Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2019

Departamento de Computación - FCEyN - UBA

Introducción a la especificación de problemas
Lógica proposicional

Algoritmos y Estructuras de Datos I

Objetivo: Aprender a programar en lenguajes imperativos.

- Especificar problemas.
  - Describirlos en lenguaje formal.
- Pensar algoritmos para resolver los problemas.
  - En esta materia nos concentramos en programas para tratamiento de secuencias.
- Razonar acerca de estos algoritmos.
  - 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

Régimen de aprobación

- Parciales
  - Dos parciales (LU con Algebra I firmada)
  - Dos recuperatorios (al final de la cursada)
- Trabajos prácticos
  - Dos entregas
  - Dos recuperatorios
  - Grupos de 2 (dos) alumnos
- Un coloquio al finalizar la cursada (en caso de tener aprobado el final de Álgebra I) y sino un examen final.

Página de la materia: `https://campus.exactas.uba.ar`
Buscar:

```
Dto de Computacion
 |-> 2019
      |-> 2do Cuatrimestre
           |-> Algoritmos y Estructura de Datos 1 -
               Turno Mañana
```

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 desde 2 hasta a n
  [2.] Para $i \in \mathbb{Z}$ desde 2 hasta $\lfloor \sqrt{n} \rfloor$
  Si i no ha sido marcado,
  Entonces Para $j \in \mathbb{Z}$ desde i hasta $\frac{n}{i}$ haga lo siguiente:
  Marcar 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 = erastotenes_aux [x|x ← [2..n]] 0

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


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 descripto en un lenguaje de programación.

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.
- La especificación de un problema incluye un conjunto de parámetros: datos de entrada cuyos valores serán conocidos recién al ejecutar el programa.
- Además de cumplir un rol "contractual", la especificación del problema es insumo para las actividades de ...
  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"

¿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.

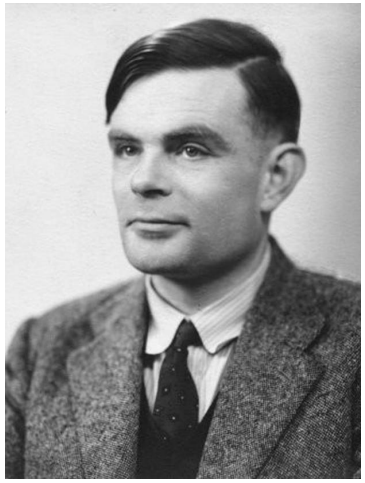
Verificación (Automática) de Programas

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**?

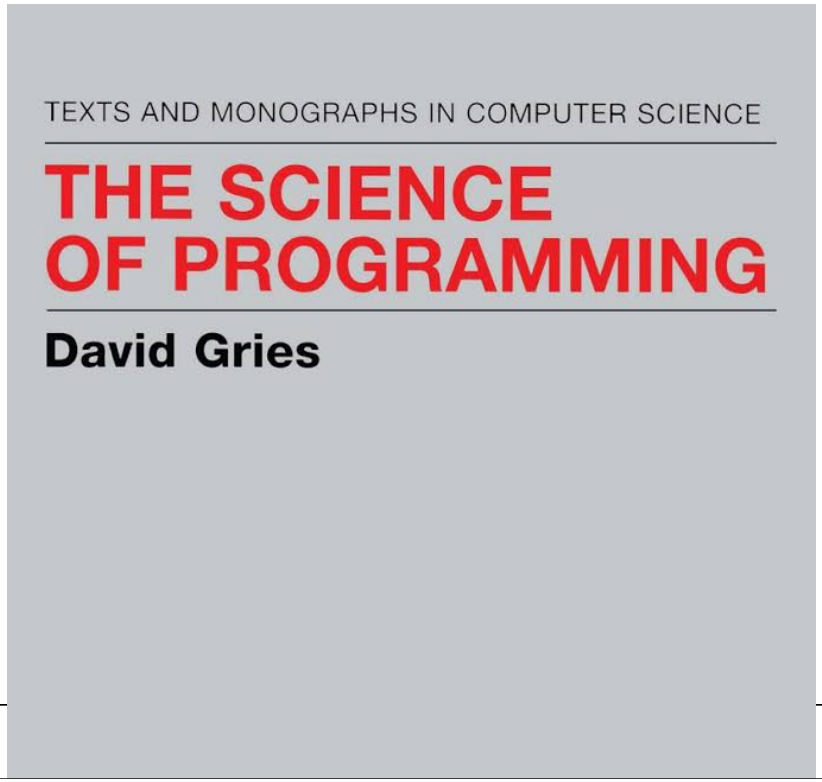
Die Entscheidungsproblem (aka The Halting Problem)



- 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

Lenguaje de especificación



Definición (Especificación) de un problema

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

- P y Q son predicados, denominados la precondición y la postcondición del procedimiento.
- *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

Ejemplos

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

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

proc *restar*(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 $\{result > x\}$
}

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?

Interpretando una especificación

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

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?

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.

  proc incremento-modificando(inout $a : \mathbb{Z}$){
    Pre $\{a = A_0\}$
    Post $\{a = A_0 + 1\}$
  }
- La variable A_0 es una metavariable, que representa el valor inicial de la variable a, y que usamos en la postcondición para relacionar el valor de salida de a con su valor inicial.

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.
- 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 {$result \neq x$}
  }

Tipos de datos

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

Tipos de datos de nuestro lenguaje de especificación

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

Tipo $\mathbb{Z}$ (números enteros)

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

Tipo $\mathbb{R}$ (números reales)

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

Tipo Bool (valor de verdad)

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

Tipo Char (caracteres)

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

Tipos enumerados

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

  enum *Nombre* { *constantes* }

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

Ejemplo de tipo enumerado

Definimos el tipo `Día` así:

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

- Valen:
  - ord(LUN) = 0
  - Día(2) = MIE
  - JUE < VIE

Tipo upla (o tupla)

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

Funciones y predicados auxiliares

- Asignan un nombre a una expresión.
- Facilitan la lectura y la escritura de especificaciones.
- Modularizan la especificación.

aux f(argumentos) : *tipo* $= e$;

- f es el nombre de la función, que puede usarse en el resto de la especificación en lugar de la expresión e.
- Los argumentos son opcionales y se reemplazan en e cada vez que se usa f.
- *tipo* es el tipo del resultado de la función (el tipo de e).

pred p(argumentos){f}

- p es el nombre del puede usarse en el resto de la especificación en lugar de la formula f.

Ejemplos de funciones auxiliares

- aux *suc*$(x : \mathbb{Z}) : \mathbb{Z} = x + 1$;
- aux *e*() : $\mathbb{R} = 2{,}7182$;
- aux *inverso*$(x : \mathbb{R}) : \mathbb{R} = 1/x$;
- pred *esPar*$(n : \mathbb{Z})$ { (n mod 2) $= 0$ }

  pred *esImpar*$(n : \mathbb{Z})$ { $\neg$ (esPar(n)) }
- pred *esFinde*$(d :$ Día$)\{d =$ SAB $\vee$ $d =$ DOM$\}$

  Otra forma:

  pred *esFinde2*$(d :$ Día$)\{d >$ VIE$\}$

Expresiones condicionales

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

- si la guarda es verdadera, elige el primero
- si no, elige el segundo

Por ejemplo

- expresión que devuelve el máximo entre dos elementos:

  aux máx$(a, b : \mathbb{Z}) : \mathbb{Z} =$ IfThenElseFi$\langle\mathbb{Z}\rangle(a > b, a, b)$;

  cuando los argumentos se deducen del contexto, se puede escribir directamente

  aux máx$(a, b : \mathbb{Z}) : \mathbb{Z} =$ IfThenElseFi$(a > b, a, b)$; o bien
  aux máx$(a, b : \mathbb{Z}) : \mathbb{Z} =$ if $a > b$ then a else b fi;
- expresión que dado x devuelve $1/x$ si $x \neq 0$ y 0 sino

  aux unoSobre$(x : \mathbb{R}) : \mathbb{R} = \underbrace{\text{if } x \neq 0 \text{ then } 1/x \text{ else } 0 \text{ fi;}}_{\text{no se indefine cuando } x = 0}$

Definir funciones auxiliares versus especificar problemas

Definimos funciones auxiliares

- Expresiones del lenguaje, que se usan dentro de las especificaciones como reemplazos sintácticos. Son de cualquier tipo.
- Dado que es un reemplazo sintáctico, ¡no se permiten definiciones recursivas!

Especificamos problemas

- Condiciones (el contrato) que debería cumplir un algoritmo para ser solución del problema.
- En una especificación dando la precondición y la postcondición con predicados de primer orden.
- No podemos usar otros problemas en la especificación. Sí podemos usar predicados y funciones auxiliares ya definidos.

Lógica proposicional - Sintaxis

- Símbolos:

  $$\text{True} \ , \ \text{False} \ , \ \neg \ , \ \wedge \ , \ \vee \ , \ \rightarrow \ , \ \leftrightarrow \ , \ (\ , \ )$$

- Variables proposicionales (infinitas)

  $$p \ , \ q \ , \ r \ , \ \ldots$$

- Fórmulas
  1. True y False son fórmulas
  2. Cualquier variable proposicional es una fórmula
  3. Si A es una fórmula, $\neg A$ es una fórmula
  4. Si $A_1, A_2, \ldots, A_n$ son fórmulas, $(A_1 \wedge A_2 \wedge \cdots \wedge A_n)$ es una fórmula
  5. Si $A_1, A_2, \ldots, A_n$ son fórmulas, $(A_1 \vee A_2 \vee \cdots \vee A_n)$ es una fórmula
  6. Si A y B son fórmulas, $(A \rightarrow B)$ es una fórmula
  7. Si A y B son fórmulas, $(A \leftrightarrow B)$ es una fórmula

Semántica clásica

- Dos valores de verdad: "verdadero" (V) y "falso" (F).
- Conociendo el valor de las variables proposicionales de una fórmula, podemos calcular el valor de verdad de la fórmula.

p	$\neg p$
V	F
F	V

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

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

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

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

Tautologías, contradicciones y contingencias

- Una fórmula es una tautología si siempre toma el valor V para valores definidos de sus variables proposicionales.
  Por ejemplo, $((p \wedge q) \rightarrow p)$ es tautología:

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

- Una fórmula es una contradicción si siempre toma el valor F para valores definidos de sus variables proposicionales.
  Por ejemplo, $(p \wedge \neg p)$ es contradicción:

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

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

Equivalencias entre fórmulas

- **Teorema.** Las siguientes son tautologías.
  1. Idempotencia
     $(p \wedge p) \leftrightarrow p$
     $(p \vee p) \leftrightarrow p$
  2. Asociatividad
     $(p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$
     $(p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$
  3. Conmutatividad
     $(p \wedge q) \leftrightarrow (q \wedge p)$
     $(p \vee q) \leftrightarrow (q \vee p)$
  4. Distributividad
     $p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$
     $p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$
  5. Reglas de De Morgan
     $\neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$
     $\neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$

Relación de fuerza

- Decimos que A es más fuerte que B cuando $(A \to B)$ es tautología.
- También decimos que A fuerza a B o que B es más débil que A.
- Por ejemplo,

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

Expresión bien definida

- Toda expresión está bien definida en un estado si todas las proposiciones valen T o F.
- Sin embargo, existe la posibilidad de que haya expresiones que no estén bien definidas.
  - Por ejemplo, la expresión x/y no está bien definida si $y = 0$.
- Por esta razón, necesitamos una lógica que nos permita decir que está bien definida la siguiente expresión
  - $y = 0 \vee x/y = 5$
- Para esto, introducimos tres valores de verdad:

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

Semántica trivaluada (secuencial)

Se llama secuencial porque ...

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

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

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

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

Semántica trivaluada (secuencial)

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

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

Especificar problemas

- **Ejemplo:** Especificar el problema de retornar el i-ésimo dígito de la representación decimal del número π.
- proc *piesimo*(in $i : \mathbb{Z}$, out *result* : $\mathbb{Z}$) {
  Pre $\{i > 0\}$
  Post $\{result = \lfloor \pi * 10^i \rfloor \bmod 10\}$
  }

Cuantificadores

El lenguaje de especificación provee formas de predicar sobre los elementos de un tipo de datos

- $(\forall x : T)\ P(x)$: Fórmula lógica. Afirma que **todos** los elementos de tipo T cumplen la propiedad P.
  - Se lee "Para todo x de tipo T se cumple $P(x)$"
- $(\exists x : T)\ P(x)$: Fórmula lógica. Afirma que **al menos un** elemento de tipo T cumple la propiedad P.
  - Se lee "Existe al menos un x de tipo T que cumple $P(x)$"

En la expresión $(\forall x : T)\ P(x)$, la variable x está ligada al cuantificador. Una variable es libre cuando no está ligada a ningún cuantificador.

Ejemplo

- **Ejemplo:** Crear un predicado `esPrimo` que sea **Verdadero** si y sólo si el número n es un número primo.
- pred esPrimo($n : \mathbb{Z}$) {
  $n > 1 \ \wedge\ (\forall n' : \mathbb{Z})(1 < n' < n \rightarrow_L n \bmod n' \neq 0)$
  }
- **Observación:** `x mod y` se indefine si $y = 0$.
- **Ejemplo:** Especificar el problema de, dado un número mayor a 1, indicar si el número es un número primo.
- proc *primo*(in $n : \mathbb{Z}$, out *result* : Bool) {
  Pre $\{n > 1\}$
  Post $\{result = true \leftrightarrow \text{esPrimo}(n)\}$
  }

Operando con cuantificadores

- **Ejemplo:** Todos los enteros entre 1 y 10 son pares:
  $(\forall n : \mathbb{Z})(1 \leq n \leq 10 \rightarrow n \bmod 2 = 0)$.
- **Ejemplo:** Existe un entero entre 1 y 10 que es par:
  $(\exists n : \mathbb{Z})(1 \leq n \leq 10 \wedge n \bmod 2 = 0)$.
- En general, si queremos decir que todos los enteros x que cumplen $P(x)$ también cumplen $Q(x)$, decimos:
  $(\forall x : \mathbb{Z})(P(x) \rightarrow Q(x))$.
- Para decir que existe un entero que cumple $P(x)$ y que también cumple $Q(x)$, decimos:
  $(\exists x : \mathbb{Z})(P(x) \wedge Q(x))$.

Operando con cuantificadores

- La negación de un cuantificador universal es un cuantificador existencial, y viceversa:
  $\neg(\forall n : \mathbb{Z})P(n) \leftrightarrow (\exists n : \mathbb{Z})\neg P(n)$.
  $\neg(\exists n : \mathbb{Z})P(n) \leftrightarrow (\forall n : \mathbb{Z})\neg P(n)$.
- Un cuantificador universal generaliza la conjunción:
  $(\forall n : \mathbb{Z})\big(a \leq n \leq b \rightarrow P(n)\big) \ \wedge \ P(b+1)$
  $\leftrightarrow \ (\forall n : \mathbb{Z})\big(a \leq n \leq b+1 \rightarrow P(n)\big)$.
- Un cuantificador existencial generaliza la disyunción:
  $(\exists n : \mathbb{Z})\big(a \leq n \leq b \wedge P(n)\big) \ \vee \ P(b+1)$
  $\leftrightarrow \ (\exists n : \mathbb{Z})\big(a \leq n \leq b+1 \wedge P(n)\big)$.

Especificar problemas

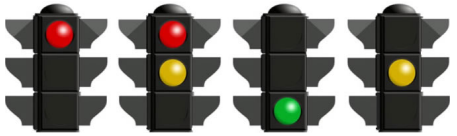
- **Ejemplo:** Especificar un procedimiento que calcule el máximo común divisor (mcd) entre dos números positivos.
- proc *mcd*(in $n : \mathbb{Z}$, in $m : \mathbb{Z}$, out *result* $: \mathbb{Z}$) {
  Pre $\{n \geq 1 \wedge m \geq 1\}$
  Post $\{n \bmod result = 0 \wedge m \bmod result = 0 \wedge$
  $\neg(\exists p : \mathbb{Z})(p > result \wedge n \bmod p = 0 \wedge m \bmod p = 0)\}$
  }
- Observar que no damos una fórmula que especifica el valor de retorno, sino que solamente damos las propiedades que debe cumplir!

Especificando un semáforo

- **Ejemplo:** Representamos con tres valores de tipo *Bool* el estado de la luz verde, amarilla y roja de un semáforo.
- Escribir el procedimiento que inicializa el semáforo con la luz roja y el resto de las luces apagadas.
- proc *iniciar*(out v, a, r : Bool) {
  Pre $\{true\}$
  Post $\{v = \textbf{false} \wedge a = \textbf{false} \wedge r = \textbf{true}\}$
  }

Especificando un semáforo

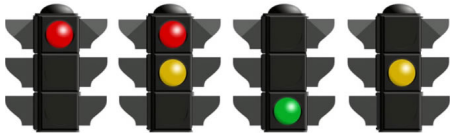
Estado de las luces



- Podemos especificar un predicado para representar cada estado válido del semáforo:
- pred esRojo(v, a, r: Bool) { $v = \textbf{false} \wedge a = \textbf{false} \wedge r = \textbf{true}$ }
- pred esRojoAmarillo(v, a, r: Bool) {
  $v = \textbf{false} \wedge a = \textbf{true} \wedge r = \textbf{true}$ }
- pred esVerde(v, a, r: Bool){ $v = \textbf{true} \wedge a = \textbf{false} \wedge r = \textbf{false}$ }
- pred esAmarillo(v, a, r: Bool){ $v = \textbf{false} \wedge a = \textbf{true} \wedge r = \textbf{false}$ }

Especificando un semáforo

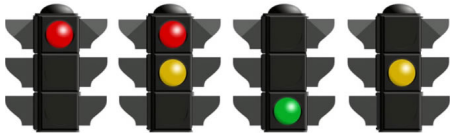
Estado válido



- Podemos especificar un predicado para representar que el semáforo está en un estado válido:
- pred esValido(v, a, r: Bool) {
  $esRojo(v, a, r)$
  $\vee\ esRojoAmarillo(v, a, r)$
  $\vee\ esVerde(v, a, r)$
  $\vee\ esAmarillo(v, a, r)$
  }

Especificando un semáforo

Avance del estado de las luces



- proc *avanzar*(inout v, a, r : Bool) {
  Pre {
  $esValido(v, a, r)$
  $\wedge v = V_0 \wedge r = R_0 \wedge a = A_0$
  }
  Post {
  $(esRojo(V_0, A_0, R_0) \rightarrow esRojoAmarillo(v, a, r))$
  $\wedge\ (esRojoAmarillo(V_0, A_0, R_0) \rightarrow esVerde(v, a, r)$
  $\wedge\ (esVerde(V_0, A_0, R_0) \rightarrow esAmarillo(v, a, r))$
  $\wedge\ (esAmarillo(V_0, A_0, R_0) \rightarrow esRojo(v, a, r))\}$
  }

Bibliografía

- David Gries - The Science of Programming
  - Chapter 1 - Propositions (Fórmulas, Tautologías, etc.)
  - Chapter 2 - Reasoning using Equivalence Transformations (Propiedades, De Morgan, etc.)