 25 posiciones y tres variables x, y y z

Memoria					
	1	2	3	4	5
A					
B				5	
C					
D	25				
E			13		

Variables		
Nombre	Valor	Referencia
x	5	B4
y	25	D1
z	13	E3

- ▶ La variable x tiene un valor de 5 y su referencia es B4
- ▶ La variable y tiene un valor de 25 y su referencia es D1
- ▶ La variable z tiene un valor de 13 y su referencia es E3

20

## Pasaje de argumentos en lenguajes de programación

En los lenguajes imperativos, en general, existen dos tipos de pasajes de parámetros:

- ▶ **Pasaje por valor** se crea una copia local de la variable dentro de la función.
- ▶ **Pasaje por referencia** se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera.

[Ver ejemplo](#)

21

## Pasaje de argumentos en lenguajes de programación

### Pasaje por referencia

- ▶ La función no recibe un valor sino que implícitamente recibe una dirección de memoria donde encontrar el argumento.
- ▶ La función puede leer esa posición de memoria pero también puede escribirla.
- ▶ Todas las asignaciones hechas dentro del cuerpo de la función cambian el contenido de la memoria. Así, los argumentos por referencia sirven para dar resultados de salida (o de entrada y salida).
- ▶ La expresión con la que se realiza la invocación debe ser necesariamente una *variable*, porque necesita tener asociada una posición de memoria.
- ▶ Es decir, la expresión con la que se realiza la invocación no puede ser una constante, ni una función aplicada.



23

## Pasaje de argumentos en lenguajes de programación

### Pasaje por valor

- ▶ Coloca en la posición de memoria del argumento de entrada el *valor* de la expresión usada en la invocación.
- ▶ Se llama también pasaje por **por copia**.
- ▶ La expresión original con la que se realizó la invocación queda protegida contra escritura.



22

## Pasaje de argumentos en lenguajes de programación

### Pasaje por referencia

- ▶ La función no recibe un valor sino que implícitamente recibe una dirección de memoria donde encontrar el argumento.
- ▶ La función puede leer esa posición de memoria pero también puede escribirla.
- ▶ Todas las asignaciones hechas dentro del cuerpo de la función cambian el contenido de la memoria. Así, los argumentos por referencia sirven para dar resultados de salida (o de entrada y salida).
- ▶ La expresión con la que se realiza la invocación debe ser necesariamente una *variable*, porque necesita tener asociada una posición de memoria.
- ▶ Es decir, la expresión con la que se realiza la invocación no puede ser una constante, ni una función aplicada.



## Programación imperativa

### Recapitulando

- ▶ Funciones y Procedimientos: ambos ejecutan un grupo de sentencias.
  - ▶ Funciones: devuelve un valor.
  - ▶ Procedimiento: no devuelve un valor.
- ▶ Conceptualmente, existen tres tipos de pasajes de parámetros:
  - ▶ Entrada (**in**): al salir de la función o procedimiento, la variable pasada como parámetro **continuará teniendo su valor original**.
  - ▶ Salida (**out**): al salir de la función o procedimiento, la variable pasada como parámetro **tendrá un nuevo valor asignado dentro de dicha función o procedimiento**. Su valor inicial **no importa ni debería ser leído dentro de la función o procedimiento en cuestión**.
  - ▶ Entrada y salida (**inout**): al salir de la función o procedimiento, la variable pasada como parámetro **tendrá un nuevo valor asignado dentro de dicha función o procedimiento**. Su valor inicial **SI importa dentro de la función o procedimiento en cuestión**.

Veamos como podemos incorporar estas ideas a nuestro lenguaje de especificación.

24

## Especificación de un problema: Extensión

Pasaje de parámetros: in, out e inout - Funciones y Procedimientos

```
problema nombre(parámetros) : tipo de dato del resultado (opcional){  
    requiere etiqueta: { condiciones sobre los parámetros de entrada }  
    modifica: parámetros que se modificarán  
    asegura etiqueta: { condiciones sobre los parámetros de salida }  
    Si x es un parametro inout, x@pre se refiere al valor que tenía x al entrar a la  
    función }  
    ► nombre: nombre que le damos al problema  
        ► será resuelto por una función con ese mismo nombre  
    ► parámetros: lista de parámetros separada por comas, donde cada parámetro  
    contiene:  
        ► Tipo de pasaje (entrada: in, salida: out, entrada y salida: inout)  
        ► Nombre del parámetro  
        ► Tipo de datos del parámetro o una variable de tipo  
    ► tipo de dato del resultado: tipo de dato del resultado del problema (initialmente  
    especificaremos funciones) o una variable de tipo  
        ► En los asegura, podremos referenciar el valor devuelto con el nombre de res  
    ► El tipo de dato del resultado se vuelve opcional, pues ahora podemos especificar  
    programas que no devuelvan resultados, sino que sólo modifiquen sus parámetros.
```

25

## Ejemplos de pasaje de argumentos en Python

```
def duplicar(valor: str, referencia: list):  
    valor *= 2  
    print("Dentro de la función duplicar: str: " + valor)  
    referencia *= 2  
    print("Dentro de la función duplicar: referencia: " + str(referencia))  
  
x: str = "abc"  
y: list = ['a', 'b', 'c']  
print("ANTES: ")  
print("x: " + x)  
print("y: " + str(y))  
duplicar(x, y)  
print("DESPUES: ")  
print("x: " + x)  
print("y: " + str(y))
```

```
ANTES:  
x: abc  
y: ['a', 'b', 'c']  
Dentro de la función duplicar: str: abcabc  
Dentro de la función duplicar: referencia: ['a', 'b', 'c', 'a', 'b', 'c']  
DESPUES:  
x: abc  
y: ['a', 'b', 'c', 'a', 'b', 'c']
```

27

## Pasaje de argumentos en Python

En Python suceden las siguientes situaciones:

- Conceptualmente, el comportamiento va a depender del tipo de datos de las variables:
  - Tipo primitivos (int, char, strings, etc): se pasan por valor.
  - Tipos compuestos y estructuras (listas, etc): se pasan por referencia.
- Aunque técnicamente, sucede lo siguiente:
  - Todos los parámetros son por referencia siempre.
  - Las variables de tipos primitivos, tienen referencias a valores **inmutables** (Ej x: int = 1, la constante 1 nunca cambia... si hacemos x = 3, lo que estamos haciendo es que ahora x referencia al valor 3... pero en IP no profundizaremos).

26

## Ejemplos de pasaje de argumentos en Python

```
problema duplicar(inout x : seq(T)){  
    modifica: x  
    asegura: {x tendrá todos los elementos de x@pre y además, se le  
    concatenará otra copia de x@pre a continuación.}  
    asegura: {x tendrá el doble de longitud que x@pre.}  
}  
  
def duplicar(x: list):  
    x *= 2
```

28

## Ejemplos de pasaje de argumentos en Python

```
problema duplicar(in x : seq(T)) : seq(T){  
    asegura: {resu será una copia de x y además, se le concatenará  
              otra copia de x a continuación.}  
    asegura: {resu tendrá el doble de longitud que x.}  
}  
  
def duplicar(x: list) -> list:  
    y: list = x.copy()  
    y *= 2  
    return y
```

Nota: más adelante veremos el tipo lista, sus operaciones y como recorrer sus elementos.

29

## Ejemplos de pasaje de argumentos en Python

```
problema duplicar(in x : seq(T), out y : seq(T)){  
    asegura: {y será una copia de x y además, se le concatenará  
              otra copia de x a continuación.}  
    asegura: {y tendrá el doble de longitud que x.}  
}  
  
def duplicar(x: list, y: list):  
    y.clear()  
    y += x  
    y *= 2
```

Nota: más adelante veremos el tipo lista, sus operaciones y como recorrer sus elementos.

30

## Variables en imperativo

- ▶ Nombre asociado a un espacio de memoria.
- ▶ La variable puede tomar distintos valores a lo largo de la ejecución del programa.
- ▶ En Python se **declaran** dando su nombre (y opcionalmente su tipo):  
x: int # x es una variable de tipo int.  
c: chr # c es una variable de tipo char.
- ▶ Programación imperativa:
  - ▶ Conjunto de variables.
  - ▶ Instrucciones que van cambiando sus valores.
  - ▶ Los valores finales, deberían resolver el problema.

31

## Programa en imperativo

- ▶ Colección de tipos, funciones y procedimientos.
- ▶ Su evaluación consiste en ejecutar una por una las instrucciones del bloque.
- ▶ El orden entre las instrucciones es importante:
  - ▶ Siempre de arriba hacia abajo.

32

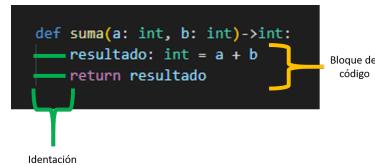
## Instrucciones

- ▶ Asignacion
- ▶ Condicional (if ... else ...)
- ▶ Ciclos (while ...)
- ▶ Procedimientos  
(funciones que no devuelven valores pero modifican sus argumentos)
- ▶ Retorno de control (con un valor, return)

33

## Indentación

- ▶ La indentación es a un lenguaje de programación, lo que la sangría al lenguaje humano escrito.
- ▶ En ciertos lenguajes de programación, la indentación determina la presencia de un bloque de instrucciones (Python es uno de ellos).
- ▶ En otros lenguajes, un bloque puede determinarse de otra manera: por ejemplo encerrandolo entre llaves { }.



35

## Asignación

Semántica de la asignación:

- ▶ Sea  $e$  una expresión cuya evaluación no modifica el estado

```
#estado a  
v = e;  
#vale v == e@a ∧ z1 = z1@a ∧ ⋯ ∧ zk = zk@a
```

donde  $z_1, \dots, z_k$  son todas las variables del programa en cuestión distintas a  $v$  que están definidas hasta ese momento.

- ▶ Las otras variables se supone que no cambian así que, por convención, no hace falta decir nada.
- ▶ Si la expresión  $e$  es la invocación a una función que recibe parámetros por referencia, puede haber más cambios, pero al menos sabemos que:

```
#vale v == e@a
```

está en la *poscondición* o *asegura* de la asignación.

34

## Alcance, ámbito o scope de las variables

- ▶ El alcance de una variable, se refiere al ámbito o espacio donde una variable es reconocida.
- ▶ Una variable sólo será válida dentro del bloque (función/procedimiento) donde fue declarada. Al terminar el bloque, la variable se destruye. Estas variables se denominan **variables locales**.
- ▶ Las variables declaradas fuera de todo bloque son conocidas como **variables globales** y cualquier bloque puede acceder a ella y modificarla.

36

## Variables locales

### Un ejemplo con Python

- x sólo está definida dentro del bloque de instrucciones del procedimiento ejemploLocalScope.
- El intento de acceder a x fuera del procedimiento termina en un error en tiempo de ejecución.
- Aunque el IDE ya nos lo había advertido.

```
def ejemploLocalScope():
    x: int = 19
    print("x: " + str(x))

ejemploLocalScope()
print("x: " + str(x))
```

Variable Local  
Imprimirá: x: 19  
NameError: name 'x' is not defined

```
ejemplo_scope.py > ...
1 def ejemploLocalScope():
2     x: int = 19
3     print("x: " + "x" is not defined Pylance(reportUndefinedVariable)
4
5
6 ejemploLocalScope() View Problem [Alt+F8] Quick Fix... [Ctrl+]
7 print("x: " + str(x))
```

37

## Variables globales

### Un ejemplo con Python

- x sólo está definida de manera global.
- Cualquier bloque puede acceder a ella.

```
def ejemploGlobalScope():
    print("x: " + str(x))

def sumarEnElGlobal():
    global x
    x = x + 120
    print("x: " + str(x))

x: int = 20
ejemploGlobalScope()
sumarEnElGlobal()
ejemploGlobalScope()
print("x: " + str(x))
```

Imprimirá:  
- Primera vez: x: 20  
- Segunda vez: x: 140

En Python, explícitamente hay que referenciar a la variable global para modificarla.

Variable Global

## Variables locales

### Particularidades de Python: bloque del IF

- Los conceptos de variables locales y globales trascienden a los lenguajes.
- De hecho, además de otros tipos de scope, habrá scope a nivel de funciones, procedimientos, clases, packages, etc.
- Y no todos los lenguajes tendrán siempre el mismo comportamiento con respecto a estos temas.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    if(x > 5){
        int y = 6;
    } else {
        int y = 12;
    }
    cout<<y;
    return 0;
}
```

Y tiene un scope local al bloque del IF (Ejemplo en C++)

```
x: int = 10
if(x > 5):
    y: int = 6
else:
    y: int = 12
print("y: " + str(y))

y = 6
```

En Python el bloque mínimo es a nivel función. Los bloques dentro de un IF o un WHILE están dentro del anterior.

```
main.cpp: In function 'int main()':
main.cpp(21,12): error: 'y' was not declared in this scope
21 |     cout<<y;
```

39

## Alcance de Variables en Python

### Python distingue 4 niveles de visibilidad o alcance

- Local: corresponde al ámbito de una función.
- No local o Enclosed: no está en el ámbito local, pero aparece en una función que reside dentro de otra función.
- Global: declarada en el cuerpo principal del programa, fuera de cualquier función.
- Integrado o Built-in: son todas las declaraciones propias de Python (por ejemplo: def, print, etc)

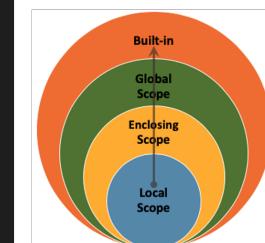
```
def outer():
    enclosed: int = 1

    def inner():
        local: int = 2
        print("INNER: variableGlobal declarada fuera de todo: ", variableGlobal)
        print("INNER: enclosed declarada en outer: ", enclosed)
        print("INNER: local declarada en inner: ", local)

    inner()

    print("OUTER: variableGlobal declarada fuera de todo: ", variableGlobal)
    print("OUTER: enclosed declarada en outer: ", enclosed)
    print("OUTER: local declarada en inner: ", local)

variableGlobal: int = 3
outer()
print("GLOBAL: variableGlobal declarada fuera de todo: ", variableGlobal)
print("GLOBAL: enclosed declarada en outer: ", enclosed)
print("GLOBAL: local declarada en inner: ", local)
```



40

## Alcance de Variables en Python

La referencias también tienen su scope:

- ▶ Al pasar un parámetro por referencia, esta referencia vivirá dentro del scope de la función
- ▶ Analicemos este caso:
  - ▶ En la primer instrucción, *y* toma las referencias de *x* (en este scope, las referencias de *y* se ' pierden')
  - ▶ Al modificar *y*, se está modificando el valor de *x*
  - ▶ Al salir de la función, *y* nunca cambió

```
4 ✓ def duplicar(x: list, y: list):
5     y = x
6     y *= 2
7
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
ANTES:
x: ['a', 'b', 'c']
y: ['d', 'e']
DESPUES:
x: ['a', 'b', 'c', 'a', 'b', 'c']
y: ['d', 'e']
```

41

## Condicionales

```
if (B):
    uno
else:
    dos
```

- ▶ *B* Tiene que ser una expresión booleana. Se llama *guarda*.
- ▶ *uno* y *dos* son bloques de instrucciones.
- ▶ Pensemos el condicional desde la transformación de estados...

42

## Condicionales

Pensemos el condicional desde la transformación de estados

- ▶ Todo el condicional tiene su precondición y su postcondición:  $P_{if}$  y  $Q_{if}$
- ▶ Cada bloque de instrucciones, también tiene sus precondiciones y postcondiciones.

```
# estado  $P_{if}$ 
if (B):
    # estado  $P_{uno}$ 
    uno
    # estado  $Q_{uno}$ 
else:
    # estado  $P_{dos}$ 
    dos
    # estado  $Q_{dos}$ 
# estado  $Q_{if}$ 
# ( $B \wedge$  estado  $Q_{uno}$ )  $\vee$  ( $\neg B \wedge$  estado  $Q_{dos}$ )
# Despues del IF, se cumplió  $B$  y  $Q_{uno}$  o, no se cumplió  $B$  y  $Q_{dos}$ 
```

43

## Condicionales

Un ejemplo con Python

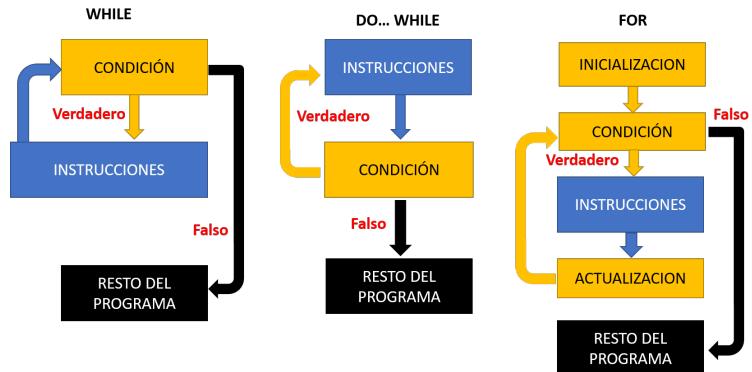
```
def elegirMayor(x: int, y: int) -> int:
    z: int
    print("x = " + str(x) + " | y = " + str(y) )
    if x > y :
        print("x es mayor que y")
        z = x
        print("(Se cumple B) -> z toma el valor de x")
    else:
        print("y es mayor o igual que x")
        z = y
        print("(No se cumple B) -> z toma el valor de y")

    return z
```

44

## Ciclos

- ▶ En los lenguajes imperativos, existen estructuras de control encargadas de repetir un bloque de código mientras se cumpla una condición.
- ▶ Cada repetición suele llamarse iteración.
- ▶ Existen diferentes esquemas de iteración, los más conocidos son:
  - ▶ While, Do While, For

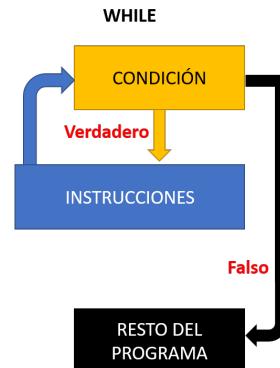


45

## While en Python

### Sintaxis del While

```
while(condición de finalización):  
    ...  
    ...  
    Bloque de Instrucciones...  
    Dentro del while  
    ...  
    ...  
  
Bloque de Instrucciones...  
FUERA del while
```

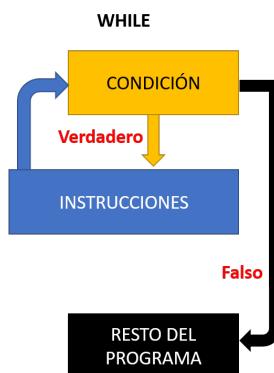


46

## While en Python

### Sintaxis del While

```
while(condición de finalización):  
    ...  
    ...  
    Bloque de Instrucciones...  
    Dentro del while  
    ...  
    ...  
  
Bloque de Instrucciones...  
FUERA del while
```



47

## While en Python

Un programa que muestra por pantalla el número ingresado por el usuario, hasta que el usuario ingresa 0.

```
numero = int(input('Ingresa un número. 0 para terminar: '))  
  
while(numero != 0):  
    print('Usted ingresó: ', numero)  
    numero = int(input('Ingresa un número. 0 para terminar: '))  
  
print('Fin del programa.')
```



- ▶ `input`: espera que el usuario ingrese algo por teclado.
- ▶ `int(input(...))`: convierte en int lo que el usuario ingresó por teclado.

48

## For en Python

### Sintaxis del For

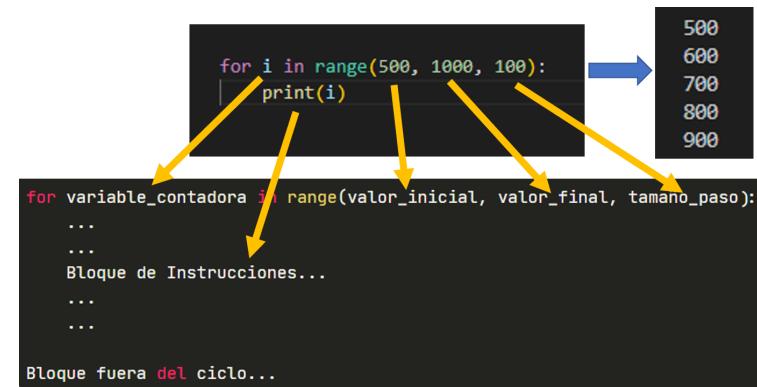
```
for variable_contadora in range(valor_inicial, valor_final, tamaño_paso):  
    ...  
    ...  
    Bloque de Instrucciones...  
    ...  
    ...  
  
Bloque fuera del ciclo...
```



49

## For en Python

¿Qué hace este programa?



50

## Interrumpiendo ciclos: Break

- ▶ La instrucción Break permite romper la ejecución de un ciclo.
- ▶ No fomentamos su uso, sólo mencionamos su existencia (por varios motivos que van más allá del alcance de la materia).
  - ▶ Su uso le quita declaratividad al código.
  - ▶ Desde el punto de vista de analizar la correctitud de un programa, traerá problemas (pero eso ya lo verán más adelante en la carrera).

```
while(True):  
    numero = int(input('Ingresa un número. 0 para terminar: '))  
    print('Usted ingresó: ', numero)  
    if(numero==0):  
        break  
  
print('Fin del programa.')
```

51

## Ciclos y transformación de estados...

```
while(condición de finalización):  
    ...  
    ...  
    Bloque de Instrucciones...  
    Dentro del while  
    ...  
    ...  
  
    Bloque de Instrucciones...  
    FUERA del while
```

- ▶ En un programa imperativo, cada instrucción transforma el estado.
- ▶ Mediante la transformación de estados, podemos hacer una ejecución simbólica del programa.
- ▶ ¿Cómo sería la transformación de estados de un ciclo?
  - ▶ Podemos pensar en el ciclo como una instrucción: con un estado previo y uno posterior
  - ▶ ¿Qué sucede dentro del ciclo? ¿Qué sucede en cada iteración?
- ▶ Más adelante en la carrera, verán cómo manejar estas situaciones.

52

# Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Departamento de Computación - FCEyN - UBA

Programación Imperativa: Arrays y Listas

1

## Arreglos

- ▶ Secuencias de una cantidad fija de valores del mismo tipo.
- ▶ Se declaran con un nombre y un tipo:
  - ▶ Según el lenguaje, además se debe indicar su tamaño (el cual permanece fijo).
  - ▶ Veremos que en Python, los arrays tienen longitud variable.
- ▶ Solamente hay valores en las posiciones válidas (dentro de su tamaño).
- ▶ Una sola variable contiene muchos valores:
  - ▶ A cada valor se lo accede directamente mediante corchetes.
  - ▶ Si  $a$  es un arreglo de 10 elementos,  $a[5]$  devuelve el 6to valor.
  - ▶ El primer elemento es el de índice 0.

3

## Variables en imperativo

Rpasando

- ▶ Nombre asociado a un espacio de memoria.
- ▶ La variable puede tomar distintos valores a lo largo de la ejecución.
- ▶ En Python se **declaran** dando su nombre (y opcionalmente su tipo)  
`x: int; # x es una variable de tipo int.`  
`c: chr; # c es una variable de tipo char.`
- ▶ Programación imperativa:
  - ▶ Conjunto de variables.
  - ▶ Instrucciones que van cambiando sus valores.
  - ▶ Los valores finales, deberían resolver el problema.

2

## Arreglos

- ▶ Supongamos que la variable  $a$  tiene tipo de dato arreglo de int.
- ▶ Podemos ejemplificar que sucede con su referencia en esta simplificación:
  - ▶ La variable  $a$  tiene su referencia en B1.
  - ▶ Como es un arreglo de tamaño 4, tiene asociadas 4 posiciones más de memoria.
  - ▶ Por ejemplo:  $a[2]$  tiene el valor de 7 (el valor que está en B3).
  - ▶ Si tenemos que describir el estado de la variable  $a$ , el mismo es [5,6,7,8].
  - ▶  $a[2]$  no es una variable en sí misma, la variable es  $a$ .

Memoria					
	1	2	3	4	5
A					
B	5	6	7	8	
C					
D					
E					

Variables				
Nombre	Tipo	Tamaño	Valor	Referencia
a	Array de Int	4	[5,6,7,8]	B1

4

## Arreglos y Listas

- ▶ Ambos representan secuencias de elementos de un tipo.
- ▶ Los arreglos suelen tener longitud fija, las listas, no.
- ▶ Los elementos de un arreglo pueden accederse en forma independiente y directa:
  - ▶ Los de la lista se acceden secuencialmente, empezando por la cabeza,
  - ▶ Para acceder al  $i$ -ésimo elemento de una lista, hay que obtener  $i$  veces la cola y luego la cabeza.
  - ▶ Para acceder al  $i$ -ésimo elemento de un arreglo, simplemente se usa el índice.

Memoria					
	1	2	3	4	5
A			7		
B	5				
C					
D		6			8
E					

Variables				
Nombre	Tipo	Tamaño	Valor	Referencia
a	Lista de int		[5,6,7,8]	B1

5

## Arreglos Python

- ▶ `array` es uno de los módulos que permiten utilizar arreglos (otra posibilidad es NumPy).
- ▶ Al crear el arreglo, se indica su tipo y su contenido inicial.

```
from array import *
a: array = array(typecode, [initializers])
```

```
from array import *
a: array = array('i', [10, 20, 30])
```

6

## Arreglos Python

- ▶ Tipos de datos del módulo `array`

Code Type	Python Type	Full Form	Tamaño (en Bytes)
u	unicode character	Python Unicode	2
b	int	Signed Char	1
B	int	Unsigned Char	1
h	int	Signed Short	2
I	int	Signed Long	4
L	int	Unsigned Long	4
q	int	Signed Long Long	8
Q	int	Unsigned Long Long	8
H	int	Unsigned Short	2
f	float	Float	4
d	float	Double	8
i	int	Signed Int	2
l	int	Unsigned Int	2

7

## Arreglos Python

### Operaciones básicas sobre arrays

Sea  $a$  un array:

- $a[i]$  → obtiene el valor del elemento  $i$  de  $a$
- $a[i] = x$  → asigna  $x$  en el elemento  $i$  de  $a$
- $a.append(x)$  → añade  $x$  como nuevo elemento de  $a$
- $a.remove(x)$  → elimina el primer elemento en  $a$  que coincide con  $x$
- $a.index(x)$  → obtiene la posición donde aparece por primera vez el elemento  $x$
- $a.count(x)$  → devuelve la cantidad de apariciones del elemento  $x$
- $a.insert(p,x)$  → inserta el elemento  $x$  delante de la posición  $p$

8

# Listas en Python

En Python, las listas son un tipo de array.

En Python, las listas pueden tener elementos de diferentes tipos de datos.

Al igual que los arrays en Python, tienen tamaño dinámico.

## ¿Cómo se declaran?

- ▶ `variableLista = [] #lista vacia`
- ▶ `otraVariableLista = list() #lista vacia`
- ▶ `otraVariable = [1, 2, True, 'Hola', 5.8]`
- ▶ `unaMas = list([4, 9, False, 'texto'])`

9

# Listas en Python

Operaciones básicas sobre listas

Un ejemplo de una operación que tiene el array y no la lista:

Buffer Info devuelve una tupla con dirección de memoria actual de los arrays y número de elementos (Útil sólo para operaciones de bajo nivel).

```
◆ operaciones_arrays.listas.py > ...
1  # Importar módulo array
2  import array
3
4  # Declarar lista con valores numéricos
5  lista1 = [1, 0, 1, 0, 1, 0, 0]
6
7  # Declarar 'array1' de tipo 'char sin signo' con datos de 'lista1'
8  array1 = array.array('B', lista1)
9
10 print("Buffer info: " + str(array1.buffer_info()))
11 print("Buffer info: " + str(lista1.buffer_info()))
12
13
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Buffer info: (2071285481920, 8)
Traceback (most recent call last):
  File "c:\@Martin\Facultad\intro-programacion\teoricas\ti2 - listas\ejemplos\operaciones_arrays_listas.py", line 11, in <module>
    print("Buffer info: " + str(lista1.buffer_info()))
                                         ^
AttributeError: 'list' object has no attribute 'buffer_info'
```

11

# Listas en Python

Operaciones básicas sobre listas

Las listas y los arrays comparten muchas operaciones.

Sea  $a$  una lista:

- $a[i]$  —> obtiene el valor del elemento  $i$  de  $a$
- $a[i] = x$  —> asigna  $x$  en el elemento  $i$  de  $a$
- $a.append(x)$  —> añade  $x$  como nuevo elemento de  $a$
- $a.remove(x)$  —> elimina el primer elemento en  $a$  que coincide con  $x$
- $a.index(x)$  —> obtiene la posición donde aparece por primera vez el elemento  $x$
- $a.count(x)$  —> devuelve la cantidad de apariciones del elemento  $x$
- $a.insert(p, x)$  —> inserta el elemento  $x$  delante de la posición  $p$

10

# Listas en Python

Como recorrer una lista

Podemos utilizar las distintas estructuras de control de ciclo para recorrer los elementos de una lista utilizando índices:

```
edades: list = [20, 41, 6, 18, 23]

# Recorriendo los índices
for i in range(len(edades)):
    print(edades[i])

# Con while y los índices
indice = 0
while indice < len(edades):
    print(edades[indice])
    indice += 1
```

12

## Listas en Python

Como recorrer una lista

También podremos utilizar el for para recorrer directamente sus elementos:

```
edades: list = [20, 41, 6, 18, 23]

# Recorriendo los elementos
for edad in edades:
    print(edad)
```

13

## Listas en Python

Iterables

En Python, aquellas variables cuyo tipo de dato sea *iterable* pueden ser recorridas con un for.

NOTA (para quienes sabes Python): los conceptos de iterable e iterators están fuera del temario de la materia.

```
edades: list = [20, 41, 6, 18, 23]

# Recorriendo los elementos
for edad in edades:
    print(edad)
```

15

## Listas en Python

Matrices

Podemos pensar una matriz como una lista de listas. Podemos recorrerlas a través de sus índices:

```
ganadores = [['Messi', 'Cristiano', 'Mbappe'], [7, 5, 1]]

# Recorriendo los indices
# i serian las filas
print("++ Con for - i son filas ++")
for i in range(len(ganadores)):
    for j in range(len(ganadores[i])):
        print("ganadores["+str(i)+"]["+str(j)+"] = " + str(ganadores[i][j]))

print("++ Con while y los indices ++")
# Con while y los indices
fila = 0

while fila < len(ganadores):
    columna = 0
    while columna < len(ganadores[fila]):
        print("ganadores["+str(fila)+"]["+str(columna)+"] = " + str(ganadores[fila][columna]))
        columna += 1
    fila += 1
```

14

## Tipos Abstractos de Datos

Un Tipo Abstracto de Datos (TAD) es un modelo que define valores y las operaciones que se pueden realizar sobre ellos.

- ▶ Se denomina abstracto ya que la intención es que quien lo utiliza, no necesita conocer los detalles de la representación interna o bien el cómo están implementadas sus operaciones.

El tipo lista que estuvimos viendo es un TAD:

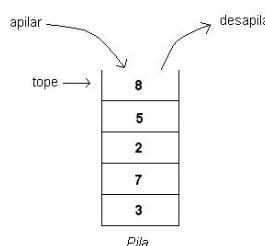
- ▶ Se define como una serie de elementos consecutivos
- ▶ Tiene diferentes operaciones asociadas: append, remove, etc
- ▶ Desconocemos cómo se usa/guarda la información almacenada dentro del tipo

16

## Pila

Una pila es una lista de elementos de la cual se puede extraer el último elemento insertado.

- ▶ También se conocen como listas LIFO (Last In - First Out / el último que entra es el primero que sale)
- ▶ Operaciones básicas
  - ▶ apilar: ingresa un elemento a la pila
  - ▶ desapilar: saca el último elemento insertado
  - ▶ tope: devuelve (sin sacar) el ultimo elemento insertado
  - ▶ vacia: retorna verdadero si está vacía



17

## Pila

Ejemplos de problemas que naturalmente se modelarían con una pila

- ▶ Una pila de platos acumulados en la bacha esperando a ser lavados (no se puede sacar los de abajo, sin antes lavar los últimos apoyados).
- ▶ Una pila de libros o meter y sacar libros de una caja.
- ▶ En las góndolas del supermercado, el fondo del estando es el fondo de la pila, y el tope son los artículos que se pueden tomar facilmente.
- ▶ Ponerse muchas remeras, una arriba de la otra.

18

## Pila

En Python, el tipo lista provee los métodos necesarios para poder usar una lista como una pila

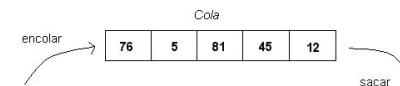
- ▶ Operaciones básicas
  - ▶ apilar: ingresa un elemento a la pila
    - ▶ `append`
  - ▶ desapilar: saca el último elemento insertado
    - ▶ `pop`
  - ▶ tope: devuelve (sin sacar) el ultimo elemento insertado
    - ▶ `a[-1]`
  - ▶ vacia: retorna verdadero si está vacía
    - ▶ `len(a)==0`

19

## Cola

Una cola es una lista de elementos en donde siempre se insertan nuevos elementos al final de la lista y se extraen elementos desde el inicio de la lista.

- ▶ También se conocen como listas FIFO (First In - First Out / el primero que entra es el primero que sale)
- ▶ Operaciones básicas
  - ▶ encolar: ingresa un elemento a la cola
  - ▶ sacar: saca el primer elemento insertado
  - ▶ vacia: retorna verdadero si está vacía



20

## Cola

Ejemplos de problemas que naturalmente se modelarían con una cola

- ▶ La fila en la parada de colectivos
- ▶ La fila de la caja de un supermercado
- ▶ La fila en la cabina de peaje

21

## Cola

En Python, el tipo lista provee los métodos necesarios para poder usar una lista como una cola

- ▶ Operaciones básicas
  - ▶ encolar: ingresa un elemento a la pila
    - ▶ `append`
  - ▶ desencolar: saca el primer elemento insertado
    - ▶ `pop(0)`
  - ▶ vacia: retorna verdadero si está vacía
    - ▶ `len(a)==0`

22

## +TADs & Testing Estructural o Caja Blanca

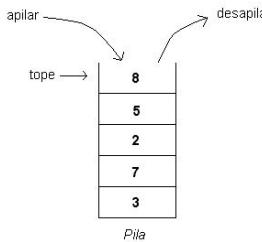
### Introducción a la Programación

1

## Pila

Una pila es una lista de elementos de la cual se puede extraer el último elemento insertado.

- ▶ También se conocen como listas LIFO (Last In - First Out / el último que entra es el primero que sale)
- ▶ Operaciones básicas
  - ▶ apilar: ingresa un elemento a la pila
  - ▶ desapilar: saca el último elemento insertado
  - ▶ tope: devuelve (sin sacar) el ultimo elemento insertado
  - ▶ vacia: retorna verdadero si está vacía



3

## Tipos Abstractos de Datos

### Repasando

Un Tipo Abstracto de Datos (TAD) es un modelo que define valores y las operaciones que se pueden realizar sobre ellos.

- ▶ Se denomina abstracto ya que la intención es que quien lo utiliza, no necesita conocer los detalles de la representación interna o bien el cómo están implementadas sus operaciones.

El tipo lista que estuvimos viendo es un TAD:

- ▶ Se define como una serie de elementos consecutivos
- ▶ Tiene diferentes operaciones asociadas: append, remove, etc
- ▶ Desconocemos cómo se usa/guarda la información almacenada dentro del tipo

2

## Pila

- ▶ En Python, el tipo lista provee los métodos necesarios para poder usar una lista como una pila
- ▶ También, podemos importar el módulo LifoQueue que nos da una implementación de Pila

```
from queue import LifoQueue  
pila = LifoQueue()
```

- ▶ Operaciones implementadas en el tipo:

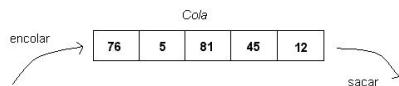
- ▶ apilar: ingresa un elemento a la cola
  - ▶ put
- ▶ desapilar: devuelve y quita el último elemento insertado
  - ▶ get
- ▶ tope: devuelve (sin sacar) el ultimo elemento insertado
  - ▶ No está implementado
- ▶ vacia: retorna verdadero si está vacía
  - ▶ empty

4

## Cola

Una cola es una lista de elementos en donde siempre se insertan nuevos elementos al final de la lista y se extraen elementos desde el inicio de la lista.

- ▶ También se conocen como listas FIFO (First In - First Out / el primero que entra es el primero que sale)
- ▶ Operaciones básicas
  - ▶ encolar: ingresa un elemento a la cola
  - ▶ sacar: saca el primer elemento insertado
  - ▶ vacia: retorna verdadero si está vacía



5

## Cola

- ▶ En Python, el tipo lista provee los métodos necesarios para poder usar una lista como una cola
- ▶ También, podemos importar el módulo Queue que nos da una implementación de Cola

```
from queue import Queue  
cola = Queue()
```

- ▶ Operaciones implementadas en el tipo:

- ▶ encolar: ingresa un elemento a la cola
  - ▶ put
- ▶ desencolar: saca el primer elemento insertado
  - ▶ get
- ▶ vacia: retorna verdadero si está vacía
  - ▶ empty

6

## Diccionario

Un diccionario es una estructura de datos que permite almacenar y organizar pares clave-valor.

- ▶ Las claves deben ser inmutables (como cadenas de texto, números, etc), mientras que los valores pueden ser de cualquier tipo de dato.
- ▶ La clave actúa como un identificador único para acceder a su valor correspondiente.
- ▶ Los diccionarios son mutables, lo que significa que se pueden modificar agregando, eliminando o actualizando elementos.
- ▶ No ordenados: Los elementos dentro de un diccionario no tienen un orden específico. No se garantiza que se mantenga el orden de inserción de los elementos.

```
diccionario = clave1:valor2, clave2:valor2, clave3:valor3
```

- ▶ Operaciones básicas de un diccionario:
  - ▶ Agregar un nuevo par Clave-Valor
  - ▶ Eliminar un elemento
  - ▶ Modificar el valor de un elemento
  - ▶ Verificar si existe una clave guardada
  - ▶ Obtener todas las claves
  - ▶ Obtener todos los elementos

7

## Diccionario

Un diccionario es una estructura de datos que permite almacenar y organizar pares clave-valor.

- ▶ El valor puede ser cualquier tipo de dato, en particular podría ser otro diccionario

```
infoPaisFrancia = {'Capital':'Paris',  
                   'Campeonatos de Mundo':2}  
  
infoPaisArgentina = {'Capital':'Buenos Aires',  
                     'Campeonatos de Mundo':3}  
  
infoPaisChile = {'Capital':'Santiago',  
                 'Campeonatos de Mundo':0}  
  
infoPaises = {'Chile': infoPaisChile ,  
             'Argentina': infoPaisArgentina,  
             'Francia':infoPaisFrancia}
```

8

## Manejo de Archivos

El manejo de archivos, también puede pensarse mediante la abstracción que nos brindan los TADs

- ▶ Necesitamos una operación que nos permita abrir un archivo
- ▶ Necesitamos una operación que nos permita leer sus líneas
- ▶ Necesitamos una operación que nos permita cerrar un archivo

```
# Abrir un archivo en modo lectura
archivo = open("archivo.txt", "r")

# Leer el contenido del archivo
contenido = archivo.read()
print(contenido)

# Cerrar el archivo
archivo.close()
```

9

## ¿Podremos implementar este problema?

```
problema invertirTexto(in archivoOrigen: string, in archivoDestino: string) : {
    requiere: {El archivo nombreArchivo debe existir.}
    asegura: {Se crea un archivo llamado archivoDestino cuyo contenido será el resultado de hacer un reverse en cada una de sus filas}
    asegura: {Si el archivo archivoDestino existia, se borrará todo su contenido anterior}
}
```

11

## Manejo de Archivos

archivo = open("PATH AL ARCHIVO", MODO, ENCODING)

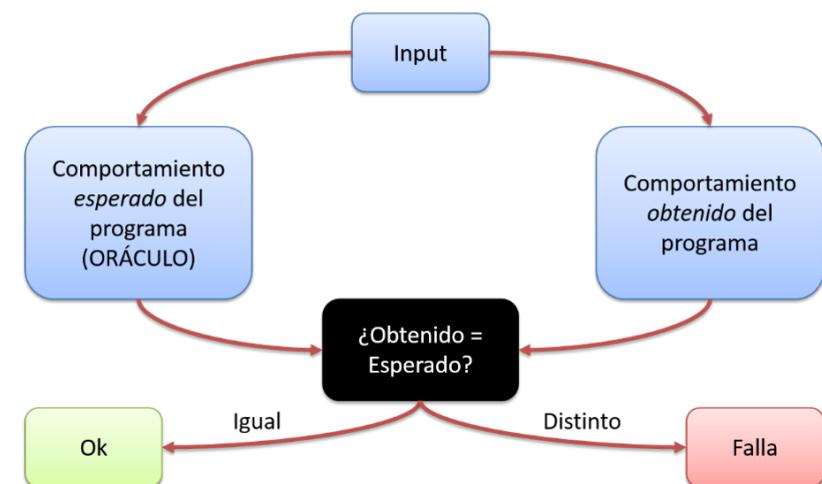
- ▶ Algunos de los modos posibles son: escritura (w), lectura (r), texto (t - es el default)
- ▶ El encoding se refiere a como está codificado el archivo: UTF-8 o ASCII son los más frecuentes.

### Operaciones básicas

- ▶ Lectura de contenido:
  - ▶ read(size): Lee y devuelve una cantidad específica de caracteres o bytes del archivo. Si no se especifica el tamaño, se lee el contenido completo.
  - ▶ readline(): Lee y devuelve la siguiente línea del archivo.
  - ▶ readlines(): Lee todas las líneas del archivo y las devuelve como una lista.
- ▶ Escritura de contenido:
  - ▶ write(texto): Escribe un texto en el archivo en la posición actual del puntero. Si el archivo ya contiene contenido, se sobrescribe.
  - ▶ writelines(lineas): Escribe una lista de líneas en el archivo. Cada línea debe terminar con un salto de línea explícito.

10

## ¿Cómo se hace testing?

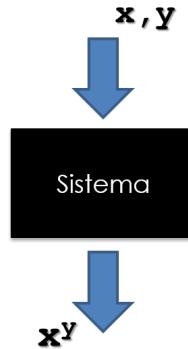


12

## Criterios de caja negra o funcionales

- ▶ Los datos de test se derivan a partir de la descripción del programa sin conocer su implementación.

```
problema fastexp(x : Z, y : Z) : Z{  
    requiere: {(0 < x ∧ 0 ≤ y)}  
    asegura: {res = xy}  
}
```

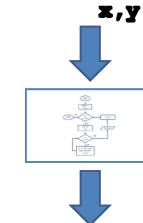


13

## Criterios de caja blanca o estructurales

- ▶ Los datos de test se derivan a partir de la estructura interna del programa.

```
def fastexp(x: int, y: int) → int:  
    z: int = 1  
    while(y != 0):  
        if(esImpar(y)):  
            z = z * x  
            y = y - 1  
  
        x = x * x  
        y = y / 2  
  
    return z
```



¿Qué pasa si y es potencia de 2?

¿Qué pasa si  $y = 2n - 1$ ?

## Criterios de caja blanca o estructurales

Los criterios de caja blanca permiten identificar casos especiales según el flujo de control de la aplicación.

- ▶ Ver que sucede si entra o no en un IF
- ▶ Ver que sucede si entra o no a un ciclo
- ▶ Etc

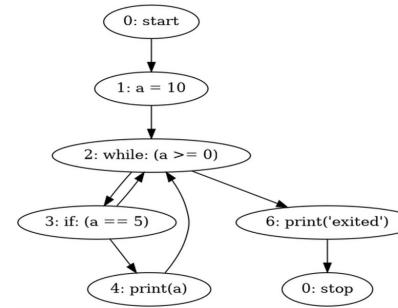
Pero tiene una tremenda dificultad: determinar el resultado esperado de un programa sin una especificación no es para nada trivial.

Por este motivo, el test de caja blanca se suele utilizar como:

- ▶ **Complemento al Test de Caja Negra:** permite encontrar más casos o definir casos más específicos
- ▶ Como **criterio de adecuación** del Test de Caja Negra: brinda herramientas que nos ayudar a determinar cuan bueno o confiable resultaron ser los test suites
- ▶ En este contexto hablaremos de **Criterios de Cubrimiento**

15

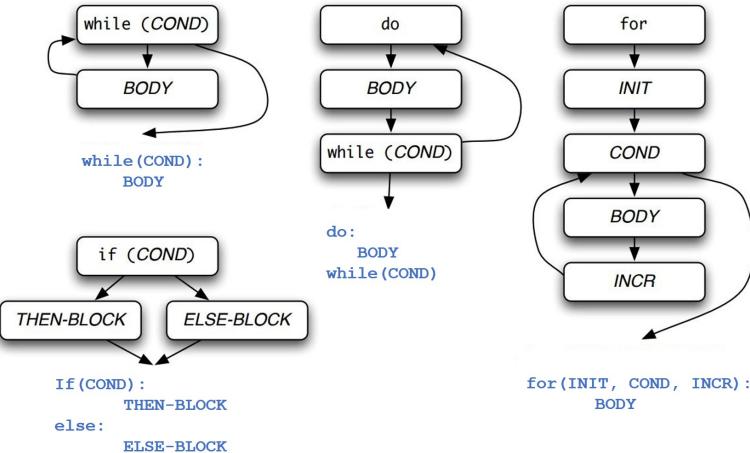
## Control-Flow Graph



- ▶ El control flow graph (CFG) de un programa es sólo una **representación gráfica del programa**.
- ▶ El CFG es independiente de las entradas (su definición es estática)
- ▶ Se usa (entre otras cosas) para definir criterios de adecuación para test suites.
- ▶ Cuanto más *partes* son ejercitadas (cubiertas), mayores las chances de un test de descubrir una falla
- ▶ *partes* pueden ser: nodos, arcos, caminos, decisiones...

16

## Control Flow Patterns



17

## Ejemplo #1

```
problema valorAbsoluto(in x : Z) : Z{
    requiere: {True}
    asegura: {res = ||x||}
}
```

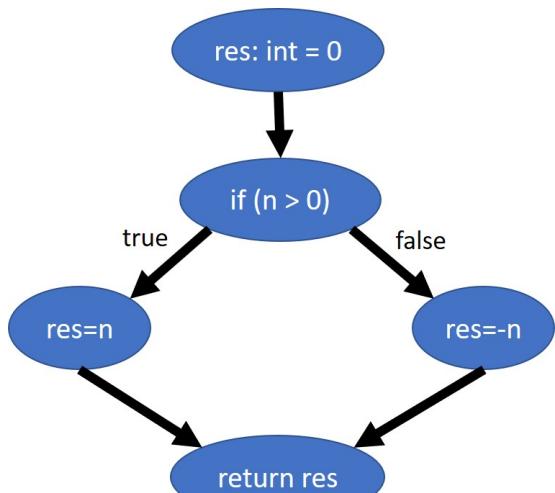
```
def valorAbsoluto(n: int) → int:
    res: int = 0

    if( n > 0 ):
        res = n
    else:
        res = -n

    return res
```

18

## CFG de valorAbsoluto



19

## Ejemplo #2

```
problema sumar(in n : Z) : Z{
    requiere: {n ≥ 0}
    asegura: {res = ∑_{i=1}^n i}
}
```

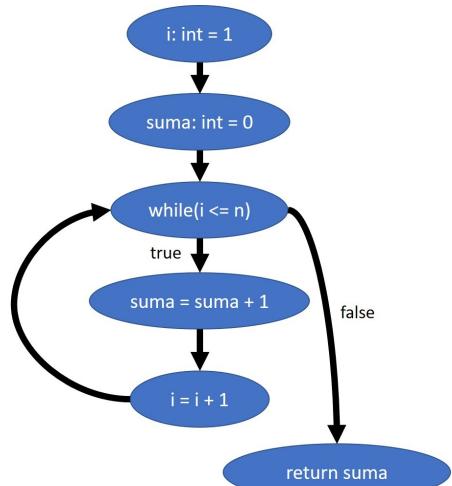
```
def sumar(n: int) → int:
    i:int = 1
    suma:int = 0

    while( i ≤ n ):
        suma = suma + i
        i = i + 1

    return suma
```

20

## CFG de sumar



21

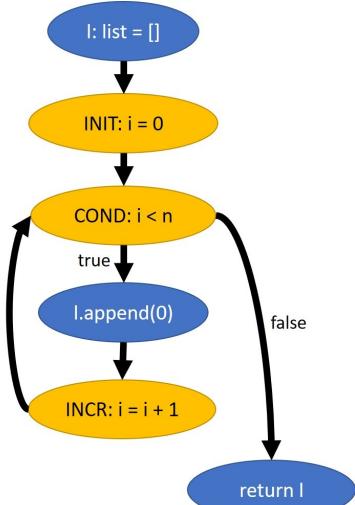
## Ejemplo #3

```
problema crearListaN(in n : Z) : seq(Z){  
    requiere: {n ≥ 0}  
    asegura: {|res| = n ∧ #apariciones(res, 0) = n}  
}
```

```
def crearListN(int n) → list:  
    l: list = []  
  
    for i in range(0, n, 1):  
        l.append(0)  
  
    return l
```

22

## CFG de crearListaN



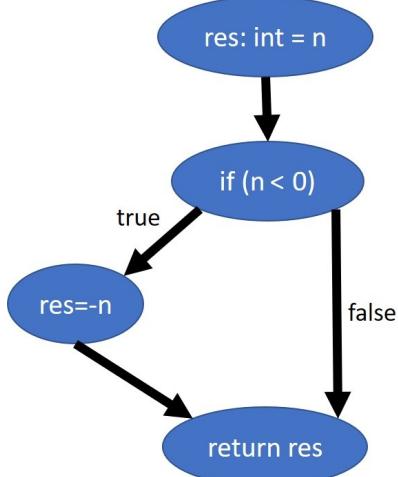
23

## Ejemplo #4

```
def valorAbsoluto(n: int) → int:  
    res: int = n  
  
    if( n < 0 ):  
        res = -n  
  
    return res
```

24

## CFG de valorAbsoluto



25

## Criterios de Adecuación

- ▶ ¿Cómo sabemos que un *test suite* es *suficientemente bueno*?
- ▶ Un criterio de adecuación de test es un predicado que toma un valor de verdad para una tupla  $\langle \text{programa}, \text{test suite} \rangle$
- ▶ Usualmente expresado en forma de una regla del estilo:  
*todas las sentencias deben ser ejecutadas*

26

## Cubrimiento de Sentencias

- ▶ Criterio de Adecuación: cada nodo (sentencia) en el CFG debe ser ejecutado al menos una vez por algún test case.
- ▶ Idea: un defecto en un sentencia sólo puede ser revelado ejecutando el defecto.
- ▶ Cobertura:

$$\frac{\text{cantidad nodos ejercitados}}{\text{cantidad nodos}}$$

27

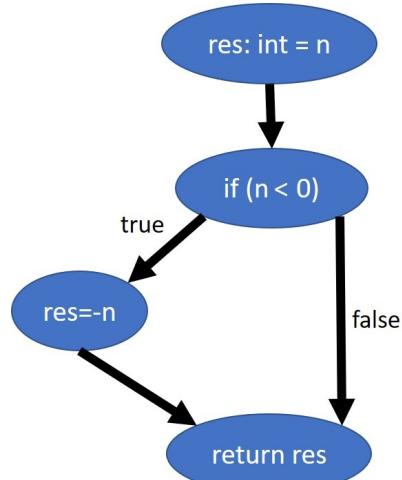
## Cubrimiento de Arcos

- ▶ Criterio de Adecuación: todo arco en el CFG debe ser ejecutado al menos una vez por algún test case.
- ▶ Si recorremos todos los arcos, entonces recorremos todos los nodos. Por lo tanto, el cubrimiento de arcos incluye al cubrimiento de sentencias.
- ▶ Cobertura:  
$$\frac{\text{cantidad arcos ejercitados}}{\text{cantidad arcos}}$$
- ▶ El cubrimiento de sentencias (nodos) no incluye al cubrimiento de arcos. *¿Por qué?*

28

## Cubrimiento de Nodos no incluye cubrimiento de Arcos

Sea el siguiente CFG:



En este ejemplo, puedo construir un test suite que cubra todos los nodos pero que no cubra todos los arcos.

29

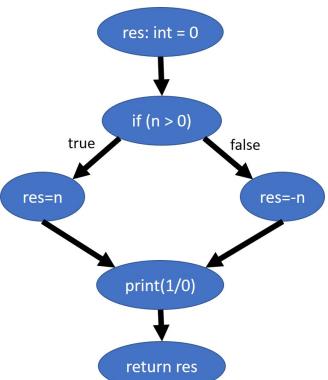
## Cubrimiento de Decisiones (o Branches)

- ▶ Criterio de Adecuación: cada decisión (arco True o arco False) en el CFG debe ser ejecutado.
- ▶ Por cada arco **True** o arco **False**, debe haber al menos un test case que lo ejerza.
- ▶ Cobertura:  
$$\frac{\text{cantidad decisiones ejercitadas}}{\text{cantidad decisiones}}$$
- ▶ El cubrimiento de decisiones **no implica** el cubrimiento de los arcos del CFG. *¿Por qué?*

30

## Cubrimiento de Branches no incluye cubrimiento de Arcos

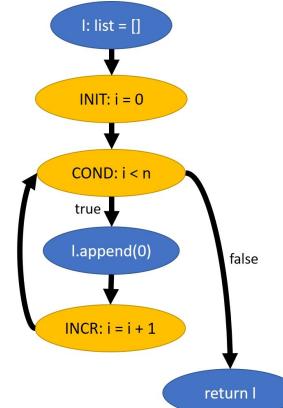
Sea el siguiente CFG:



En este ejemplo, puedo construir un test suite que cubra todos los branches pero que no cubra todos los arcos.

31

## CFG de crearListaN



- ▶ ¿Cuántos nodos (sentencias) hay? 6
- ▶ ¿Cuántos arcos (flechas) hay? 6
- ▶ ¿Cuántas decisiones (arcos True y arcos False) hay? 2

32

## Cubrimiento de Condiciones Básicas

- ▶ Una condición básica es una fórmula atómica (i.e. no divisible) que componen una decisión.
  - ▶ Ejemplo: `(digitHigh==1 || digitLow===-1) && len>0`
  - ▶ Condiciones básicas:
    - ▶ `digitHigh==1`
    - ▶ `digitLow===-1`
    - ▶ `len>0`
  - ▶ No es condición básica: `(digitHigh==1 || digitLow===-1)`
- ▶ Criterio de Adecuación: cada condición básica de cada decisión en el CFG debe ser evaluada a verdadero y a falso al menos una vez
- ▶ Cobertura:

$$\frac{\text{cantidad de valores evaluados en cada condición}}{2 \times \text{cantidad condiciones basicas}}$$

33

## Cubrimiento de Condiciones Básicas

- ▶ Sea una única decisión:  
`(digitHigh==1 || digitLow===-1) && len>0`
- ▶ Y el siguiente test case:

Entrada	<code>digitHigh==1?</code>	<code>digitLow===-1?</code>	<code>len&gt;0?</code>
<code>digitHigh=1, digitLow=0 len=1,</code>	True	False	True
<code>digitHigh=0, digitLow=-1 len=0,</code>	False	True	False
- ▶ ¿Cuál es el cubrimiento de condiciones básicas?

$$C_{\text{cond.básicas}} = \frac{3}{2 \times 3} = \frac{3}{6} = 50\%$$

34

## Cubrimiento de Condiciones Básicas

- ▶ Sea una única decisión:  
`(digitHigh==1 || digitLow===-1) && len>0`
- ▶ Y el siguiente test case:

Entrada	<code>digitHigh==1?</code>	<code>digitLow===-1?</code>	<code>len&gt;0?</code>
<code>digitHigh=1, digitLow=0 len=1,</code>	True	False	True
<code>digitHigh=0, digitLow=-1 len=0,</code>	False	True	False
- ▶ ¿Cuál es el cubrimiento de condiciones básicas?

$$C_{\text{cond.básicas}} = \frac{6}{2 \times 3} = \frac{6}{6} = 100\%$$

35

## Cubrimiento de Branches y Condiciones Básicas

- ▶ **Observación** Branch coverage no implica cubrimiento de Condiciones Básicas
  - ▶ Ejemplo: `if(a && b)`
  - ▶ Un test suite que ejercita solo `a = true, b = true` y `a = false, b = true` logra cubrir ambos branches de `if(a && b)`
  - ▶ **Pero:** no alcanza cubrimiento de decisiones básicas ya que falta `b = false`
- ▶ El criterio de cubrimiento de Branches y condiciones básicas necesita 100 % de cobertura de branches y 100 % de cobertura de condiciones básicas
- ▶ Para ser aprobado, todo software que controla un avión necesita ser testeado con cubrimiento de branches y condiciones básicas (RTCA/DO-178B y EUROCAE ED-12B).

36

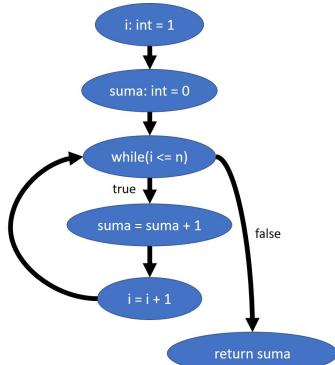
## Cubrimiento de Caminos

- ▶ Criterio de Adecuación: cada camino en el CFG debe ser transitado por al menos un test case.
- ▶ Cobertura:  
$$\frac{\text{cantidad caminos transitados}}{\text{cantidad total de caminos}}$$

37

## Caminos para el CFG de sumar

Sea el siguiente CFG:

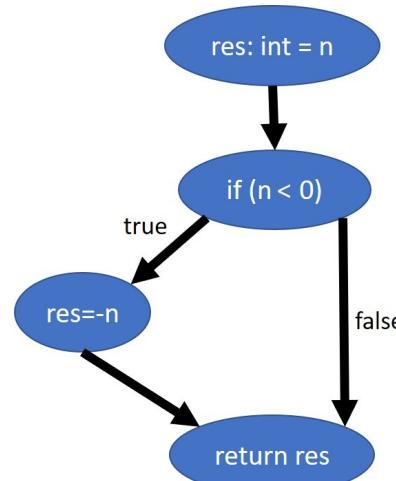


¿Cuántos caminos hay en este CFG? La cantidad de caminos no está acotada ( $\infty$ )

39

## Caminos para el CFG de valorAbsoluto

Sea el siguiente CFG:



¿Cuántos caminos hay en este CFG? 2

38

## Recap: Criterios de Adecuación Estructurales

- ▶ En todos estos criterios se usa el CFG para obtener una métrica del test suite
- ▶ **Sentencias:** cubrir todas los nodos del CFG.
- ▶ **Arcos:** cubrir todos los arcos del CFG.
- ▶ **Decisiones (Branches):** Por cada if, while, for, etc., la guarda fue evaluada a verdadero y a falso.
- ▶ **Condiciones Básicas:** Por cada componente básico de una guarda, este fue evaluado a verdadero y a falso.
- ▶ **Caminos:** cubrir todos los caminos del CFG. Como no está acotado o es muy grande, se usa muy poco en la práctica.

40

## esPrimo()

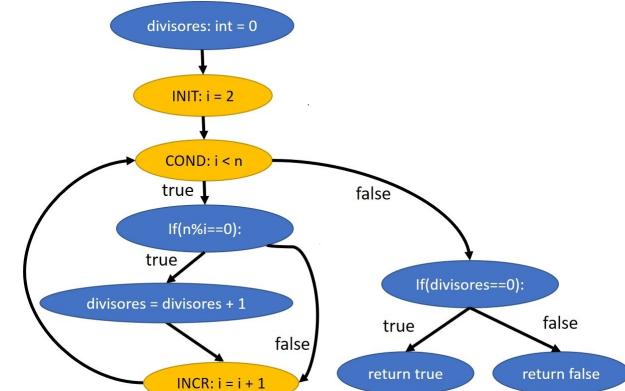
Sea la siguiente implementación que decide si un número  $n > 1$  es primo:

```
def esPrimo(n : int) → bool:  
    divisores: int = 0  
    for i in range(2, n, 1):  
        if (n % i == 0):  
            divisores = divisores + 1  
  
    if (divisores == 0):  
        return true  
    else:  
        return false
```

Graficar el CFG de la función esPrimo().

41

## esPrimo()



42

## Cubrimientos

Sea el siguiente test suite para esPrimo():

- ▶ Test Case #1: valorPar
  - ▶ Entrada:  $n = 2$
  - ▶ Salida esperada: *result = true*
- ▶ Test Case #2: valorImpar
  - ▶ Entrada:  $n = 3$
  - ▶ Salida esperada: *result = true*
- ▶ ¿Cuál es el cubrimiento de sentencias (nodos) del test suite?

$$Cov_{sentencias} = \frac{7}{9} \sim 77\%$$

- ▶ ¿Cuál es el cubrimiento de decisiones (brances) del test suite?

$$Cov_{branches} = \frac{4}{6} \sim 66\%$$

43

## Cubrimientos

Sea el siguiente test suite para esPrimo():

- ▶ Test Case #1: valorPrimo
  - ▶ Entrada:  $n = 3$
  - ▶ Salida esperada: *result = true*
- ▶ Test Case #2: valorNoPrimo
  - ▶ Entrada:  $n = 4$
  - ▶ Salida esperada: *result = false*
- ▶ ¿Cuál es el cubrimiento de sentencias (nodos) del test suite?

$$Cov_{sentencias} = \frac{9}{9} = 100\%$$

- ▶ ¿Cuál es el cubrimiento de decisiones (brances) del test suite?

$$Cov_{branches} = \frac{6}{6} = 100\%$$

44

## Discusión

- ▶ ¿Puede haber partes (nodos, arcos, branches) del programa que no sean alcanzables con **ninguna** entrada válida (i.e. que cumplan la precondición)?
- ▶ ¿Qué pasa en esos casos con las métricas de cubrimiento?
- ▶ Existen esos casos (por ejemplo: código defensivo o código que sólo se activa ante la presencia de un estado inválido)
- ▶ El 100 % de cubrimiento suele ser no factible, por eso es una medida para analizar con cuidad y estimar en función al proyecto (ejemplo: 70 %, 80 %, etc.)

45

## Introducción a la POO y APIs

### Introducción a la Programación

1

## ¿Qué es la Programación Orientada a Objetos (POO)?

- ▶ Es un paradigma de programación (como Funcional e Imperativo)
- ▶ Es pensar y organizar el código en términos de clases y objetos (también se los llama instancias de una clase)
- ▶ Las clases son entidades que combinan variables (atributos) y operaciones (métodos) para definir las propiedades y el comportamiento de los objetos.
- ▶ Las clases definen nuevos tipos de datos.
- ▶ Los objetos pueden interactuar entre sí a través de sus métodos y atributos.
- ▶ Se asemeja a la forma en que pensamos y modelamos el mundo real mediante TADs, pero a nivel de lenguaje de programación, no de especificación.
- ▶ Algunos conceptos clave de la POO incluyen encapsulamiento, herencia y polimorfismo.
- ▶ Python está fuertemente orientado a objetos, no obstante, no es condición necesaria hacer uso de las clases para crear un programa (como ocurre en otros lenguajes como Java o Smalltalk).

2

## ¿Qué es una Clase en Python?

- ▶ Definición de una clase en Pyhton

```
class Persona:  
    def __init__(self, nombre: str, edad:int):  
        self.nombre = nombre  
        self.edad = edad  
  
    def presentarse(self):  
        print(f"Hola, mi nombre es {self.nombre} y mi edad es {self.edad}")
```

- ▶ `__init__` es un método especial o constructor que se utiliza para inicializar los objetos (instancias) de una clase.
- ▶ Este método se llama automáticamente cuando se crea un nuevo objeto.
- ▶ El primer parámetro de `__init__` es `self`, que es una referencia al propio objeto.
- ▶ Todos los métodos (funciones) definidos dentro de una clase deben tener el parámetro `self` a través del cual se pueden acceder y modificar los atributos del objeto.

3

## ¿Qué es una Clase en Python?

- ▶ Creación de una instancia de una clase (objeto)

```
# Crear una instancia de la clase Persona  
persona1 = Persona("Juan", 30)
```

- ▶ Acceso a los atributos de un objeto

```
print(persona1.nombre) # Imprime "Juan"  
print(persona1.edad) # Imprime 30  
persona1.edad = 20 # Asigna 20 al atributo edad
```

- ▶ Uso de los metodos de la instancia

```
persona1.presentarse()  
# Imprime "Hola, mi nombre es Juan y mi edad es 20"
```

4

## Encapsulamiento

- ▶ Permite ocultar los detalles internos de implementación de una clase.
- ▶ Los usuarios de una clase solo necesitan conocer la interfaz pública, es decir, los métodos y atributos accesibles, respetando la documentación (especificación) de la clase.
- ▶ En Python, tanto los atributos como los métodos de una clase pueden tener tres niveles de visibilidad en términos de acceso desde fuera de la clase: público, protegido y privado.
- ▶ Estos niveles de visibilidad se indican mediante el uso de guiones bajos (*underscores*) en los nombres de los atributos y métodos.
- ▶ Para atributos o métodos protegidos se utiliza un guion bajo y para privados doble guion bajo como prefijo en sus nombres.

5

## Encapsulamiento

```
class Persona:  
    def __init__(self, nombre: str, edad: int):  
        self.__nombre = nombre  
        self.__edad = edad  
  
    def dameNombre(self):  
        return self.__nombre  
  
    def dameEdad(self):  
        return self.__edad  
  
    def definirNombre(self, nombre: str):  
        self.__nombre = nombre  
  
    def definirEdad(self, edad: int):  
        self.__edad = edad  
  
    def presentarse(self):  
        print(f'Hola, mi nombre es {self.__nombre} y mi edad es {self.__edad}')  
  
personal1 = Persona("Pablo", 30)  
personal1.presentarse()  
#print(personal1.__nombre) #Esto da error, no se puede acceder directamente  
print(personal1.dameNombre())  
personal1.definirNombre("Pepe")  
print(personal1.dameNombre())
```

6

## Herencia

- ▶ Permite la creación de nuevas clases basadas en clases existentes, heredando sus atributos y métodos.
- ▶ Las clases nuevas se llaman derivadas o subclases y la clase existente de la cual heredan se llama clase base o superclase.
- ▶ Define jerarquías de clases que comparten diversos métodos y atributos.
- ▶ Podemos sobrecargar (sobre-escribir) métodos para cada subclase.

7

## Herencia

```
class Animal:  
  
    def __init__(self, nombre: str, edad: int):  
        self.nombre = nombre  
        self.edad = edad  
  
    def emitir_sonido(self):  
        pass  
  
class Perro(Animal):  
    def emitir_sonido(self):  
        return "Guau!"  
  
class Gato(Animal):  
    def emitir_sonido(self):  
        return "Miau!"  
  
class Pato(Animal):  
    def emitir_sonido(self):  
        return "Cuac!"  
  
# Crear instancias de diferentes animales  
miPerro = Perro("Dylan", 10)  
miGato = Gato("Azrael", 15)  
miPato = Pato("Daffy", 2)  
  
# Acceder a los atributos y métodos de las subclases  
print(f'{miPerro.nombre} tiene {miPerro.edad} y hace {miPerro.emitir_sonido()}')  
print(f'{miGato.nombre} tiene {miGato.edad} y hace {miGato.emitir_sonido()}')
```

8

## Polimorfismo

- ▶ Recordatorio: se llama polimorfismo a una función que puede aplicarse a distintos tipos de datos sin redefinirla.
- ▶ En Python, el polimorfismo se logra a través de la herencia y la implementación de métodos con el mismo nombre en diferentes subclases.
- ▶ Mediante la sobrecarga de métodos puedo obtener una función que puede recibir distintos objetos (tipos de datos) sin redefinirla.

9

## Polimorfismo

```
class Animal:  
    def __init__(self, nombre: str, edad: int):  
        self.nombre = nombre  
        self.edad = edad  
  
    def emitir_sonido(self):  
        pass  
  
class Perro(Animal):  
    def emitir_sonido(self):  
        return "Guau!"  
  
class Gato(Animal):  
    def emitir_sonido(self):  
        return "Miau!"  
  
# Funcion que utiliza polimorfismo  
def hacer_emitir_sonido(animal):  
    return animal.emitir_sonido()  
  
# Crear instancias de diferentes animales  
miPerro = Perro("Dylan", 10)  
miGato = Gato("Azrael", 15)  
  
# Llamar a la funcion polimorifica con diferentes instancias  
print(hacer_emitir_sonido(miPerro)) # Imprime "Guau!"  
print(hacer_emitir_sonido(miGato)) # Imprime "Miau!"
```

10

## Un paso más allá: ¿Qué es una API?

Un poquito fuera del alcance de la materia...

- ▶ El término API es muy usado actualmente y está relacionado con poder usar desde un programa funcionalidades de otro programa.
- ▶ API significa *Application Programming Interface* (Interfaz de Programación de Aplicaciones, en español). Una API define cómo las distintas partes de un software deben interactuar, especificando los métodos y formatos de datos que se utilizan para el intercambio de información.
- ▶ En el contexto de desarrollo de software, una API puede ser considerada como un contrato entre dos aplicaciones.
- ▶ Una API encapsula el comportamiento de otro programa y en muchos casos, su utilización es similar al uso de un TAD. Detrás de este encapsulamiento se esconden un gran número de problemas a resolver como ser: conexiones de red, uso de protocolos, manejo de errores, transformaciones de datos, etc (y son muchos etcs).

11

## Veamos una API cualquiera: Google Translate API

Un paso más allá: ¿Qué es una API?

- ▶ Instalamos el módulo: pip install googletrans==3.1.0a0
- ▶ Y veamos que nos ofrece su contrato:  
translator = Translator()
  - ▶ El método translate(texto, idioma origen, idioma destino) devuelve la siguiente estructura :
    - ▶ src: idioma original dest: idioma destino
    - ▶ origin: texto en idioma original
    - ▶ text: texto traducido
    - ▶ pronunciation: pronunciación del texto traducido

12

## ¿Podremos implementar este problema?

```
problema traducirTexto(in nombreArchivo: string, in idiomaOrigen:  
string, in idiomaDestino: string) : {  
    requiere: {El archivo nombreArchivo debe existir.}  
    asegura: {Se crea un archivo llamado idiomaDestino – nombreArchivo  
    cuyo contenido será el resultado de traducir cada una de sus filas}  
    asegura: {Si el archivo archivoDestino existia, se borrará todo su  
    contenido anterior}  
}
```

13

## Documentación de Python

Python (como muchos de los otros lenguajes) tiene documentación pública con la descripción del comportamiento de sus distintos tipos de datos.

The screenshot shows a section of the Python 3.11.3 documentation titled "5. Estructuras de datos". It specifically covers lists. The left sidebar has a "Tabla de contenido" (Table of Contents) with sections like "5. Estringentes de datos" and "5.1. Más sobre listas". The main content area starts with "5.1. Más sobre listas", which includes a list of methods: `append()`, `extend(iterable)`, `insert(i, x)`, and `remove(x)`. Below each method is its description in Spanish. At the bottom of the page, there's a note about `pop(i)`.

14

## Documentación de API Google Translate

La API de Google Translate tiene su propia página de especificación

- <https://py-googletrans.readthedocs.io/en/latest/>

The screenshot shows the "API Guide" for the `googletrans.Translator` class. It includes the class definition, parameters for `Translator`, and detailed descriptions for the `translate` method. The `translate` method takes `text`, `dest`, `src`, and `**kwargs` parameters. It translates text from source language to destination language. The `dest` parameter specifies the language to translate the source text into, and `src` specifies the source language.

15