Notas: Introducción a la programación

Created on Thursday, March 30, 2023

Facundo Coronel

Clase 1

¿Qué es una computadora?

Una computadora es una máquina que procesa información automáticamente de acuerdo con un programa almacenado.

- Es una máquina.
- Su función es procesar información, y estos términos deben entenderse en sentido amplio.
- El procesamiento se realiza en forma automática.
- El procesamiento se realiza siguiendo un programa.
- Este programa está almacenado en una memoria interna de la misma computadora.

¿Qué es un algoritmo?

Un algoritmo es la descripción de los pasos precisos para resolver un problema a partir de datos de entrada adecuados.

- Es la descripción de los pasos a dar.
- Específica una sucesión de pasos primitivos.
- El objetivo es resolver un problema.
- Un algoritmo típicamente trabajar a partir de datos de entrada.

¿Qué es un programa?

Un programa es la descripción de un algoritmo en un lenguaje de programación.

- Corresponde a la implementación concreta del algoritmo para ser ejecutado en una computadora.
- Se describe en un lenguaje de programación.

Especificación, algoritmo y 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?

3 **Programa:** descripción de la solución para ser ejecutada en una computadora.

- Nos volvemos a preguntar cómo resolver el problema.
- La solución encontrada la queremos transcribir a un lenguaje de programación.

Resolución de problemas

Dado un problema a resolver, queremos:

1 Poder describir de una manera clara y unívoca (especificación).

- Esta descripción debería poder ser validada contra el problema real.

2 Poder diseñar una solución acorde a dicha especificación.

- Este diseño debería poder ser verificado con respecto a la especificación.

3 Poder implementar un programa acorde a dicho diseño.

- Este programa debería poder ser verificado con respecto a su especificación y diseño.
- Este programa debería ser la solución al problema planteado.

Los conceptos de modularización y encapsulamiento siempre estarán relacionados con los principios de diseño de software. La estrategia la resumiremos en:

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

Hay que tener en cuenta que:

- Dado un problema, será importante describirlo sin ambiguedades.
- Una buena descripción no debería condicionarse con sus posibles soluciones.
- Saber que dado un problema, hay muchas de describirlo y a su vez muchas formas de solucionarlo.

Especificación de problemas

Una especificación es un contrato que define qué se debe resolver y qué propiedades debe tener la solución. **Importante:** 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 cunplir un rol "contractual", la especificación del problema es insumo para las actividades de: testing, verificación formal de corrección y derivación formal (construir un programa a partir de la especificación).

Lenguajes naturales y formales

Lenguajes naturales

- Idiomas como el castellano.
- Mucho poder expresivo (modos verbales, tiempos verbales, y más).
- Existe un contexto y se pueden dar suposiciones.
- No se usan para especificar porque pueden ser ambiguos y no tienen un cálculo formal.

Lenguajes formales

- Tienen una 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.
- Ejemplos:

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

    2 Lógica: proposicional, de primer orden, modales, etcétera.

    3 Lenguajes de especificación de programas.

Contratos

Una especificación es un contrato entre el programador de una función y el usuario de esa función.

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.

3 Postcondiciones o cláusualas "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).

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 párametro tiene un tipo de datos. Los tipo de datos son un conjunto de valores provisto de ciertas operaciones para trabajar con estos valores.

¿Por qué escribir la especificación del problema?

- Nos ayuda a entender mejor el problema.
- Nos ayuda a construir el programa.
- Nos ayuda a prevenir errores en el programa.

Clase 2

Defición (especificación) de un problema

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

Características

1 **nombre:** nombre que le damos al problema.

- Será resuelto por una función con ese mismo nombre

2 **parámetros:** lista de parámetros separada por comas, donde cada parametro contiene:

- Nombre del parámetro
- Tipo de datos del parámetro

3 **tipo de dato del resultado:** tipo de dato del resultado del problema (incialmente especificaremos funciones).

- En los asegura, podremos referenciar el valor devuelto con el nombre de **res**

4 **etiquetas:** son nombres opcionales que nos servirán para nombrar declarativamente a las condiciones de los requiere o aseguras.

Sobre los requiere

- Describen todas las condiciones y posibles valores o casuísticas de los parámetros de entrada.
- Puede haber más de un requiere (se recomienda uno por renglón). Se asume que valen todos juntos (es una conjunción).
- Evitar contradicciones. Un requiere no puede contradecir al 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 (se recomienda uno por renglón). Se asume que valen todos juntos (es una conjunción).
- Evitar contradicciones. Un asegura no puede contradecir al otro.

Sintaxis de la lógica proposicional

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 formula, $\neg A$ también lo es.

4 Sea A_i una formula, la conjunción de los A_i también lo es

5 Sea A_i una formula, la disyunción de los A_i también lo es.

6 Si A y B son fórmulas, $(A \longrightarrow B)$ es una fórmula.

7 Si A y B son fórmulas, $(A \leftrightarrow B)$ es una fórmula.

Semántica clásica

Tiene dos valores de verdad: "verdadero" (V) y "falso" (F). Conociciendo 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

Tautologías, contradicciones y contingencias

- Una fórmula es una **tautología** si siempre tomar el valor V para valores definidos de sus variables proposicionales.
- Una fórmula es una **contradicción** si siempre toma el valor F para valores definidos de sus variables proposicionales.
- Una fórmula es una **contingencia** cuando no es ni tautología ni contradicción.

Las siguientes fórmulas son tautologías:

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

Relación de fuerza

Decimos que A es más fuerte que B cuando $(A \rightarrow B)$ es tautología. Se puede decir que A fuerza a B o que B es más debil que A.

Ejemplos:

1. $(p \wedge q)$ es más fuerte que p
2. $(p \vee q)$ no es más fuerte que p
3. p es más fuerte que $(p \rightarrow q)$
4. p no es más fuerte que q
5. False es la fórmula más fuerte de todas
6. True es la fórmula más débil de todas

Expresión bien definida: toda expresión esta bien definida si todas las proposiciones valen T o F. Sin embargo, existe la posibilidad de que haya expresiones que no estén bien definidas. Para esto, introduciremos un nuevo valor de verdad: el indefinido ($\perp$)

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 $\wedge_L(y - luego), \vee_L(o - luego)$

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	$\perp$
$\perp$	$\perp$	$\perp$

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	$\perp$
$\perp$	$\perp$	$\perp$

Cuantificadores

La lógica proposicional no alcanza para expresar o describir propiedades que tendran los elementos de un conjunto. Formalmente, ese grado de abstracción se alcanzaría introduciendo lógica de primer orden (LPO). En la LPO existen cuantificadores que permiten predicar sobre algunos o todos los elementos de un conjunto.

Ejemplo:

($\forall x : T$) P(x): fórmula lógica que afirma que todos los elementos de tipo T cumplen la propiedad P. En esta expresión, la variable x está ligada al cuantificador. Una variable es libre cuando no está ligada a ningún cuantificador.

Operando con cuantificadores:

1 La negación de un cuantificador universal es un cuantificador existencial, y viceversa:

- $\neg(\forall n : Z)P(n) \leftrightarrow (\exists n : Z)\neg P(n)$
- $\neg(\exists n : Z)P(n) \leftrightarrow (\forall n : Z)\neg P(n)$

2 Un cuantificador universal generaliza la conjunción:

- $(\forall n : Z)(a \leq n \leq b \rightarrow P(n)) \wedge P(b+1) \leftrightarrow (\forall n : Z)(a \leq n \leq b+1 \rightarrow P(n))$

3 Un cuantificador existencial generaliza la disyunción:

- $(\exists n : Z)(a \leq n \leq b \wedge P(n)) \vee P(b+1) \leftrightarrow (\exists n : Z)(a \leq n \leq b+1 \wedge P(n))$

Clase 3

El 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.

Problema comunes de las especificaciones

- **Sobre-especificación:** consiste en dar una postcondición más restrictiva de la que se necesita, o bien dar una precondición más laxa. Limita los posibles algoritmo que resuelven el problema, porque impone más condiciones para la salida, o amplía los datos de entrada.
- **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).

Ejemplos de estos problemas

- Sobre-especificado:
  problema distinto($x : Z$) : Z{
  requiere: $\{True\}$
  asegura: $\{res = x + 1\}$
  }
- Sub-especificado:
  problema distinto($x : Z$) : Z{
  requiere: $\{x > 0\}$
  asegura: $\{res \neq x\}$
  }
- Bien especificado::
  problema distinto($x : Z$) : Z{
  requiere: $\{True\}$
  asegura: $\{res \neq x\}$
  }

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, 'a', etc)
- Función (operación) aplicada a otros términos (del tipo T o de otro tipo)

Todos los tipos tienen un elemento distinguido: $\perp$

Lista de tipos de datos

1 Básicos:

- Enteros (Z)
- Reales ($\Re$)
- Booleanos (Bool): tienen los siguientes conectivos lógicos: !, , ||, con la semántica bi-valuada estándar. Las fórmulas que comparan términos de tipo Bool son a = b y $a \neq b$ (a != b)
- Caracteres (Char): sus elementos son las letras, dígitos y símbolos.

2 Enumerados: cantidad finita de elementos. Cada uno, denotado por una constante. Se los define en la forma: enum Nombre constantes

- Nombre (del tipo): tiene que ser nuevo.
- Constantes: nombres nuevos separados por comas. Estas constantes por convención deben estar completamente en mayusculas.
- ord(a) da la posición del elemento en la definición (empezando de 0)
- inversa: se usa el nombre del tipo y funciona como inversa de ord.

3 Uplas: tuplas, de dos o más elementos, cada uno de cualquier tipo.

- $T_0xT_1x...xT_K$: tipos de las k-uplas de elementos de tipos $T_0, T_1, ..., T_K$,respectivamente donde k es fijo.
- nésimo: $(a_0, ..., a_K)_m eselvalora_m$ en caso de que $0 \leq m \leftrightarrow k$. Si no, está indefinido.

4 Secuencias: varios elementos del mismo tipo T, posiblemente repetidos, ubicados en un cierto orden.

- seq T es el tipo de secuencias cuyos elementos son de tipo T.
- T es un tipo arbitrario.
- La secuencia vacía se escribe , cualquiera sea el tipo de elementos de la secuencia.
- Dos secuencias $s_0 y s_1$ son iguales si y sólo si tienen la misma cantidad de elementos y dada una posición, el elemento contenido en la secuencia s_0 es igual al elemento contenido en la secuencia s_1.

Predicados

- Asignan un nombre a una expresión.
- Facilitan la lectura y la escritura de especificaciones.
- Modularizan la especificación: pred p(argumentos) $\{f\}$ donde p es el nombre del predicado y puede usarse en el resto de la especificación en lugar de la fórmula f.

Expresiones condicionales

Función que elige entre dos elementos del mismo tipo según una fórmula lógica (guarda):

- Elige al primero y lo guarda si es verdadero.
- Si no es verdadero, ahora elige al segundo.
- Ejemplo en el lenguaje de especificación:
  { res = if $x \neq 0$ then 1/x else 0 fi }

Modularización

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 facil 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).·

Técnicas Top Down y Bottom Up

- **Top Down:** consiste en establecer una serie de niveles de mayor a menor complejidad (arriba-abajo) que den solución al problema. Al diseñar un sistema siguiendo una estrategia "top-down" se comienza pensando en una visión global de cómo va a funcionar todo. Luego se definen cuáles van a ser los grandes componentes del sistema. Y poco a poco se va refinando y definiendo las funciones de partes más y más pequeñas hasta que se termina construyendo el sistema entero.

- **Bottom Up:** a partir de una necesidad se crean soluciones. El diseño Bottom up (de abajo hacia arriba) o ascendente se refiere a la identificación de aquellos procesos que necesitan computarizarse conforme vayan apareciendo, su análisis como sistema y su codificación, o bien, la adquisición de paquetes de software para satisfacer el problema inmediato.

  Un ejemplo muy simple: Si necesitas de momento un programa que solamente te sume números pares, diseñas el código y listo. Utilizas el programa para la necesidad que surgió. No necesitas dividirlo para crear otros subprogramas que se ajusten después a otros programas.

- Ejemplo en el que se ven las dos técnicas: Sabemos que una empresa cuenta con una estructura interna, una de ellas es que está dividida en varios departamentos como: recursos humanos, mantenimiento, ventas, mercadeo, contabilidad, etcétera. Supongamos que existen problemas en los departamentos, podríamos aplicar una técnica de diseño para llegar a una solución. Primero ocuparemos el diseño Bottom Up, creando un programa que solo solucione el problema generado. Si la empresa después de un tiempo integra un sistema global que permita la solución de los problemas de todos los departamentos, se notará que las soluciones no coinciden, esto pasa porque con éste no hay un análisis previo. Sin embargo, si para llegar a la solución del problema se utiliza el diseño Top- Down se obtendrá el diseño ideal que cubra todas las necesidades, toda vez que existirá un análisis que permita un buen mantenimiento.

Importante: una explicación más detallada de los tipos de datos en el lenguaje de especificación y de sus operaciones está proximo a digitalizarse.

Clase 4

Luego de haber aprendido a especificar problemas, nuestro objetivo es ahora escribir un **algoritmo** que cumpla esa especificación (puede haber varios que hagan eso). Una vez que se tiene el algoritmo, se escribe el **programa** (expresión formal de un algoritmo).

Lenguajes de programación

En palabras simples, un lenguaje de programación es el conjunto de instrucciones a través del cual los humanos interactúan con las computadoras. Permite escribir programas que son ejecutados por computadoras.

Niveles de los lenguajes

- **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:** fueron diseñados para que las personas puedan escribir y entender más fácilmente los programas que escriben.

Código fuente, compiladores, intérpretes

- **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.

Integrated Development Environment (IDE)

Para escribir y ejecutar un programa alcanza con tener:

- Un editor de texto para escribir programas. Por ejemplo, Notepad.
- Un compilador o intérprete (según el lenguaje a utilizar) para procesar y ejecutar el programa.

Ventajas de trabajar con un 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.
- Los IDEs son capaces de verificar la sintaxis de los programas escritor (linters).

- Generar previews de cierto tipo de archivos. Por ejemplo, archivos HTML.
- Suelen tener herramientas integradas. Por ejemplo, el Android Studio tiene emuladores integrados.
- Se pueden especializar según cada lenguaje particular.
- Permiten hacer depuración o debugging.

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 precisarla a la máquina de forma exacta el proceso que quiere realizar.

Características de cada grupo

- Programación Declarativa: describe un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución.

  1 Paradigma Lógico: los programas están construidos únicamente por expresiones lógicas (es decir, son Verdaderas o Falsas). Por ejemplo, PROLOG.

  2 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. Por ejemplo, LISP, GOFER y HASKELL.

- Programación Imperativa: describe la programación como una secuencia de instrucciones o comandos que cambian el estado de un programa.

  1 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. Por ejemplo, PASCAL, C, FORTRAN, FOX y COBOL.

  2 Paradigma Orientado a Objetos: se basa en la idea de encapsular estados y comportamientos en objetos. Los objetos son entidades que se comunican entre sí por medio de mensajes. Por ejemplo, SMALLTALK.

Existen lenguajes multiparadigma que son lenguajes que soportan más de un paradigma de programación. Por ejemplo, JAVA, PYTHON, .NET y PHP.

Paradigma funcional e imperativo

- **Programación funcional:**

  1 En el paradigma funcional, el énfasis está en la evaluación de expresiones y en la programación basada en funciones matemáticas.

  2 Se basa en el concepto de funciones puras, donde una función siempre produce el mismo resultado para los mismos argumentos y no tiene efectos secundarios.

  3 Las variables son inmutables, lo que significa que no pueden cambiar su valor una vez asignadas.

  4 El flujo de control se logra mediante la evaluación de expresiones y llamadas a funciones recursivas.

- **Programación imperativa:**

  1 El paradigma imperativo se basa en la idea de dar instrucciones detalladas a la computadora sobre cómo realizar una tarea.

  2 Se utiliza el concepto de variables mutables, donde se pueden cambiar los valores almacenados en las variables.

  3 Se utilizan estructuras de control como bucles y condicionales para modificar el flujo de ejecución del programa.

  4 Los programas imperativos suelen estar compuestos por una serie de instrucciones que se ejecutan secuencialmente.

- **Principales diferencias:**

  1 Enfoque: El paradigma funcional se centra en la evaluación de expresiones y en el uso de funciones puras, mientras que el paradigma imperativo se centra en dar instrucciones detalladas y modificar el estado de las variables.

  2 Inmutabilidad vs. Mutabilidad: En el paradigma funcional, las variables son inmutables, lo que significa que no pueden cambiar su valor una vez asignadas. En el paradigma imperativo, las variables son mutables y pueden cambiar su valor en cualquier momento.

  3 Efectos secundarios: El paradigma funcional evita los efectos secundarios, lo que significa que las funciones no modifican el estado global y solo dependen de sus argumentos. En el paradigma imperativo, las instrucciones pueden tener efectos secundarios y modificar el estado del programa o del entorno.

  4 Control de flujo: En el paradigma funcional, el flujo de control se logra mediante la evaluación de expresiones y llamadas recursivas a funciones. En el paradigma imperativo, se utilizan estructuras de control como bucles y condicionales para modificar el flujo de ejecución.

Buenas prácticas de programación

Utilizar nombres declarativos

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

  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.

- Evitar la desinformación.
- Usar nombres pronunciables.
- Se deben tener un nombre por concepto. No tener funciones llamadas grabar, guardar y registrar.
- Los nombres de las funciones deben representar el concepto calculado, o la acción realizada en el caso de las funciones que no devuelven un resultado.
- No hacer chistes.

1 Ser prolijo con la indentación (sangría).

2 Respetar la code style (guía de estilo) del lenguaje que estoy usando. En caso de equipos con más de 1 desarrollador, es importante que todos estén alineados y utilizando el mismo estilo.

3 Además de escribir comandos, los lenguajes de programación permiten escribir comentarios. Es importante hacer uso adecuado de los comentarios, para que no oscurezcan el código. Los comentarios no arreglan código de mala calidad. En lugar de comentar el código, hay que clarificarlo. Es importante expresar las ideas en el código, y no en los comentarios. Los mejores comentarios son los conceptos que no se pueden escribir en el lenguaje de programación utilizado:

- Explicar la intención del programador.
- Explicar precondiciones o suposiciones.
- Clarificar código que a primera vista puede no ser claro.

Sobre las funciones auxiliares: A la hora de especificar vimos que partir el problema principal en problemas más pequeños es una buena estrategia. A la hora de programar, también es así.

- Las funciones deben ser pequeñas. Una función con demasiado código es díficil de entender y mantener.
- Encapsular comportamiento dentro de funciones auxiliares.

- Cada función debe: hacer sólo una cosa, hacerla bien, y ser el único componente del programa encargado de esa tarea particular.

Cyclomatyc Complexity: es una métrica que se utilizar para aproximar la dificultad para testear un programa. Se calcula a partir del control-flow-graph. Hay herramientas que la calculan automáticamente. Se puede fijar un máximo por proyecto que todos deben respetar.

Cognitive Complexity: es una métrica que se utiliza para aproximar la dificultad para que un ser humano entienda un programa. Se base en que:

- Code is not considered more complex when it uses shorthand that the language provides for collapsing multiple statements into one.
- Code is considered more complex for each "break in the linear flow of the code"
- Code is considered more complex when "flow breaking structures are nested"

Como con la Cyclomatic Complexity, se puede computar automáticamente a partir del control-flow graph. Existen herramientas que lo analizar y emiten warnings si se ha roto alguna regla.

Naming Conventions: indican que reglas hay que seguir para nombrar variables, constantes, funciones, archivos, tipos de datos, etc. Hay que evitar un uso inconsistente. Cada lenguaje suele tener su propia convención.

Formato vertical

- Los archivos del proyecto no deben ser demasiado grandes.
- Conceptos relacionados deben aparecer verticalmente cerca en los archivos.
- Las funciones más importantes deben estar en la parte superior, seguidas de las funciones auxiliares. Siempre la función llamada debe estar debajo de la función llamadora. Las funciones menos importantes deben estar en la parte inferior del archivo.
- Se suele equiparar un archivo de código con un artículo periodístico. Debemos poder interrumpir la lectura en cualquier momento, teniendo información acorde con la lectura realizada.

Modularización

- Cada archivo del proyecto debe tener código homogéneo, y relacionado con un concepto o grupo de conceptos coherentes.
- Los nombres de los archivos también deben ser representativos.
- Muchos lenguajes de programación dan facilidad para organizar archivos en paquetes para tener una organización en más de un nivel.
- Single responsibility principle: cada función debe tener un único motivo de cambio.

    1 Interfaz de usuario

2 Tecnología para la interfaz de usuario.

3 Lógica de negocio.

4 Consideraciones sobre los algoritmos.

5 Almacenamiento permanente.

- El código responsable de cada uno de estos aspectos debe estar separado del resto.

Al momento de escribir código:

- Desarrollo incremental: escribir bloques pequeños y testeables de funcionalidad, teniendo en todo momento una aplicación (parcialmente) funcional.

  1 Esto implica conocer en todo momento el objetivo del código que estabamos escribiendo.

  2 Si se usa un respositorio de control de versiones, el próximo commit debe estar bien definido.

- No incorporar expectativas en las estimaciones. Si vemos que los plazos no se van a poder cumplir, reaccionar rápidamente.
- Para el código más crítico, recurrir a pair programming: dos personas juntas sobre una única computadora.
- Descansar. Es importante descansar entre las sesiones de código.

Control de versiones

- 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, etc.

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.

Git 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, trabajar en una nueva versión o experimentar con el código. Se suelen dividir en: master, develop y hotfixes.

Consejos para Git

- 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.

Integración continua

- Cada vez que hay una modificación (i.e. push) en el repositorio el servidor de integración: descarga la última versión, compila el código y ejecuta el test suite.
- Reporta a los desarrolladores si falló la compilación o el test suite.
- Antes de continuar, se deben solucionar todos los problemas de integración.
- El test suite puede ser distinto del test suite que corrió localmente el programador (i.e más costoso).
- Evitar acumular mucha deuda de integración.

Clase 5

Paradigmas de programación

Existen distintos paradigmas de programación.

- Formas de pensar un algoritmo que cumple 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 = coleccion de funciones que transforman datos de entrada en un resultado.
- Los lenguajes funcionales nos dan herramientas para explicarle a la computadora cómo computar esas funciones.

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

Las ecuaciones orientadas junto con el mecanismo de reducción describen algoritmos.
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.

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.

Transparencia referencial

Es la propiedad de un lenguaje que garantiza 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).

    1 Podemos usar propiedades ya probadas para subexpresiones.

    2 El valor no depende de la historia.

    3 Valen en cualquier contexto.

Expresiones atómicas:

1 También se llaman formas normales.

2 Son las más simples, no se puede reducir más.

3 Son la forma más intuitiva de representar un valor.

4 Ejemplos: 2, False, (3, True)

Expresiones compuestas:

1 Se construyen combinando expresiones atómicas con operaciones.

2 Ejemplos: 1+1, 1==2, (4-1, True || False)

Formación de expresiones

- Algunas cadenas de símbolos no forman expresiones por problemas sintácticos (e.g: (True) o por error de tipos (e.g: 2 + False).
- Para saber si una expresión está bien formada, aplicamos: reglas sintácticas o reglas de asignación o inferencia de tipos (algoritmo de Hindley-Mikner).
- 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.

El mecanismo de evaluación en un lenguaje funcional es la reducción. Resolvamos suma (resta 2 (negar 42)) 4

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 llamada radical o redex (reducible expression). - Buscamos un redex: resta 2 (negar 42)

3 La reemplazamos 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 cómputado.

suma (resta 2 negar(42)) 4 ≡ suma (2 - (-42)) 4 ≡ suma (44) 4 ≡ 44 + 4 ≡ 48

Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy**: se reduce al 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).

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

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- Funciones totales: nunca se indefinen
- Funciones parciales: hay argumentos para los cuales se indefinen.

Tipos de datos

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

1. $\texttt{Int} = (\mathbb{Z}, \{+, -, *, \texttt{div}, \texttt{mod}\})$ es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
2. $\texttt{Float} = (\mathbb{Q}, \{+, -, *, /\})$ es el tipo de datos que representa a los racionales, con la aritmética de punto flotante.
3. $\texttt{Char} = (\{\texttt{'a'}, \texttt{'A'}, \texttt{'1'}, \texttt{'?'}\}, \{\texttt{ord}, \texttt{chr}, \texttt{isUpper}, \texttt{toUpper}\})$ es el tipo de datos que representan los caracteres.
4. $\texttt{Bool} = (\{\texttt{True}, \texttt{False}\}, \{\texttt{\&\&}, \texttt{||}, \texttt{not}\})$ 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 estricticamente necesario hacerlo (Haskell puede inferir el tipo), pero suele ser una buena práctica.

Aplicación de funciones

En programación funcional (como en matemática) las funciones son elementos (valores).
Notación: f :: $T1 \longrightarrow T2 \longrightarrow T3 \longrightarrow ... \longrightarrow T_n$

- Una función es un valor
- La operación básica que podemos realizar con ese valor es la aplicación. Aplicar la funcón a un elemento para obtener un resultado.
- Sintácticamente, la aplicación se escribe como una yuxtaposición (la función seguida de su parámetro).

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.

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

Clases de tipos

- Conjunto de tipos a los que se le pueden aplicar ciertas funciones.
- Un tipo puede pertenecer a distintas clases. Por ejemplo, los Float son números (Num) con orden (Ord) de punto flotante (Floating), etc.
- 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, ... },{ (==), (/=)
   })
```

Tuplas

- Dados tipos $A_1, ..., A_k$, el tipo k-upla $(A_1, ..., A_k)$ es el conjunto de las k-uplas $(v_1, ..., v_k)$ donde v_i es de tipo A_i
- Las funciones de acceso a los valores de un par en Prelude: fst(a,b) devuelve a y snd(a,b) devuelve b.

Currificación: es una notación donde en vez de pasar multiples parametros con sus distintos tipos le asignamos una k-upla y especificamos los tipos dentro de ella.

Funciones binarias

- Notación prefija: función antes de los argumentos (e.g: suma x y).
- Notación infija: función entre argumentos (e.g: x+y).
- 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: (+*)).
- Una función binaria f puede ser usada de forma infija escribiendo 'f'.

Clase 6

Recursión

"Recursión es para el paradigma funcional lo que es la iteración para el paradigma imperativo"

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.

Clase 7

Listas

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]
Las listas tienen dos "pintas":

- **Lista vacía:** []
- **Lista no vacía:** algo : lista

Definición de tipo usando type:

- **type** es la palabra reservada del lenguaje para definir un nuevo tipo. Por ejemplo, si quiero definir el tipo Set en Haskell puedo escribir **type Set a = [a]**.
- Si bien internamente Set es una lista, la idea es tratarlo como si fuera un conjunto (es un contrato entre programadores).

Clase 8

Validación y verificación

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:** 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.

Los objetivos principales de la validación y verificación (VV) son:

- Descubrir defectos en el sistema.
- Asegurar que el software respeta su especificación.
- Determinar si satisface las necesidades de sus usuarios.

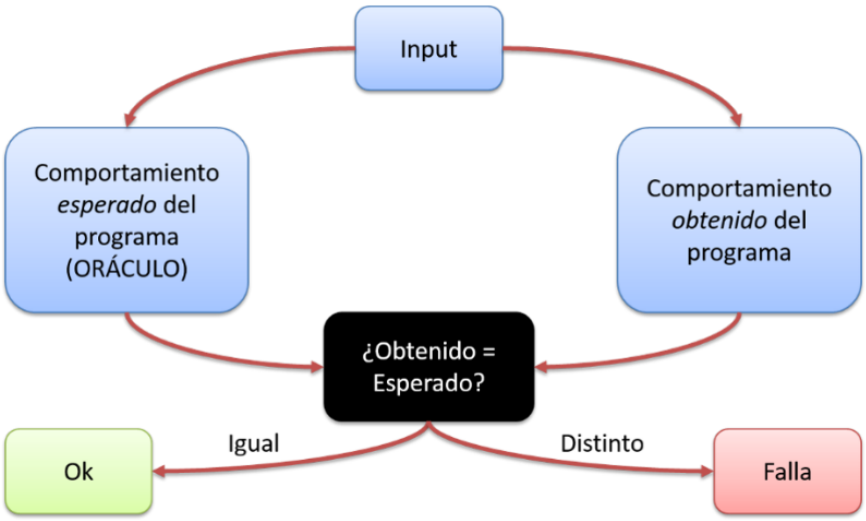
La VV deberían establacer la confianza de que el software es adecuado a su propósito. Sin embargo, esto no significa que esté completamente libre de defectos. Sino que debe ser lo suficientemente bueno para su uso previsto y el tipo de uso determinará el grado de confianza que se necesita.
Una forma de realizar tareas de VV es a través de análisis. En particular para el código, tenemos análisis estático y análisis dinámico.

- **Dinamica:** trata con ejecutar y observar el comportamiento de un producto.
- **Estática:** trata con el análisis de una representación estática del sistema

Testing

Testing es el proceso de ejecutar un producto para verificar que satisface los requerimientos (en nuestro caso, la especificación). Identificar las diferencias entre el comportamiento real y el comportamiento esperado. El objetivo del testing es encontrar defectos en el software.

¿Cómo hacer testing?



Niveles de Test

- Tests 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.

Conceptos

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

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 chequear? La condición que me indica la post-condición (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.

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"
- 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, y suficientemente pequeño para poder ejecutar el proceso con cada elemento del conjunto y minimizar el costo del testing.

Clase 9

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.

Ejemplo de como usar este método:

- 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 espeficado que sucede en este caso.	Como la elección No es un ERROR, no importa el valor
Caso 2: E no pertenece	-	No	ERROR: no está espeficado que sucede en este caso.	Como la elección No es un ERROR, no importa el valor
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, unicos o restricciones entre sus posibles valores: porque descartan casos hacia la derecha.

Testing de caja negra: los datos de test se derivan a partir de la descripción del programa sin conocer su implementación.

Clase 10

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:
    1. Posiciones de memoria
    2. Cambian explicitamente de valor a lo largo de la ejecución de un programa
    3. Pérdida de la transparencia referencial
- Nueva operación: la asignación. Cambia 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
    1. Secuencia de valores de un tipo (como las listas)
    2. Longitud prefijada
    3. Acceso directo a una posición (en las listas, hay que acceder primero a las anteriores)

Python

Características:

- Es un lenguaje interpretado.
- Tiene tipado dinámico. Una variable puede tomar valores de distintos tipos.
- Es fuertemente tipado. Una vez que una variable toma un tipo, ya no puede cambiar.

Además del tipado dinámico, hay que saber que también está el tipado estático. En este la comprobación de tipificación se realiza durante la compilación (y no durante la ejecución).

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).

Instrucciones:

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

La asignación

- Es la operación fundamental para modificar el valor de una variable.
- Es una operación asimétrica. Del lado izquierdo debe ir una variable u otra expresión que represente una posición de memoria. Por otro lado, del lado derecho debe ir una expresión del mismo tipo que la variable. Puede ser una constante, otra variable o una función aplicada a argumentos.
- Efecto de la asignación: se evalúa la expresión de la derecha y se obtiene un valor. Luego, ese valor se copia en el espacio de memoria de la variable. El resto de la memoria no cambia.

La instrucción return

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

Transformación de estados

Llamos 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.
- Después de ejecutar la última instrucción.

Veremos la ejecución de un programa como una sucesión de datos. 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.

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.

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

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. Algunes técnicas de verificación estática utilizan estos recursos. 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. Coloca en la posición de memoria del argumento de entrada el valor de la expresión usada en la invocación. Se llama también pasaje por copia. 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. 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.

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.

Funciones y Procedimientos: ambos ejecutan un grupo de sentencias. Las funciones devuelven un valor y los procedimientos no.

Existen tres tipos de pasajes de parametros

- **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 pasaba 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.

problema *nombre*(*parámetros*) : tipo de dato del resultado (*opcional*){
requiere *etiqueta*: { condiciones sobre los parámetros de entrada }
modifica: parametros 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 (inicialmente 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 podremos especificar programas que no devuelvan resultados, sino que sólo modifiquen sus parámetros.

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.
- **Tipo 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.
Ejemplo de pasaje de argumentos en Python:

```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']
```

Diferencias entre in, out e inout con una misma función:

problema *duplicar*(*in* x : $seq\langle T \rangle$) : $seq\langle T \rangle$ {
  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.}
}

```python
def duplicar(x: list) -> list:
  y: list = x.copy()
  y *= 2
  return y
```

problema *duplicar*(*inout* x : $seq\langle T \rangle$){
  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.}
}

```python
def duplicar(x: list):
  x *= 2
```

problema *duplicar*(*in* $x : seq\langle T \rangle$, *out* $y : seq\langle T \rangle$){
  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
```

Clase 11

Python

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.

Identación

- La identanción es a un lenguaje de programación, lo que la sangría al lenguaje humano escrito.
- En ciertos lenguajes de programación, la identación 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.

Alcance, ámbito o scope de las variables

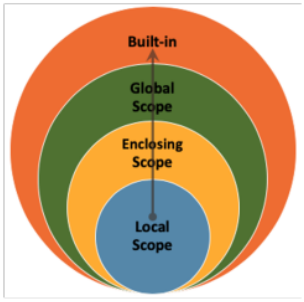
El alcance de una variable, se refiere al ámbito o espacio donde una variable es reconocida. Una variable sólo será válida dentro del bloque (función/procedimiento) donde fue declarada. Al terminar el bloque, la variable se destruye. Estas variables se denominan **variables locales**. Las variables declaradas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

En Python se puede modificar a una variable global desde dentro de una función si la referenciamos con **global variable-name** y luego la modificamos.

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).

```python
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)
```



Las referencias también tienen su scope: al pasar un parámetro por referencia, esta referencia vivirá dentro del scope de la función.

Condicionales

if (B):
uno
else:
dos

- B tiene que ser una expresión booleana. Se llama guarda.
- uno y dos son bloques de instrucciones.

Veamos al 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.

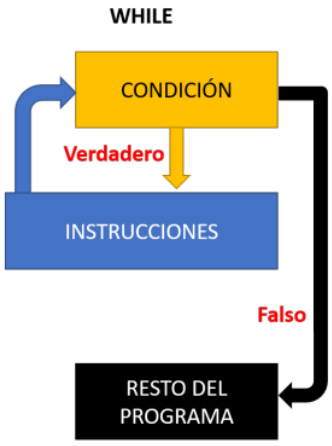
```
  # estado P_if
if (B):
      # estado P_uno
    uno
      # estado Q_uno
else:
      # estado P_dos
    dos
      # estado Q_dos
  # estado Q_if
  # (B ∧ estado Q_uno) ∨ (¬B ∧ estado Q_dos)
  # Después del IF, se cumplió B y Q_uno o, no se cumplió B y Q_dos
```

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.
- Existe diferentes de iteración, los más conocidos son: while, do while y for.

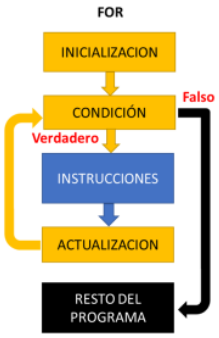
Sintaxis del While:

```
while(condición de finalización):
    ...
    ...
    Bloque de Instrucciones...
    Dentro del while
    ...
    ...

Bloque de Instrucciones...
FUERA del while
```



Sintaxis del For:

```
for variable_contadora in range(valor_inicial, valor_final, tamaño_paso):
    ...
    ...
    Bloque de Instrucciones...
    ...
    ...

Bloque fuera del ciclo...
```



La instrucción **Break** permite romper la ejecución de un ciclo. No se fomenta su porque le quita declaratividad al código. Desde el punto de vista de analizar la correctitud de un programa, traerá problemas.

Ciclos y transformación de estados

- 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?

Clase 12

Arreglos y listas

Arreglos

- Secuencias de una cantidad fija de valores del mismo tipo.
- Se declaran con un nombre y 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.

Ejemplo de arreglo:

- 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					

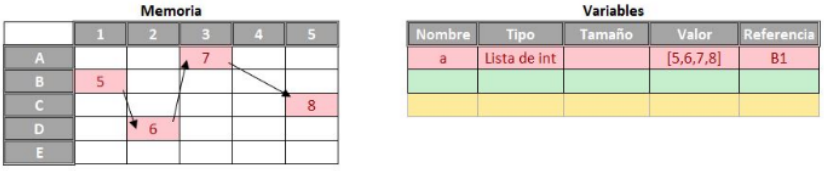
Variables

Nombre	Tipo	Tamaño	Valor	Referencia
a	Array de Int	4	[5,6,7,8]	B1

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.

Ejemplo de lista:



Arreglos y listas en Python

Arreglos en 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.

```
1 from array import *
2 a: array = array('i', [20,30,10])
```

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 coincida 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

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.
- Las operaciones básicas sobre listas son las mismas que las de los arreglos.
- Sin embargo, hay operaciones que son exclusivas de los arreglos (ej:Buffer Info).
- Una matriz en Python se piensa como una lista de listas. Cada elemento de la lista representa una fila.

Iterable: en Python aquellas variables cuyo tipo de dato sea iterable pueden ser recorridas con un for.

Tipos Abstractos de Datos

Un Tipo Abstracto de Dato (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.

Clase 13

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

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.

Manejo de Archivos

El manejo de archivos, también puede pensarse mediante la abstracción que nos brindan los TADs. Queremos operaciones que nos permitan abrir un archivo, leer sus líneas y cerrarlo.

```
# 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()
```

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 especıfica de caracteres o bytes del archivo. Si no se especifica el tamaño, se lee el contenido completo.
- readline(): Lee y devuelve la siguiente lınea del archivo.
- readlines(): Lee todas las lıneas 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 lıneas en el archivo. Cada lınea debe terminar con un salto de lınea explıcito.

Application Programming Interface (API)

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 números de problemas a resolver como ser: conexiones de red, uso de protocolos, manejo de errores, transformaciones de datos, etc (y son muchos etcs).

Clase 14

Testing de caja blanca

En el Testing de caja blanca los datos de test se derivan a partir de la estructura interna del programa. Los criterios de caja blanca permiten identifcar casos especiales según el flujo de control de la aplicación.

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:

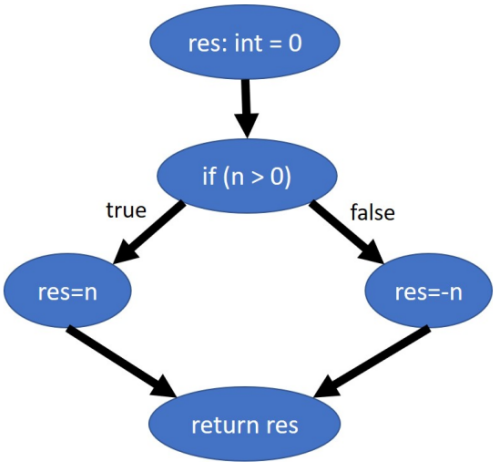
- Complemento al Test de Caja Negra: permite encontrar más casos o definir casos más específicos.
- Como **criterio de adecuación** del Test de Caja Negra: brinda herramientas que nos ayudar a determinar cuan bueno o confiable resultaron ser los test suites definidos. En este contexto hablaremos de **Criterios de Cubrimiento**.

Control-Flow Graph

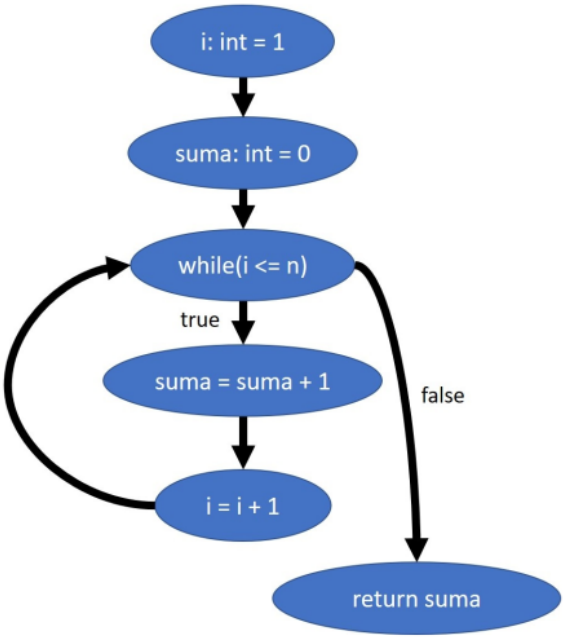
- El control flow graph (CFG) de un programa es sólo una **representación gráfica del programa**.
- El CFG es independiente de las entradas (su definición es estática).
- Se usa (entre otras cosas) para definir criterios de adecuación para test suites.
- Cuanto más partes son ejercitadas (cubiertas), mayores las chances de un test de descubrir una falla.
- partes pueden ser: nodos, arcos, caminos, decisiones ...

Ejemplos de Control Flow Patterns:

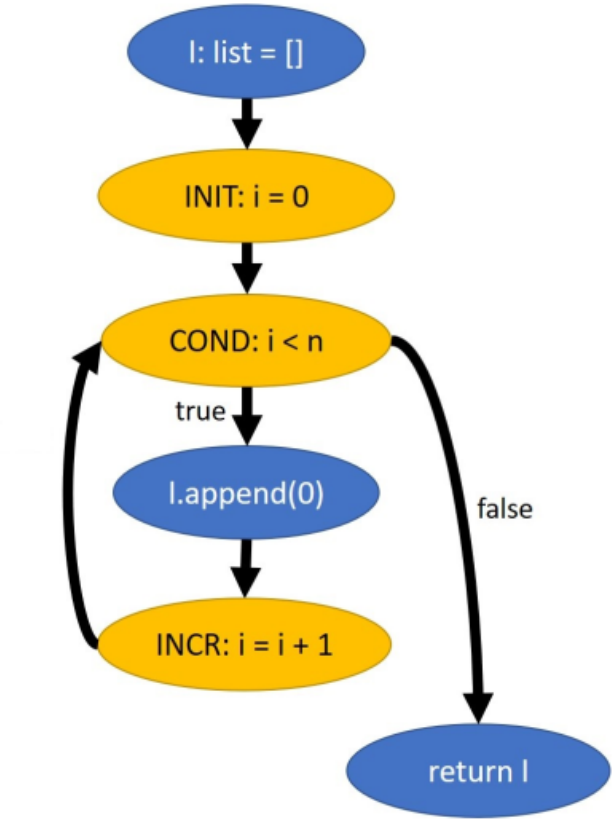
CFG If-Else



CFG While



CFG For



Criterios de Adecuación

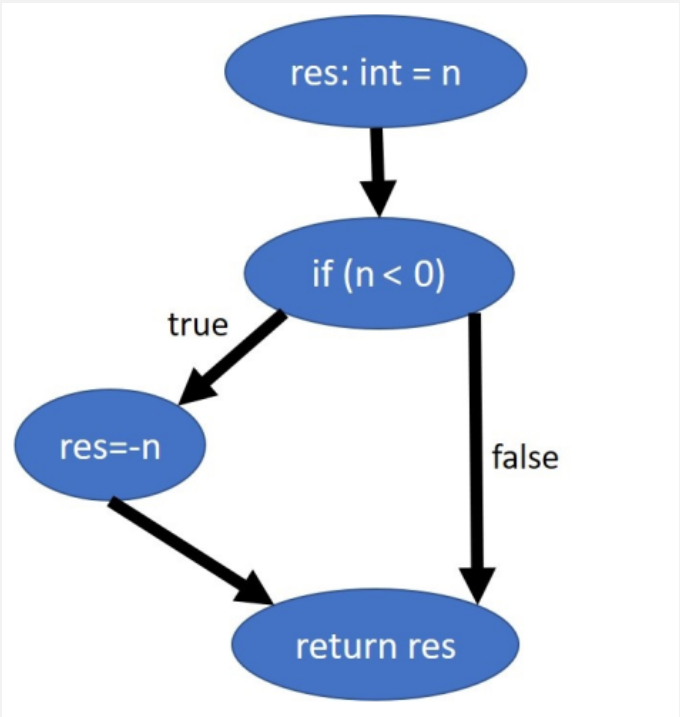
Un criterio de adecuación de test es un predicado que toma un valor de verdad para una tupla <programa, test suite>.

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 una sentencia sólo puede ser revelado ejecutando el defecto.
- Cobertura: $\frac{\text{cantidad nodos ejercitados}}{\text{cantidad nodos}}$

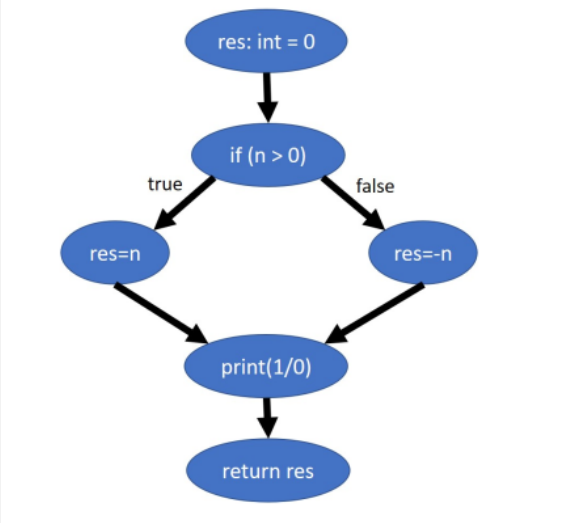
Cubrimiento de Arcos

- Criterio de Adecuación: todo arco(flecha) 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. Muestro un ejemplo en el que se puede construir un test suite que cubra todos los nodos pero que no cubra todos los arcos.

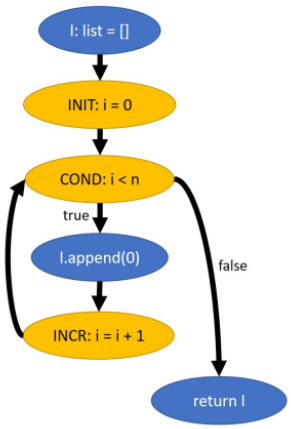


Cubrimiento de Decisiones (o Branches)

- 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 ejercite.
- Cobertura: $\frac{\text{cantidad decisiones ejercitadas}}{\text{cantidad decisiones}}$
- El cubrimiento de decisiones no implica el cubrimiento de los arcos del CFG. Muestro un ejemplo en el que se puede construir un test suite que cubra todos los branches pero que no cubra todos los arcos.



Ejemplo que ilustra lo anterior:



- ¿Cuántos nodos (sentencias) hay? 6
- ¿Cuántos arcos (flechas) hay? 6
- ¿Cuántas decisiones (arcos True y arcos False) hay? 2

Cubrimiento de Condiciones Básicas

- Una condición básica es una fórmula atómica (indivisible) que componen una decisión.
- 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: $\frac{\text{cantidad de valores evaluados en cada condición}}{2 * \text{cantidad condiciones básicas}}$

Observación: el cubrimiento de decisiones no implica el cubrimiento de Condiciones Básicas.

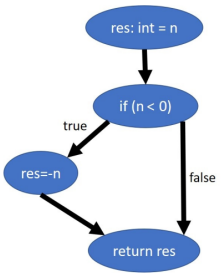
- **Observación** Branch coverage no implica cubrimiento de Condiciones Básicas
  - Ejemplo: **if(a && b)**
  - Un test suite que ejercita solo $a = true, b = true$ y $a = false, b = true$ logra cubrir ambos branches de **if(a && b)**
  - Pero: no alcanza cubrimiento de decisiones básica ya que falta $b = false$

El criterio de cubrimiento de Branches y condiciones básicas necesita 100% de cobertura de branches y 100% de cobertura de condiciones básicas.

Cubrimiento de Caminos

- Criterio de Adecuación: cada camino en el CFG debe ser transitado por al menos un test case.
- Cobertura: $\frac{\text{cantidad camino transitados}}{\text{cantidad total de caminos}}$
- Ejemplo 1:

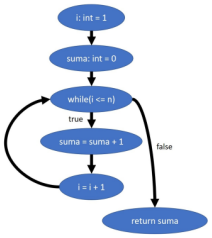
Sea el siguiente CFG:



¿Cuántos caminos hay en este CFG? 2

- Ejemplo 2:

Sea el siguiente CFG:



¿Cuántos caminos hay en este CFG? La cantidad de caminos no está acotada (∞)

En todos los criterios de adecuación estructurales se usa el CFG para obtener una métrica del test suite.

- **Sentencias:** cubrir todos 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 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.

Python

Funciones Python

Acontinuación voy a listar las funciones y operadores que vimos en clase.

Observación: me limité a las funciones de la teórica más lo que se vio en la práctica de la noche.

1. abs()
2. append(x)
3. break
4. close()
5. count(x)
6. del diccionario[clave]
7. diccionario.items()
8. diccionario.keys()
9. format()
10. for elemento in lista
11. for i in range(i,n,k)
12. if
13. index(x)
14. insert(p,x)
15. insert(posicion,elemento)
16. int()
17. input()
18. len()
19. lista[i]
20. open(nombre-archivo, modo-lectura, encoding)
21. pop()
22. print ()
23. readlines()
24. remove(x)
25. range(i,n,k)

26. str()
27. truncate()
28. while
29. write()
30. ==
31. !=
32. \n para hacer un salto de linea
33. , para imprimir variables
34. + para concatenar str
35. * para duplicar str o lista
36. [::-1] da vuelta una palabra o lista

Haskell

Funciones Haskell

Acontinuación voy a listar las funciones y operadores que vimos en clase.

Observación: me limité a las funciones de la teórica más lo que se vio en la práctica de la noche.

1. div n m (devuelve el cociente de n/m)
2. fromIntegral (transforma un entero en un float)
3. fst (sirve solo para tupla)
4. head lista
5. mod n m (devuelve el resto de n/m)
6. snd (sirve solo para tupla)
7. tail lista
8. type NombreTipo = t (t es un tipo de dato)
9. (x:xs)
10. &&
11. ==
12. /=
13. ∧
14. >=
15. <=
16. ||