Teórico 1 y algunas definiciones

Qué es una computadora

▸ Es una máquina electrónica.
▸ Su función es procesar datos.
▸ El procesamiento se realiza en forma automática.
▸ El procesamiento se realiza siguiendo un programa que está almacenado en una memoria interna.

Qué es un algoritmo

▸ Es la descripción de los pasos a realizar.
▸ Especifica una sucesión de instrucciones primitivas.
▸ El objetivo es resolver un problema.

Qué es un programa

▸ Es la descripción de un algoritmo en un lenguaje de programación.
▸ Se describe en un lenguaje de programación

Qué es una especificación

▸ Es la descripción del problema a resolver. Habitualmente, dada en lenguaje formal.
▸ Es un contrato que da las propiedades de los datos de entrada y las propiedades de la solución.

En resumen: Vos tenes la especificación que es el qué, te describe cierto problema y te pide cierta solución. El algoritmo es la descripción de esta solución escrita para humanos, es el cómo resolvemos este problema.
En base al algoritmo diseñado, uno hace un programa en lenguaje de programación (código que ejecuta la computadora).

Modularización

- **Top-Down:** Descomponer un problema grande en problemas más pequeños.
- **Buttom-Up:** Componerlos y obtener la solución al problema original.

Lenguaje Máquina: Son lenguajes que están expresados en un código 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). No están diseñados para la interpretación de las personas.

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.

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.

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 es más cómodo con una 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 para los desarrolladores: Permiten hacer destacado (highlighting) de determinadas palabras y símbolos.
- Son capaces de verificar la sintaxis de los programas escritos (linters)
- Se pueden especializar según cada lenguaje particular
- Permiten hacer depuración o debugging!

La importancia de la especificación

La especificación tiene un rol “contractual”, es un contrato entre el programador de una función y el usuario de esa función. El programador tiene la obligación de hacer que su programa cumpla con la especificación, y el derecho del usuario es poder asumir que al ejecutar el programa, el resultado va a ser correcto.
A parte de esto, es un insumo para las actividades de Testing, Verificación formal de correctitud, Derivación formal (construir un programa a partir de la especificación).

Otra cosa: Puede haber varios algoritmos que cumplan una misma especificación y a partir de un algoritmo van a existir múltiples programas que implementan dicho algoritmo.

Paradigmas de programación

Son 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

Programación Declarativa

Son lenguajes donde el programador le indica a la máquina lo que quiere hacer y el resultado que desea, pero no necesariamente el cómo hacerlo.
Describe un conjunto de condiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución.

- ➔ **Paradigma Lógico:** Los programas están construídos ´únicamente por expresiones lógicas (es decir, son Verdaderas o Falsas). PROLOG

- ➔ **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. LISP, GOFER, HASKELL.

  En este caso, el programa es una colección de funciones que transforman datos de entrada en un resultado.
  Los lenguajes funcionales nos dan herramientas para explicarle a la computadora como computar esas funciones
  La ejecución de un programa en este caso corresponde a la evaluación de una expresión, 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.

  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

  Reducción.
  El mecanismo de evaluación en un lenguaje funcional es la reducción

  Recursión

Los programas son funciones necesariamente

Programación Imperativa

Son lenguajes en los que el programador debe precisarle a la máquina de forma exacta el proceso que quiere realizar.
Describe la programación como una secuencia de instrucciones o comandos que cambian el estado de un programa.

Diferencias con el paradigma funcional

- Los programas no necesariamente son funciones, en el funcional sí. Acá se llaman procedimientos.
- Los programas ahora pueden devolver más de un valor
- Nuevo concepto de variables, Cambian explícitamente de valor a lo largo de la ejecución de un programa. **Nueva operación: la asignación:** Cambiar el valor de una variable
- Posiciones de memoria
- **Pérdida de la transparencia referencial**
- Las funciones no pertenecen a un tipo de datos necesariamente, en el funcional sí.
- Distinto mecanismo de repetición: **En lugar de la recursión usamos la iteración**
- Nuevo tipo de datos: El arreglo

➔ **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. PASCAL, C, FORTRAN, FOX, COBOL JAVA, PYTHON, .NET, PHP

➔ **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. SMALLTALK JAVA, PYTHON, .NET, PHP

Variables en imperativo

Las variables en el imperativo, son un nombre asociado a un espacio de memoria. La variable puede tomar distintos valores a lo largo de la ejecución.

Alcance / ámbito / 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.

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)

Transformación de estados

Esto no sucede en el funcional.

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 y después de ejecutar la última instrucción.

Entonces, veremos la ejecución de un programa como una sucesión de estados. La instrucción que transforma estados es **la asignación.** El resto de las instrucciones son de control: modifican el flujo de ejecución, es decir, el orden de ejecución de las instrucciones.

Copia y referencia

Para indicar que una función recibe argumentos de entrada usamos variables. Estas variables toman valor cuando el llamador invoca a la función.

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, 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

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

➔ **Pasaje por valor / por copia:** Se crea una copia local de la variable dentro de la función. Coloca en la posición de memoria del argumento de entrada el valor de la expresión usada en la invocación, entonces la expresión original con la que se realizó la invocación queda protegida contra escritura.

➔ **Pasaje por referencia:** Se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera. La función no recibe un valor sino que **implícitamente recibe una dirección de memoria donde encontrar el argumento.** Entonces 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 que, 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.

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.

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 referencie al valor 3.

Tipos de Datos:

Es un conjunto de valores provisto de ciertas operaciones para trabajar con estos valores.

Parámetros de tipo fecha

- **Valores:** ternas de números enteros
- **Operaciones:** comparación, obtener el año, ..

Parámetros de tipo dinero

- **Valores:** Números reales con dos decimales
- **Operaciones:** suma, resta, …

Tipo de datos de nuestro lenguaje de especificación:

Básicos

- Enteros (Z)
- Reales (R)
- Booleanos (Bool)
- Caracteres (Char)

Enumerados

Ejemplo de tipo enumerado Definimos el tipo Días así: enum Día { LUN, MAR, MIER, JUE, VIE, SAB, DOM }
Cantidad finita de elementos. Cada uno, denotado por una constante.

Uplas: Tuplas, ternas, etc. Los elementos dentro de una upla admiten diferentes tipos

Secuencias

Todos los elementos de una secuencia siempre son del mismo tipo.

Clases de tipos

Se llama clase de tipo al 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 por ejemplo.

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, ... }, { (==), (/=) })

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 una longitud variable. **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.

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

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.

Tipos Abstractos de Datos (TAD)

Es un modelo que define valores y las operaciones que se pueden realizan 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.

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)

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)

Lenguaje Python

Vamos a usarlo para la **programación imperativa** (también soporta parte del paradigma de objetos, y parte del paradigma funcional)
Nosotros no lo usamos ni orientado a objetos ni con memoria dinámica

Tipado Dinámico:

Una variable puede tomar valores de distintos tipos. Nosotros lo vamos a pensar con tipado estático

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.

CVSs y Git

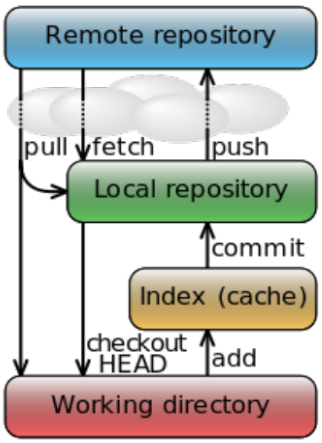
Version Control Systems (CVSs)
▸ Permite organizar el trabajo en equipo
▸ Guarda un historial de versiones de los distintos archivos que se usaron
▸ Existen distintas aplicaciones: svn, cvs, hg, git

Algunas Palabras:

- **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. Por ejemplo, está la rama Master y se trabaja en la Develop

Git workflow graph:



Lógica proposicional, semántica clásica

Es la lógica que habla sobre las proposiciones. Son oraciones que tienen un valor de verdad, Verdadero o Falso.
Sirve para poder deducir el valor de verdad de una proposición, a partir de conocer el valor de otras.

Fórmulas
1. True y False son fórmulas
2. Cualquier variable proposicional es una fórmula (p, q, r, etc)
3. Si A es una fórmula, ¬A es una fórmula .
4. Si A1, A2, . . . , An son fórmulas, (A1 ∧ A2 ∧ · · · ∧ An) es una fórmula.
5. Si A1, A2, . . . , An son fórmulas, (A1 ∨ A2 ∨ · · · ∨ An) es una fórmula.
6. Si A y B son fórmulas, (A → B) es una fórmula.
7. Si A y B son fórmulas, (A ↔ B) es una fórmula

Relación de fuerza
Decimos que A es más fuerte que B cuando (A → B) es tautología.

Semántica trivaluada (secuencial)

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 ∨ x/y = 5
Para esto, introducimos tres valores de verdad:
1. verdadero (V)
2. falso (F)
3. indefinido (⊥)

La semántica secuencial se llama así porque los términos se evalúan de izquierda a derecha. En este caso, la evaluación termina cuando se puede deducir el valor de verdad, aunque el resto esté indefinido.

Se introducen los operadores lógicos ∧L (y-luego, o conditional and, o cand), ∨L (o-luego o conditional or, o cor).

Por convención, dado que nuestros tipos de datos siempre tendrán como valor posible el indefinido o ⊥, en general, asumiremos que estamos utilizando la lógica trivaluada por default. Es decir, salvo en los casos donde se indique lo contrario: ∧ podrá ser interpretado como ∧L directamente.

Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas (⊥).

¿Cómo podemos clasificar las funciones?

- **Funciones totales:** nunca se indefinen. suc x = x + 1
- **Funciones parciales:** hay argumentos para los cuales se indefinen. división x y = div x y

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)

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)

Lenguajes fuertemente y débilmente tipados

▸ 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

Órdenes de evaluación:

Lazy / Perezoso

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).

↠ (3+4) + (suc (2*3))
↠ 7 + (suc (2*3))
↠ 7 + ((2*3) + 1)
↠ 7 + (6 + 1)
↠ 7 + 7
↠ 14

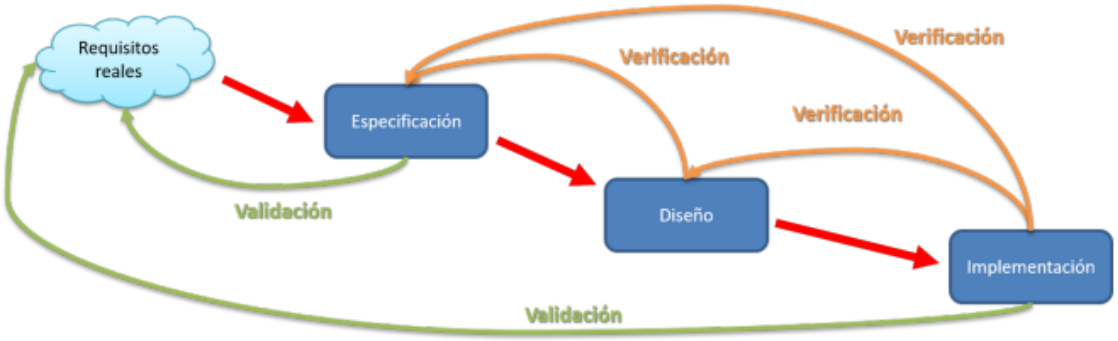
Eager / Ansioso

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

Polimorfismo

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.

Validación y Verificación



La validación es una prueba de que cumple con los requisitos y de que el producto es confiable. ¿Estamos haciendo el producto correcto?
La verificación garantiza que los equipos creen el producto de acuerdo con los procesos y estándares propios de su organización. ¿Estamos haciendo el producto correctamente?

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. Uno de los objetivos principales en el desarrollo de software es obtener productos de alta calidad.
Validación y Verificación son procesos que ayudan a mostrar que el software cubre las expectativas para las cuales fue construido.

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

Nociones Básicas

➔ **Falla**: Diferencia entre los resultados esperados y reales. Una falla es la manifestación de un defecto

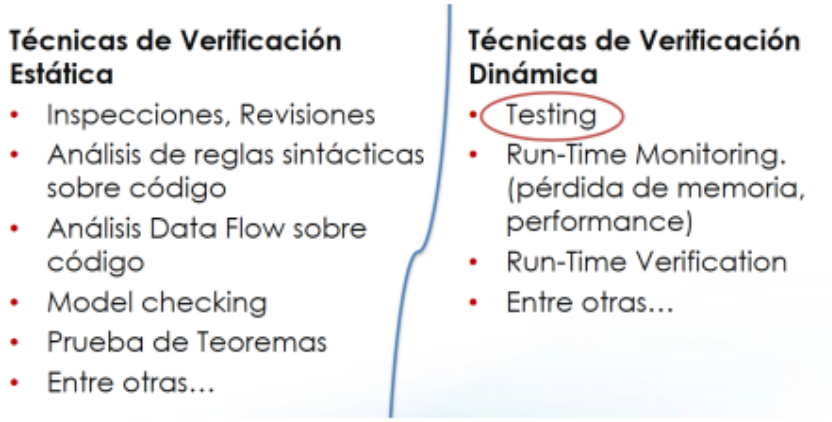
- ➔ **Defecto:** Desperfecto en algún componente del sistema (en el texto del programa, una especificación, un diseñoo, etc), que origina una o más fallas
- ➔ **Error (Bug):** Equivocación humana. Un error lleva a uno o más defectos, que están presentes en un producto de software.

Objetivos principales
▸ Descubrir defectos en el sistema
▸ Asegurar que el software respeta su especificación
▸ Determinar si satisface las necesidades de sus usuarios

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 determinaría el grado de confianza que se necesita

Verificación estática y dinámica

▸ **Dinámica:** Se trata de ejecutar y observar el comportamiento de un producto.
▸ **Estática:** Se trata del análisis de una representación estática del sistema para descubrir problemas.



Testing

Es el proceso de ejecutar un producto para verificar que satisface los requerimientos (en nuestro caso, la especificación) e Identificar diferencias entre el comportamiento real y el comportamiento esperado. El objetivo del testing es encontrar defectos en el software. Representa entre el 30 % al 50 % del costo de un software confiable.

Niveles de Test

- **Test de Sistema:** Comprende todo el sistema.

- **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 y la comunicación entre partes
- **Test de Unidad:** Se realiza sobre una unidad de código pequeña, claramente definida.

Otras Nociones

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).

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.
Y cuanto más complicada la entrada (ej: secuencias), más tiempo lleva hacer testing.
Entonces, la mayoría de las veces, el testing exhaustivo no es práctico. Una de las limitaciones del testing justamente es que 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

Test de Caja Negra

Los casos de test se generan analizando la especificación sin considerar la implementación.
Los datos de test se derivan a partir de la descripción del programa sin conocer su implementación.

Test de Caja Blanca

Los casos de test se generan analizando la implementación para determinar los casos de test. Los datos de test se derivan a partir de la estructura interna del programa.

¿Se puede automatizar el testing?
Existen herramientas que permiten programar estos casos de pruebas. Vimos que el nivel más básico del testing se denomina TEST UNITARIO.
Para hacer test unitario, la gran mayoría de los lenguajes de programación tienen herramientas que permiten programar casos de prueba. En el caso de Haskell: HUnit

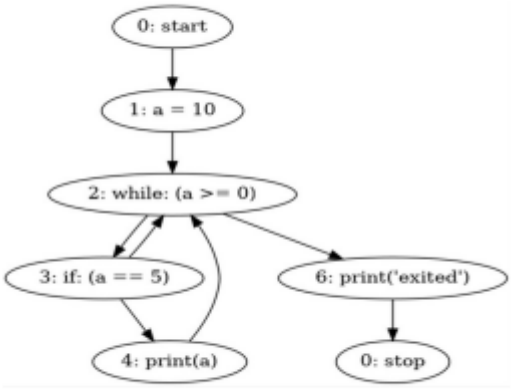
Método de partición de categorías
Se trata de buscar los conjuntos de valores donde se espera un comportamiento similar. Debería ser una participación sin dejar valores afuera.
Se trata de identificar algunas propiedades o restricciones de las elecciones en el marco de la unidad funcional. Las clasificaciones más comunes son:
▸ **Error:** Se clasificarían 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

Criterios de caja blanca o estructurales

Los criterios de caja blanca permiten identificar casos especiales según el flujo de control de la aplicación: Ver qué sucede si entra o no en un IF ; ver qué 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 ya que permite encontrar más casos o definir casos más específicos. También nos viene bien el test de caja blanca para utilizarlo como criterio de adecuación del Test de Caja Negra ya que brinda herramientas que nos ayudar a determinar cuán bueno o confiable resultaron ser los test suites definidos.

Control-Flow Graph (CFG)

El CFG de un programa es solo una representación gráfica del programa, es independiente de las entradas (su definición es estática) y 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
Por “partes” nos referimos a: nodos, arcos, caminos, decisiones…



Como regla tomamos que un test suite es suficientemente bueno cuando todas las sentencias y arcos son ejecutadas. Ya que el cubrimiento de arcos garantiza el cubrimiento de nodos, el cubrimiento de decisiones/ branches…
Ahhh pero no asegura el cubrimiento de los caminos ni el de las condiciones básicas.
Aunque hay veces que no se puede cubrir todos los arcos.

: 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.

Cubrimiento de Sentencias

Cada nodo (sentencia) en el CFG debe ser ejecutado al menos una vez por algún test case.

- Cobertura: cantidad nodos ejercitados / cantidad nodos

Cubrimiento de Arcos

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 pero el cubrimiento de sentencias (nodos) no incluye al cubrimiento de arcos.

- Cobertura: cantidad arcos ejercitados /cantidad arcos

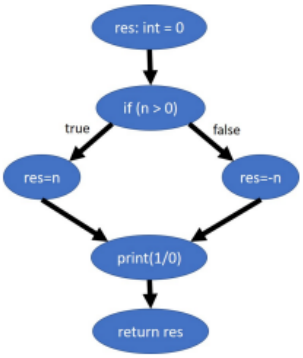
Cubrimiento de Decisiones (o Branches)

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 ejercite.

- Cobertura: cantidad decisiones ejercitadas / cantidad decisiones

El cubrimiento de decisiones no implica el cubrimiento de los arcos del CFG. A veces transitar por un arco es la única opción entonces no sería una decisión.

Sea el siguiente CFG:



En este ejemplo, puedo construir un test suite que cubra todos los branches pero que no cubra todos los arcos.

Cubrimiento de Caminos

Cada camino en el CFG debe ser transitado por al menos un test case.
Un camino es

- Cobertura: cantidad caminos transitados / cantidad total de caminos

Cubrimiento de Condiciones Básicas

Una condición básica es una fórmula atómica (i.e. no divisible) que componen una decisión. Cada condición básica de cada decisión en el CFG debe ser evaluada a verdadero y a falso al menos una vez.

- Cobertura: cantidad de valores evaluados en cada condición / 2 × cantidad condiciones básicas