Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2017

Departamento de Computación - FCEyN - UBA

Especificación - clase 1

Introducción a la especificación de problemas
Lógica proposicional

Algoritmos y Estructuras de Datos I

Objetivo: Aprender a programar en lenguajes imperativos.

"Computadora, mata a Flanders..."

Algoritmos y Estructuras de Datos I

Objetivo: Aprender a programar en lenguajes imperativos.

- Especificar problemas.
  - Describirlos en lenguaje formal.
- Escribir programas.
  - En esta materia nos concentramos en programas para tratamiento de secuencias.
- Razonar acerca de estos programas.
  - Obtener una visión abstracta del cómputo.
  - Definir un manejo simbólico y herramientas para demostrar propiedades de nuestros programas.
  - Probar corrección de un programa respecto de su especificación.

Algoritmos y Estructuras de Datos I

Contenidos

1. Especificación de problemas
   - Lenguaje formal cercano a la lógica proposicional
2. Programación imperativa
   - Lenguaje de programación imperativo
   - Correctitud de un programa usando precondición más débil
3. Algoritmos sobre secuencias y matrices

Algoritmos y Estructuras de Datos I

Régimen de aprobación

- Parciales (Turno Tarde - Se aprueba con 60)
  - 2 parciales
  - 2 recuperatorios (al final de la cursada)
- Trabajos prácticos
  - 2 entregas (aprobado o desaprobado)
  - 2 recuperatorios (cada uno a continuación)
  - Grupos de 4 alumnos (inscripción presencial+LU con Algebra Firmada)
- Examen final o un coloquio (en caso de tener dado el final de Álgebra I al finalizar la cursada)

Calificación final: p_1 promedio de la cursada (parciales) y p_2 la nota del coloquio , donde:

$$cursada = \frac{p_1 + p_2}{2}$$

www.dc.uba.ar/algo1-tm

algo1-tm-alu@dc.uba.ar

algo1-tm-doc@dc.uba.ar

Bibliografía

- ¿Cómo pensamos y programamos?
  - **The Science of Programming**. David Gries. Springer Verlag, 1981
  - **Reasoned programming**. K. Broda, S. Eisenbach, H. Khoshnevisan, S. Vickers. Prentice-Hall, 1994
- Testing de Programas
  - **Software Testing and Analysis: Process, Principles and Techniques**. M. Pezze, M. Young, Wiley, 2007.
- Referencia del lenguaje que usaremos
  - **The C++ Programming Language, 4th Edition**. B. Stroustrup. Addison-Wesley Professional, 2003

¿Qué es una computadora?

- Una **computadora** es una máquina que procesa información automáticamente de acuerdo con un programa almacenado.
  1. Es una máquina.
  2. Su función es procesar información, y estos términos deben entenderse en sentido amplio.
  3. El procesamiento se realiza en forma automática.
  4. El procesamiento se realiza siguiendo un programa.
  5. 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.

  1. Es la descripción de los pasos a dar.
  2. Especifica una sucesión de pasos primitivos.
  3. El objetivo es resolver un problema.
  4. Un algoritmo típicamente trabaja a partir de datos de entrada.

Ejemplo: Un Algoritmo

- **Problema:** Encontrar todos los números primos menores que un número natural dado n
- **Algoritmo:** Criba de Eratóstenes (276 AC - 194 AC)

  [1.] Escriba todos los números naturales anteriores a n
  [2.] Para $i \in \mathbb{Z}$ desde 2 hasta $\lfloor \sqrt{n} \rfloor$
  [1.] Si i no ha sido marcado, entonces:
  [1.] Para $j \in \mathbb{Z}$ desde i hasta $[n/i]$ haga lo siguiente:
  [1.] Ponga una marca en el número $i \times j$

¿Qué es un programa?

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

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

Ejemplo: Un Programa (en Haskell)

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

```
erastotenes :: Int → [Int]
erastotenes n = erastotenes2 [x|x ← [2..n]] 0

erastotenes2 :: [Int] → Int → [Int]
erastotenes2 lista n
 | n == length lista-1 = lista
 |otherwise=erastotenes2
           [x|x ←lista ,(x`mod`lista !!n)/=0||x==lista !!n] (n+1)
```

Especificación, algoritmo, programa

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

Etapas en el desarrollo de programas

1. **Especificación**
   Definición formal del problema.
2. **Diseño general**
   Determinar las componentes de la solución.
3. **Diseño de algoritmos**
   Dar algoritmos para cada componente.
4. **Programación y validación**
   Implementar los algoritmos y asegurarse de que la implementación es correcta.
5. **Instrumentación y mantenimiento**
   Puesta en práctica del programa implementado. Corrección de errores e introducción de cambios para responder a nuevos requerimientos.

1. Especificación

- El planteo inicial del problema puede ser vago y ambiguo.
- Al especificar damos una descripción clara y precisa en lenguaje formal, por ejemplo, el lenguaje de la lógica matemática.
- Por ejemplo, "calcular la edad de una persona" es una definición imprecisa del problema (¿por qué?).
- +Info: AED1, AED2.

2. Diseño general

- ¿Varios programas o uno muy complejo?
- ¿Cómo dividirlo en partes?
- ¿Distintas partes en distintas máquinas?
- ¿Programas ya hechos con los que interactuar?
- +Info: AED2, IS1, IS2.

3. Diseño de algoritmos

- Escribir un algoritmo para dar solución a cada componente en la que se subdividió la solución.
- El objetivo más importante es que los algoritmos propuestos sean correctos.
- Objetivos secundarios (o no tanto!):
  1. Tiempo de ejecución.
  2. Uso de memoria.
  3. Uso de otros recursos.
  4. etc.
- +Info: AED1, AED2, AED3.

4. Programación y validación

- Traducimos el algoritmo a un lenguaje adecuado para ser ejecutado por una computadora.
  - Lenguajes naturales vs lenguajes formales.
- Tomamos las acciones necesarias para asegurarnos de que la implementación reproduce adecuadamente el algoritmo propuesto.
- +Info: TL, IS1, IS2.

5. Instrumentación y mantenimiento

- Poner en práctica el programa en el entorno final.
- Si tiempo después encontramos errores o cambian los requerimientos, es necesario volver a las etapas anteriores para revisar el programa.
- +Info: IS1, IS2.

Especificación de problemas

- Una especificación es un contrato que define qué se debe resolver y qué propiedades debe tener la solución.
  1. Define el qué y no el cómo.
- Además de cumplir un rol "contractual", la especificación del problema es insumo para las actividades de ...
  1. testing,
  2. verificación formal de corrección,
  3. derivación formal (construir un programa a partir de la especificación).

Parámetros y tipos de datos

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

Contratos

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

Partes de una especificación (contrato)

1. Encabezado
2. Precondición o cláusula "requiere"
   - Condición sobre los argumentos, que el programador da por cierta.
   - Especifica lo que requiere la función para hacer su tarea.
   - Por ejemplo: "el valor de entrada es un real no negativo"
3. Postcondición o cláusula "asegura"
   - Condición sobre el resultado, que debe ser cumplida por el programador siempre y cuando el usuario haya cumplido la precondición.
   - Especifica lo que la función asegura que se va a cumplir después de llamarla (si se cumplía la precondición).
   - Por ejemplo: "la salida es la raíz cuadrada del valor de entrada"

El contrato

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

¿ Por qué escribir la especificación del problema?

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

¿Qué ocurre cuando hay errores en los programas?

El 31/12/2008 todos los dispositivos MS ZUNE se colgaron al mismo tiempo. ¿Por qué?


```
year = ORIGINYEAR; /* = 1980 */

while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days = days - 366;
            year = year + 1;
        }
    } else {
        days = days - 365;
        year = year + 1;
    }
}
```


Le Bug: El caso del Ariane 5

- Máximo exponente de los logros del programa espacial europeo
- Vuelo Inaugural: 4 de Junio 1996
- Costo de carga (solamente): U$S 500 millones
- Reusa programas del Ariane 4.

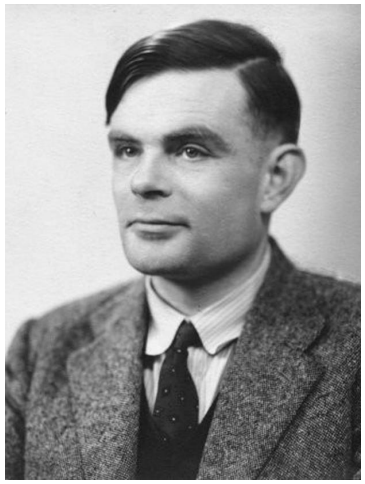
Verificación (Automática) de Programas

Si tengo una especificación formal F, y un programa P que implementa la especificación, puedo intentar probar matemáticamente (como si fuera un teorema) que P implementa correctamente a F.

- Lo puedo probar manualmente (como vamos a ver en AED1)
- Lo puedo probar semi-automáticamente (KeY, PVS, ISABELLE, Coq, etc.)
- Lo puedo probar automáticamente (CodeContracts, AutoProof, VCC, JPF, SDV, etc.)

¿Por qué no probarlo automáticamente **siempre**?

Die Entscheidungsproblem (aka The Halting Problem)



- El *Halting Problem* es el problema de determinar si un programa P termina o no.
- El problema de chequear si P implementa correctamente a F puede codificarse como un programa
  - If "P implementa correctamente a F" Then Return
  - Else while true $\{\}$.
- En 1936 Alan Turing demostró que el Halting Problem era indecidible.

+Info: LyC

Verificación Automática de Programas

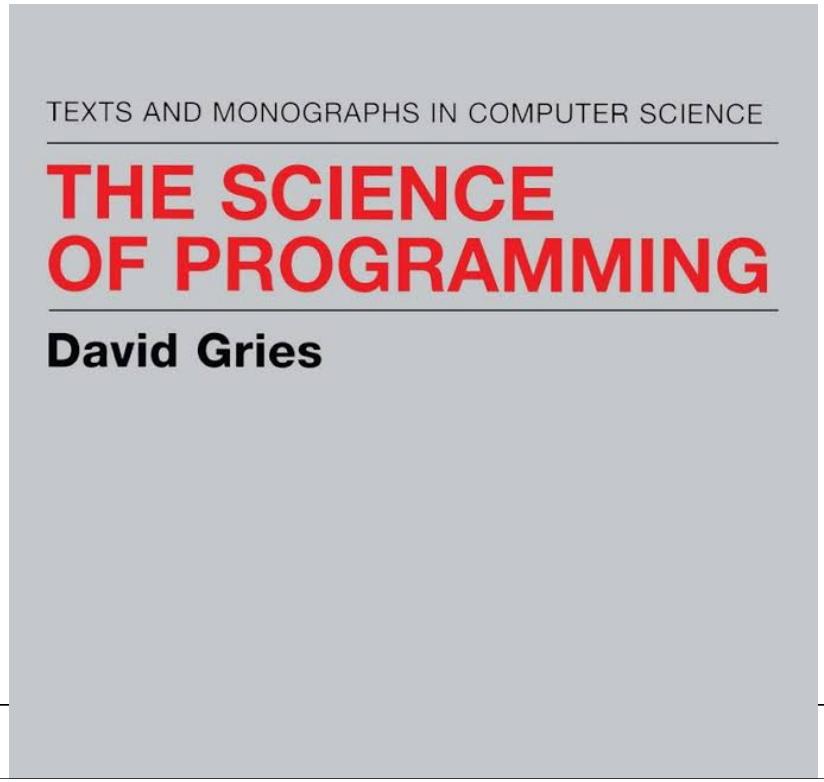
El Halting Problem no significa que para *algunos* programas sí se pueda demostrar automáticamente terminación y correctitud.
Por ejemplo `triangle` retorna un entero indicando si el triángulo es equilatero, escaleno o isósceles.

```
int triangle(int a, int b, int c)
  _(requires a>0)
  _(requires b>0)
  _(requires c>0)
  _(ensures (a=b && b=c) ⟹ result=1 ) // equilatero
  _(ensures (a=b && b!=c) ⟹ result=2 ) // isosceles
  _(ensures (a!=b && b=c) ⟹ result=2 ) // isosceles
  _(ensures (a!=b && a=c) ⟹ result=2 ) // isosceles
  _(ensures (a!=b && b!=c && a!=c) ⟹ result=3 ) // escaleno
{
   // programa
}
```


Verificación Automática de Programas

- Si el Verificador dice que P es correcto, entonces existe una demostración matemática que P es correcto.
- Equivale a testear el programa con todos sus inputs ($2^{32} \times 2^{32} \times 2^{32} = 2^{96}$ inputs posibles).
- Varias empresas de software ya emplean verificación como parte de sus procesos de desarrollo (Microsoft, Mozilla, NASA, Eiffel, etc.)
- La Verificación Automática es Limitada (Halting Problem)
- En AED1 vamos a ver como construir (y verificar) programas de tamaño pequeño.

Lenguaje de especificación



Definición de un procedimiento

proc *nombre*(*parámetros*){
  Pre { P }
  Post { Q }
}

- *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, salida, entrada y salida)
  - Nombre del parámetro
  - Tipo de datos del parámetro
- P y Q son predicados, denominados la precondición y la postcondición del procedimiento.

Ejemplos

proc *rcuad*(in $x : \mathbb{R}$, out *result* : $\mathbb{R}$) {
  Pre $\{x \geq 0\}$
  Post $\{result * result = x\}$
}

proc *suma*(in $x : \mathbb{Z}$, in $y : \mathbb{Z}$, out *result* : $\mathbb{Z}$) {
  Pre $\{True\}$
  Post $\{result = x + y\}$
}

proc *resta*(in $x : \mathbb{Z}$, in $y : \mathbb{Z}$, out *result* : $\mathbb{Z}$) {
  Pre $\{True\}$
  Post $\{result = x - y\}$
}

proc *cualquieramayor*(in $x : \mathbb{Z}$, out *result* : $\mathbb{Z}$) {
  Pre $\{True\}$
  Post $\{result > x\}$
}

Interpretando una especificación

- proc *rcuad*(in $x : \mathbb{R}$, out *result* : $\mathbb{R}$) {
  Pre $\{x \geq 0\}$
  Post $\{result * result = x\}$
  }
- ¿Qué significa esta especificación?
- Se especifica que si el programa `rcuad` se comienza a ejecutar en un estado que cumple $x \geq 0$, entonces es cierto que:
  1. el programa **termina**, y
  2. el estado final cumple $result * result = x$.

Argumentos que se modifican (inout)

Problema: Incrementar en 1 el argumento de entrada.

- Alternativa sin modificar la entrada (usual).

  proc incremento(in $a : \mathbb{Z}$, out $result : \mathbb{Z}$){
  Pre $\{True\}$
  Post $\{result = a + 1\}$
  }

- Alternativa que modifica la entrada: usamos el mismo argumento para la entrada y para la salida.

  problema incremento-modificando(inout $a : \mathbb{Z}$){
  Pre $\{a = a_0\}$
  Post $\{a = a_0 + 1\}$
  }

Otro ejemplo

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

proc *cociente*(in *dividendo* : $\mathbb{Z}$, in *divisor* : $\mathbb{Z}$, out *result* : $\mathbb{Z}$) {
  Pre $\{divisor > 0\}$
  Post {
    $result * divisor \leq dividendo$
    $\wedge (result + 1) * divisor > dividendo$
  }
}

Qué sucede si ejecutamos con ...

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

Más ejemplos

Problema: Calcular el cociente y el resto entre dos enteros.

- Retornamos el cociente y resto en dos argumentos de tipo "out" distintos.

proc cocienteyResto(in $a : \mathbb{Z}$, in $b : \mathbb{Z}$, out $r : \mathbb{Z}$, out $q : \mathbb{Z}$) {
  Pre $\{ b > 0 \}$
  Post $\{ a = q * b + r \wedge 0 \leq r < b \}$
}

Pasaje de Parámetros

in, out, inout

- Parámetros de entrada (**in**): Si se invoca el procedimiento con el argumento **c** para un parámetro de este tipo, entonces se copia el valor **c** antes de iniciar la ejecución
- Parámetros de salida (**out**): Al finalizar la ejecución del procedimiento se copia el valor al parámetro pasado. No se inicializan, y no se puede hablar de estos parámetros en la precondición.
- Parámetros de entrada-salida (**inout**): Es un parámetro que es a la vez de entrada (se copia el valor del argumento al inicio), como de salida (se copia el valor de la variable al argumento). El efecto final es que la ejecución del procedimiento modifica el valor del parámetro.
- Todos los parámetros con atributo **in** (incluso **inout**) están inicializados

Sobre-especificación

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

  proc *distinto*(in $x : \mathbb{Z}$, out *result* : $\mathbb{Z}$) {
  Pre {*True*}
  Post {$result = x + 1$}
  }
- ... en lugar de:

  proc *distinto*(in $x : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre{*True*}
  Post {$result \neq x$}
  }

Sub-especificación

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

  proc *distinto*(in $x : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre {$x > 0$}
  Post {$result \neq x$}
  }

  ... en vez de:

  proc *distinto*(in $x : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre {*True*}
  Post {$result \neq x$}
  }

Especificar los siguientes problemas

- Calcular la función **signo**.

  proc *signo*(in $x : \mathbb{R}$, out *result* : $\mathbb{Z}$){
  Pre {*True*}
  Post{$(result = 0 \wedge x = 0)$
  $\vee (result = -1 \wedge x < 0) \vee (result = 1 \wedge x > 0)$}
  }
- Calcular el área de un triángulo rectángulo.

  proc *area*(in $x : \mathbb{R}$, in $y : \mathbb{R}$, out *result* : $\mathbb{R}$) {
  Pre {$x > 0 \wedge y > 0$}
  Post {$result = x * y/2$}
  }

Especificar los siguientes problemas

- Encontrar una raíz de un polinomio de grado 2.

  proc *raiz*(in $a : \mathbb{R}$, in $b : \mathbb{R}$, in $c : \mathbb{R}$, out *result* : $\mathbb{R}$){
  Pre {*True*}
  Post {$a * result * result + b * result + c = 0$}
  }
- Está bien especificado? **No!** Existen datos de entrada que cumplen la precondición para los cuales no se puede escribir un programa que termine cumpliendo la postcondición.
- proc *raiz*(in $a : \mathbb{R}$, in $b : \mathbb{R}$, in $c : \mathbb{R}$, out *result* : $\mathbb{R}$){
  Pre {$a \neq 0 \wedge b * b \geq 4 * a * c$}
  Post {$a * result * result + b * result + c = 0$}
  }

Último ejemplo (por hoy)

- Dada una hora, minutos y segundos correspondientes a una hora de la mañana, dar la cantidad de segundos que faltan hasta el mediodía.
- proc *hastamediodia*(in $h : \mathbb{Z}$, in $m : \mathbb{Z}$, in $s : \mathbb{Z}$, out *result* : $\mathbb{Z}$) {
  Pre $\{0 \leq h < 12 \wedge 0 \leq m < 60 \wedge 0 \leq s < 60\}$
  Post$\{result + (h * 60 + m) * 60 + s = 12 * 60 * 60\}$
  }

Bibliografía

- Vickers - Reasoned Programming
  - Introduction (Especificación vs. Implementación, etc.)
- David Gries - The Science of Programming
  - Chapter 12.1 - Calls with Value and Result Parameters (parámetros, precondición, postcondición)