

Tema 2: Especificación de algoritmos

Clara María Segura Díaz

Fundamentos de Algoritmia, Curso 2020-21

Dpto. de Sistemas Informáticos y Computación

Facultad de Informática

Universidad Complutense de Madrid

En este tema vamos a hablar de la especificación de algoritmos. Una especificación de un algoritmo consiste en decir QUÉ ES LO QUE HACE un algoritmo.

- **Diseño de programas. Formalismo y abstracción;** R. Peña. Tercera edición. Prentice Hall, 2005.

Capítulo 2

- **Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios.** N. Martí, C. Segura, J.A. Verdejo. Ibergarceta Publicaciones, 2012.

Capítulo 1

- **The Derivation of algorithms;** A. Kaldewaij; Prentice Hall, 1990.

Capítulos 0, 1 y 3

Especificación algoritmos

Especificación es QUÉ DEBE HACER nuestro algoritmo e implementación es CÓMO HACERLO.

- Especificación vs implementación

Qué es la especificación y cual es la principal diferencia con la implementación

- Especificaciones pre/post

Qué es una precondition, y qué es la postcondición

- Semántica

Cómo expresar, en lenguaje matemático la especificación

- Especificación de funciones

Ejemplos de cómo especificar funciones.

Muy importante verlos todos.

Y entenderlos todos también.

Especificación vs implementación

En esta parte del tema veremos que la diferencia entre especificar e implementar consiste en que especificar es decir QUÉ HACE un algoritmo, e implementar consiste en decir CÓMO SE HACE el algoritmo, en CÓMO SE RESUELVE un problema.

- Un **algoritmo** es la descripción precisa de una sucesión de instrucciones que permiten llevar a cabo un trabajo en un número finito de pasos.
- Un **buen algoritmo** debe ser:
 - correcto,
 - eficiente,
 - reutilizable,
 - fácil de mantener.

Que podamos modificarlo sin problema alguno.
- A la hora de desarrollar un algoritmo, los pasos a seguir son los siguientes: **primero especificar** y después **implementar**. Especificamos primero para tener buenas implementaciones.
- Un **programa** es la codificación en un lenguaje de programación concreto de un algoritmo.

Especificar es decir lo que tiene que hacer y luego la implementación debemos de hacer el algoritmo a ordenador, que eso implica el cómo hacerlo.

- **Especificar**

- Consiste en detallar cuidadosamente el problema que se quiere resolver.
- Contesta a la pregunta **¿Qué hace el algoritmo?** sin dar ningún detalle de cómo se hace.
- Es deseable que las **especificaciones sean claras y precisas** a la vez.

Básicamente especificar consiste en detallar lo que va a hacer un algoritmo o lo que quieras que haga el algoritmo. Es importante que la especificación del algoritmo sea clara y precisa y que abarque todos los casos posibles. La especificación se hace de manera matemática, detallando cual es la PRECONDICIÓN, cual es la FUNCIÓN, sus ARGUMENTOS, lo que DEVUELVE y la POSTCONDICIÓN.

Especificación vs implementación

- **Especificar**

- Consiste en detallar cuidadosamente el problema que se quiere resolver.
- Contesta a la pregunta **¿Qué hace el algoritmo?**, sin dar ningún detalle de **cómo se hace**.
- Es deseable que las **especificaciones sean claras y precisas** a la vez.

- **Implementar**, por el contrario, consiste en decir **cómo** se resuelve el problema.

Nosotros nos vamos a centrar en este tema en la especificación de los algoritmos.

Especificación vs implementación

- **Especificar**

- Consiste en **detallar cuidadosamente el problema** que se quiere resolver.
- Contesta a la pregunta **¿Qué hace el algoritmo?**, sin dar ningún detalle de cómo se hace.
- Es deseable que las especificaciones sean **claras y precisas** a la vez.

- **Implementar**, por el contrario, consiste en decir **cómo** se resuelve el problema.

- La especificación debe responder a cualquier pregunta sobre el uso del algoritmo sin tener que acudir a la implementación.

Básicamente implementar es decir CÓMO hacerlo.

Tenemos que saber implementar los algoritmos pero también especificarlos.

Especificación como contrato

- La especificación de un algoritmo tiene un doble destinatario:
 - ① Los usuarios del algoritmo: ha de detallar las obligaciones del usuario al invocarlo y el resultado producido si es invocado correctamente.
Por ejemplo, que la entrada sea correcta. El que hace la especificación se tiene que encargar de decir cuál puede ser la entrada

Especificación como contrato

- La especificación de un algoritmo tiene un doble destinatario:
 - ① Los **usuarios** del algoritmo: ha de detallar las obligaciones del usuario al invocarlo y el resultado producido si es invocado correctamente.
 - ② El **implementador** del algoritmo: define los requisitos que cualquier implementación válida debe satisfacer, es decir las obligaciones del implementador.

Especificación como contrato

- La especificación de un algoritmo tiene un doble destinatario:
 - ① Los **usuarios** del algoritmo: ha de detallar las obligaciones del usuario al invocarlo y el resultado producido si es invocado correctamente.
 - ② El **implementador** del algoritmo: define los requisitos que cualquier implementación válida debe satisfacer, es decir las obligaciones del implementador.

	Precondición	Postcondición
Usuario	Debe garantizar que se cumpla inicialmente.	Tiene derecho a esperar que A termine en un estado que la cumpla.
Implementador	Tiene derecho a esperar que se cumpla.	Debe garantizar que A termine en un estado que la cumpla.

El usuario que desarrolla la especificación debe de encargarse de se cumpla para los casos iniciales, ya que el implementador tiene derecho a esperar que se cumpla, mientras que el usuario que realiza la especificación tiene derecho a esperar que se cumpla para el caso final y que el implementador debe garantizar que se cumple.

Especificación como contrato

- La especificación de un algoritmo tiene un doble destinatario:

- ① Los usuarios del algoritmo: ha de detallar las obligaciones del usuario al invocarlo y el resultado producido si es invocado correctamente. pre-post condición
- ② El implementador del algoritmo: define los requisitos que cualquier implementación válida debe satisfacer, es decir las obligaciones del implementador.

	Precondición	Postcondición
Usuario	Debe garantizar que se cumpla inicialmente.	Tiene derecho a esperar que A termine en un estado que la cumpla.
Implementador	Tiene derecho a esperar que se cumpla.	Debe garantizar que A termine en un estado que la cumpla.

- Una vez establecida la especificación, los trabajos del usuario y del implementador pueden proseguir por separado.
- La especificación actúa pues como una barrera o contrato

• Permite independizar los razonamientos de corrección de cada componente. Nosotros en los ejercicios haremos tanto de implementadores como de especificadores, pero lo más importante es que para el examen hagamos buenas especificaciones, para hacer una buena implementación.

Propiedades de una buena especificación

Precisión Ha de responder a cualquier pregunta sobre el uso del algoritmo sin tener que acudir a la implementación.

- El lenguaje natural no permite la precisión requerida si no es sacrificando la brevedad. **SE UTILIZA EL LENGUAJE MATEMÁTICO**

Brevedad Ha de ser significativamente más breve que el código que especifica.

- Los *lenguajes formales* ofrecen a la vez precisión y brevedad.

Claridad Ha de transmitir la *intuición*.

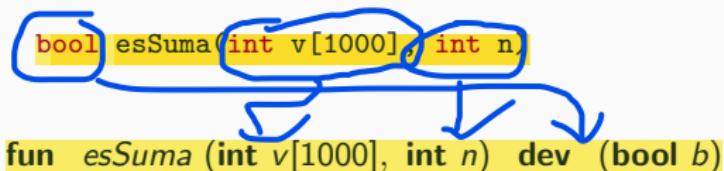
- A veces el lenguaje formal ha de ser complementado con explicaciones informales.

- Eliminan ambigüedades que el usuario y el implementador podrían resolver de formas distintas, dando lugar a errores de uso o de implementación que aparecerían en ejecución.
- Permiten realizar una **verificación formal** del algoritmo que consiste en razonar sobre la corrección del algoritmo mediante el **uso de la lógica**.
- Permite generar **automáticamente casos de prueba**.

→ Como los del juez.

Ejemplo

Supongamos que queremos una función `esSuma` que, dado un vector `int v[1000]` y un entero `n`, devuelva un valor booleano que indique si el valor de alguno de los elementos `v[0], v[1], ..., v[n-1]` es igual a la suma de todos los elementos que le preceden en el vector:



No quedan claras todas las obligaciones del usuario.

Nosotros nos encargamos de la especificación.

• ¿Serían admisibles llamadas a `esSuma` con valores negativos de `n`?

Creo que se refiere a que, en el caso de que `n` fuera las posiciones del vector, se permite un `n` negativo. Esto si no me equivoco debe aclararse en la precondición.

Es decir, puede haber un número negativo de elementos o 0 elementos por ejemplo¿?

Ejemplo

Aquí la profesora especifica mal la teoría. No sabemos qué es n . Podemos suponer que n es el número de elementos del vector.

Supongamos que queremos una función `esSuma` que, dado un vector `int v[1000]` y un entero n , devuelva un valor booleano que indique si el valor de alguno de los elementos $v[0]$, $v[1]$, ..., $v[n-1]$ es igual a la suma de todos los elementos que le preceden en el vector:

```
bool esSuma(int v[1000], int n)
```

```
fun esSuma (int v[1000], int n) dev (bool b)
```

No quedan claras todas las obligaciones del usuario.

- ¿Serían admisibles llamadas a `esSuma` con valores negativos de n ?
- ¿Y con $n = 0$?

Ejemplo

Supongamos que queremos una función `esSuma` que, dado un vector `int v[1000]` y un entero `n`, devuelva un valor booleano que indique si el valor de alguno de los elementos `v[0]`, `v[1]`, ..., `v[n-1]` es igual a la suma de todos los elementos que le preceden en el vector:

```
bool esSuma(int v[1000], int n)
```

```
fun esSuma (int v[1000], int n) dev (bool b)
```

No quedan claras todas las obligaciones del usuario.

- ¿Serían admisibles llamadas a `esSuma` con valores negativos de `n`?
- ¿Y con `n = 0`?
- ¿Y con `n ≥ 1000`?

Ejemplo

Supongamos que queremos una función `esSuma` que, dado un vector `int v[1000]` y un entero `n`, devuelva un valor booleano que indique si el valor de alguno de los elementos `v[0]`, `v[1]`, ..., `v[n-1]` es igual a la suma de todos los elementos que le preceden en el vector:

```
bool esSuma(int v[1000], int n)
```

```
fun esSuma (int v[1000], int n) dev (bool b)
```

No quedan claras todas las obligaciones del usuario.

- ¿Serían admisibles llamadas a `esSuma` con valores negativos de `n`?
- ¿Y con `n = 0`? No podemos ya que el número de elementos siempre es 1000 por tanto n no puede tener ninguno de estos valores.
- ¿Y con `n ≥ 1000`?

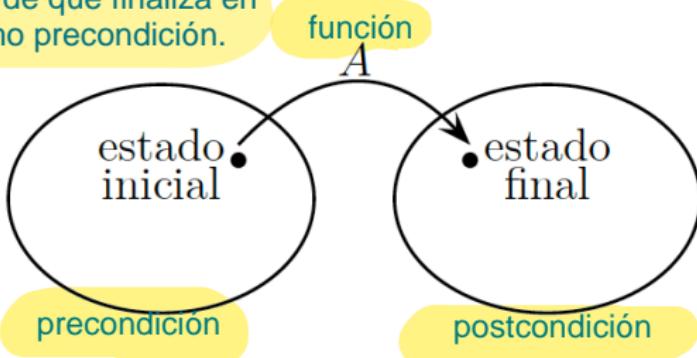
Tampoco están claras las obligaciones del implementador. Por ejemplo, si $n \geq 1$ y $v[0] = 0$, la función ¿ha de devolver `true` o `false`? Esto nos lo hemos planteado nosotros.

Como con la expresión de arriba no queda claro todo debemos de realizar una precondición y postcondición

Especificación pre/post

- Para escribir especificaciones claras y precisas necesitamos un *lenguaje formal*, con sintaxis y semántica perfectamente definidos.
- La **especificación pre/post** (C.A.R. Hoare, 1969) se basa en contemplar un algoritmo A como una caja negra. La precondition implica la postcondicion ¿?
- A actúa como una función de estados en estados: comienza su ejecución en un *estado inicial* válido, descrito por el valor de los parámetros de entrada, y termina en un *estado final* en el que los parámetros de salida contienen los resultados esperados.

Si empieza en un estado valido P $\{ P \} \ A \ \{ Q \}$
debemos asegurarnos de que finaliza en
un estado definido como precondition.



Ejemplo

El estado inicial normalmente es el numero de valores de entrada. La postcondición en este caso debe cumplir que el valor de alguna de las posiciones del vector es igual a la suma de los valores de todas las posiciones que le preceden.

$\{0 \leq n \leq 1000\}$ Precondición

~~fun esSuma(int v[1000], int n) dev (bool b)~~ definición de la función
 $\{(b = \exists i : 0 \leq i < n : V[i] = (\sum j : 0 \leq j < i : V[j]))\}$ postcondición

b es true

La postcondición por tanto dice que EXISTE un i para el cual hay una posición tal que el sumatorio de todos los anteriores sea $= v[i]$.

Responde a las preguntas planteadas más arriba:

- Las llamadas con n negativo o con $n > 1000$ son incorrectas, pero con $n = 1000$ son correctas.
- Las llamadas con $n = 0$ son correctas, y la función devolverá $b = \text{false}$.
- Las llamadas con $n \geq 1$ y $V[0] = 0$ han de devolver $b = \text{true}$.

Creo que es porque es el caso vacío.

Lo de arriba dice que b es true si existe un i tal que el $v[i] = \text{suma de los } V[j]$ con $j < i >= 0$

Especificaciones pre/post

Vamos a ver cómo especificar de manera correcta la precondition y la postcondición.

- Si S representa la función a especificar, Q es un predicado que tiene como variables libres los parámetros de entrada de S , y R es un predicado que tiene como variables libres los parámetros de entrada y de salida de S , la notación

$$\{Q\} S \{R\}$$

es la especificación formal de S y ha de leerse:

"Si S comienza su ejecución en un estado descrito por Q , S termina y lo hace en un estado descrito por R ".

La forma de expresar los enunciados de precondition - función - postcondición se expresan como $\{Q\} S \{R\}$. Lo cual se lee: si una función S comienza su estado de ejecución en un estado descrito por Q , S termina y lo hace en un estado descrito por R .

Especificación pre/post

Vamos a llamarlo mejor P

Y a la postcondición vamos a llamarla Q mejor.

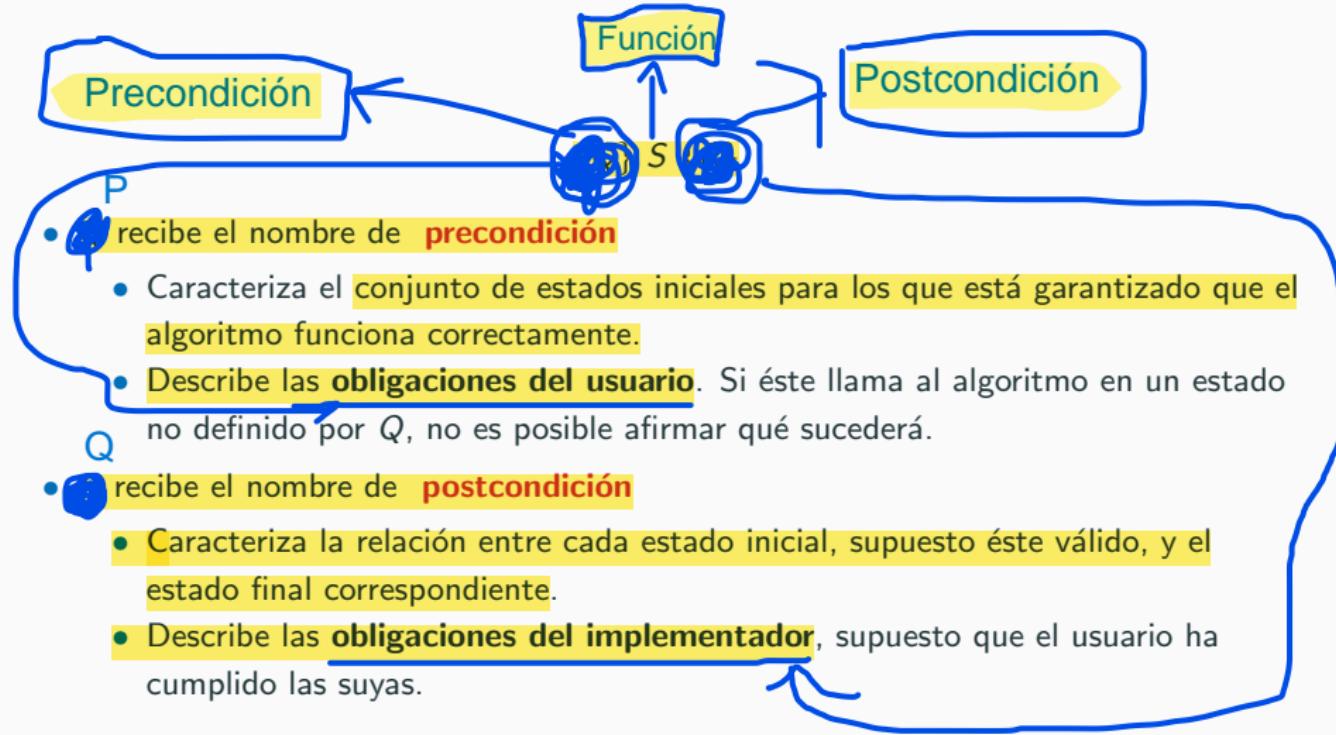


- recibe el nombre de **precondición**

- Caracteriza el conjunto de estados iniciales para los que está garantizado que el algoritmo funciona correctamente. Para qué entrada funciona bien el algoritmo?
- Describe las obligaciones del usuario. Si éste llama al algoritmo en un estado no definido por Q, no es posible afirmar qué sucederá.

El usuario debe asegurarse de que la precondición sea correcta para que el implementador luego tenga la obligación de que la postcondición también lo sea después. Más bien debe asegurarse de llamar al algoritmo con estados permitidos por Precondición.

Especificación pre/post



Si una función S comienza su ejecución en un estado descrito por una precondición Q, entonces terminará su ejecución en un estado descrito por una postcondición R

Lógica de predicados

El usuario debe encargarse de que la entrada de los datos pertenezca al estado descrito en la precondición y el implementador debe asegurarse de que, si se respeta esa precondición, se llegue a la postcondición.

Predicados

Una fórmula en lógica de predicados o, más brevemente, un *predicado*, es:

① una fórmula atómica:

- las constantes booleanas, true y false;
- una variable booleana, b ;
- las expresiones aritméticas relacionalles tales como $2 * x = y$, $u \bmod v \neq 0$, $a > 0$, $i \leq j$;
- una función matemática conocida o especificada formalmente, que devuelva un resultado booleano, como $\text{primo}(x)$, $\text{divide}(x,y)$, o $\text{mayúscula}(c)$.

$x \bmod 2 \neq 0$ implica que no sea par---> que sea impar.

Lógica de predicados

Predicados

Una fórmula en lógica de predicados o, más brevemente, un *predicado*, es:

① una **fórmula atómica**:

- las constantes booleanas, true y false;
- una variable booleana, b ;
- las expresiones aritméticas relacionales tales como $2 * x = y$, $u \bmod v \neq 0$, $a > 0$, $i \leq j$;
- una función matemática conocida o especificada formalmente, que devuelva un resultado booleano, como $\text{primo}(x)$, $\text{divide}(x,y)$, o $\text{mayúscula}(c)$.

② una negación, $\neg P$, siendo P un predicado.

③ una conjunción, $P \wedge Q$, siendo P y Q predicados.

y

④ una disyunción, $P \vee Q$, siendo P y Q predicados.

ó

⑤ un condicional $P \rightarrow Q$, siendo P y Q predicados.

⑥ un bicondicional $P \leftrightarrow Q$, siendo P y Q predicados.

Lógica de predicados

- 7 Una cuantificación universal, $\forall x : P(x)$, donde $P(x)$ representa un predicado que "depende" de x , es decir, que contenga libre a x . Generalmente utilizaremos cuantificaciones relativizadas:

- con indicación del tipo de la variable cuantificada,

$$\forall x : x \in T : P(x)$$

- con indicación del rango de la variable cuantificada,

$$\forall x : R(x) : P(x)$$

donde $R(x)$ es un predicado que define el rango de la variable x .

Abrevia

$$\forall x : R(x) \rightarrow P(x)$$

Lógica de predicados

- 7 una *cuantificación universal*, $\forall x : P(x)$, donde $P(x)$ representa un predicado que "depende" de x , es decir, que contenga libre a x . Generalmente utilizaremos cuantificaciones relativizadas:

- con indicación del tipo de la variable cuantificada,

$$\forall x : x \in T : P(x)$$

- con indicación del rango de la variable cuantificada,

$$\forall x : R(x) : P(x)$$

donde $R(x)$ es un predicado que define el *rango* de la variable x .

Abrevia

$$\forall x : R(x) \rightarrow P(x)$$

También podemos usar cuantificación sobre varias variables, como

$$\forall i, j : 0 \leq i \leq j < n : v[i] \leq v[j].$$

Para todo i, j con i y j mayores o iguales que 0 y estrictamente menores que n : $v[i]$ debe de ser menor o igual que $v[j]$

Lógica de predicados

- 7 una cuantificación universal, $\forall x : P(x)$, donde $P(x)$ representa un predicado que "depende" de x , es decir, que contenga libre a x . Generalmente utilizaremos cuantificaciones relativizadas:

- con indicación del tipo de la variable cuantificada,

$$\forall x : x \in T : P(x)$$

tipo de variable

- con indicación del rango de la variable cuantificada,

$$\forall x : R(x) : P(x)$$

Rango de nuestra variable.

donde $R(x)$ es un predicado que define el rango de la variable x .

Abrevia

$$\forall x : R(x) \rightarrow P(x)$$

También podemos usar cuantificación sobre varias variables, como

$$\forall i, j : 0 \leq i \leq j < n : v[i] \leq v[j].$$

Para todo i, j tal que $0 \leq i \leq j < n$ se cumple que $v[i] \leq v[j]$

- 8 una cuantificación existencial, $\exists x : P(x)$, $\exists x : x \in T : P(x)$ o $\exists x : R(x) : P(x)$.

Abrevia

$$\exists x : R(x) \wedge P(x)$$

existe un x tal que...

Ejemplos

Supongamos que v es un vector de tamaño N .

- x aparece como componente de v . E i: $0 \leq i < n : v[i] = x$;

(Existe un i: $0 \leq i \leq v.size() : V[i] = x$)

Ejemplos

Supongamos que v es un vector de tamaño N .

- x aparece como componente de v .

Podemos utilizar el cuantificador existencial, para decir que existe una posición del vector donde aparece el valor x :

$$(\exists i : 0 \leq i < N : v[i] = x),$$

donde $0 \leq i < N$ es una abreviatura de $0 \leq i \wedge i < N$.

Ejemplos

Supongamos que v es un vector de tamaño N .

$v[N]$ inicializamos un vector a tamaño N

- x aparece como componente de v .

Podemos utilizar el cuantificador existencial, para decir que existe una posición del vector donde aparece el valor x :

Variables ligadas

$$(\exists i : 0 \leq i < N : v[i] = x),$$

donde $0 \leq i < N$ es una abreviatura de $0 \leq i \wedge i < N$.

Nótese que la variable i está ligada, por lo que se puede renombrar por cualquier

otra (que no aparezca como libre),

Como está ligada la i , la podemos cambiar por otra que no esté libre.

$$(\exists j : 0 \leq j < N : v[j] = x).$$

Existen 2 tipos de variables:

1º. Las variables ligadas: son aquellas que tienen algo delante que les afecta.

Puede ser un cuantificador o lo que sea

2º. Variables libres/de programa: Son aquellas que no tienen nada que les afecte.

Ejemplos

Supongamos que v es un vector de tamaño N .

- x aparece como componente de v .

Podemos utilizar el cuantificador existencial, para decir que existe una posición del vector donde aparece el valor x :

$$(\exists i : 0 \leq i < N : v[i] = x),$$

donde $0 \leq i < N$ es una abreviatura de $0 \leq i \wedge i < N$.

Nótese que la variable i está ligada, por lo que se puede renombrar por cualquier otra (que no aparezca como libre),

$$(\exists j : 0 \leq j < N : v[j] = x).$$

- v se anula en algún punto. Hay algún elemento de V que es 0
E i: $0 \leq i < v.size() : v[i] = 0$.
(Existe un i: $0 \leq i < v.size() : v[i] = 0$)

Ejemplos

Supongamos que v es un vector de tamaño N .

- x aparece como componente de v .

Podemos utilizar el cuantificador existencial, para decir que existe una posición del vector donde aparece el valor x :

$$(\exists i : 0 \leq i < N : v[i] = x),$$

donde $0 \leq i < N$ es una abreviatura de $0 \leq i \wedge i < N$.

Nótese que la variable i está ligada, por lo que se puede renombrar por cualquier otra (que no aparezca como libre),

$$(\exists j : 0 \leq j < N : v[j] = x).$$

- v se anula en **algún** punto.

$$(\exists i : 0 \leq i < N : v[i] = 0).$$

La i es una variable LIGADA, que la podemos cambiar por cualquier otra que NO ESTÉ LIBRE (LA N ESTÁ LIBRE NO LA PODEMOS USAR).

Ejemplos

- v tiene valores estRICTAMENTE positivos en tODAS sus componentes.

P.T I: $0 \leq i < V.SIZE(): v[i] > 0$

Universal

{p.t i: $0 \leq i < N: v[i] > 0$ }

Como es estRICTAMENTE positivo el 0 NO lo contabilizamos como un positivo

Ejemplos

- v tiene valores estrictamente positivos en todas sus componentes.

$$(\forall i : 0 \leq i < N : v[i] > 0).$$

Ejemplos

- v tiene valores estrictamente positivos en todas sus componentes.

$$(\forall i : 0 \leq i < N : v[i] > 0).$$

- x aparece una sola vez como componente de v .

Operador de conteo.

Aquí tendríamos que decir que E una posición del vector que vale x pero sin embargo, para todo el resto de posiciones, todas valen distinto de x .

(#i: 0<=i<v.size(): V[i]=x)=1

El # es un símbolo para decir que es una variable de conteo y aquí nos sirve para decir que el número de posiciones en las que ocurre es 1

Ejemplos

- v tiene valores estrictamente positivos en todas sus componentes.

$$(\forall i : 0 \leq i < N : v[i] > 0).$$

- x aparece una sola vez como componente de v .

Utilizando el cuantificador existencial y el universal podemos escribir

$$(\exists i : 0 \leq i < N : (v[i] = x) \wedge (\forall j : 0 \leq j < N \wedge j \neq i : v[j] \neq x)).$$

Existe una posición del vector para la cual su valor vale x y además, para el resto de posiciones, tenemos que aclarar que $i \neq j$ y que para todos los valores de j ninguno vale x .

Pero para no tener que poner tantas cosas es mejor utilizar el operador de CONTEO.
Para determinar cual es el numero de veces que se debe cumplir un enunciado.

$$(\#i: 0 \leq i < N : V[i] = x) = 1$$

Que solo ocurra una vez.

Ejemplos

- v tiene valores estrictamente positivos en todas sus componentes.

$$(\forall i : 0 \leq i < N : v[i] > 0).$$

- x aparece una sola vez como componente de v .

Utilizando el cuantificador existencial y el universal podemos escribir

$$(\exists i : 0 \leq i < N : (v[i] = x) \wedge (\forall j : 0 \leq j < N \wedge j \neq i : v[j] \neq x)).$$

Pero más claro puede ser utilizar el **operador de conteo**. La forma general de este operador es la siguiente:

$$\#i : R(i) : P(i)$$

y devuelve el número de veces que se verifican a la vez $R(i)$ y $P(i)$, es decir, de los valores i que cumplen $R(i)$, cuántos verifican también $P(i)$.

Se puede poner como dice arriba pero ese mejor utilizar el OPERADOR DE CONTEO. Devuelve el número de veces que se cumple el enunciado. Por ello el enunciado correcto para decir que se cumple únicamente una vez es:

$$(\#i : 0 \leq i < N : v[i] = x) = 1 \text{ Con esto decimos que el enunciado se cumple 1 vez.}$$

Ejemplos

- v tiene valores estrictamente positivos en todas sus componentes.

$$(\forall i : 0 \leq i < N : v[i] > 0).$$

- x aparece una sola vez como componente de v .

Utilizando el cuantificador existencial y el universal podemos escribir

$$(\exists i : 0 \leq i < N : (v[i] = x) \wedge (\forall j : 0 \leq j < N \wedge j \neq i : v[j] \neq x)).$$

Pero más claro puede ser utilizar el **operador de conteo**. La forma general de este operador es la siguiente:

$$\#i : R(i) : P(i)$$

y devuelve el número de veces que se verifican a la vez $R(i)$ y $P(i)$, es decir, de los valores i que cumplen $R(i)$, cuántos verifican también $P(i)$.

Si contamos cuántas veces aparece x en el vector v , e igualamos este valor a 1, obtendremos el predicado que buscábamos:

$$(\#i : 0 \leq i < N : v[i] = x) = 1.$$

Ejemplos

CUANDO SE HABLE DEL NÚMERO DE VECES QUE DEBE OCURRIR ALGO UTILIZAREMOS EL OPERADOR DE CONTEO.

- x es el número de veces que aparece en v el primer elemento.

$$x = (\#i: 0 \leq i < N: v[i] = v[0])$$

$$x = (\#i: 0 \leq i < v.size(): v[i] = v[0])$$

Ejemplos

- x es el número de veces que aparece en v el primer elemento.

$$x = (\#i : 0 \leq i < N : v[i] = v[0]).$$

Ejemplos

- x es el número de veces que aparece en v el primer elemento.

$$x = (\#i : 0 \leq i < N : v[i] = v[0]).$$

- x es la suma de las componentes de v . $x = (\text{sum } i : 0 \leq i < N : v[i])$

$$x = (\text{sum } i : 0 \leq i < N : v[i]).$$

$$x = (\sum i : 0 \leq i < N : v[i])$$

Ejemplos

- x es el número de veces que aparece en v el primer elemento.

$$x = (\#i : 0 \leq i < N : v[i] = v[0]).$$

- x es la suma de las componentes de v .

$$x = (\sum i : 0 \leq i < N : v[i]).$$

Ejemplos

- x es el número de veces que aparece en v el primer elemento.

$$x = (\#i : 0 \leq i < N : v[i] = v[0]).$$

- x es la suma de las componentes de v .

$$x = (\sum i : 0 \leq i < N : v[i]).$$

En general, expresaremos un sumatorio de la siguiente manera:

$$\sum i : R(i) : e(i).$$

Ejemplos

- x es el máximo de las componentes de v .

$N = v.size()$

$x = (\max i : 0 \leq i < N : V[i])$

yo haría: $x = (\max i : 0 \leq i < v.size() : v[i])$

Ejemplos

- x es el máximo de las componentes de v .

Aquí vamos a utilizar otro cuantificador, que tiene la siguiente forma general:

$$\max i : R(i) : e(i).$$

Ejemplos

- x es el máximo de las componentes de v .

Aquí vamos a utilizar otro cuantificador, que tiene la siguiente forma general:

$$\max i : R(i) : e(i).$$

Para el ejemplo:

$$x = (\max i : 0 \leq i < N : v[i]).$$

Ejemplos

- x es el máximo de las componentes de v .

Aquí vamos a utilizar otro cuantificador, que tiene la siguiente forma general:

$$\max i : R(i) : e(i).$$

Para el ejemplo:

$$x = (\max i : 0 \leq i < N : v[i]).$$

Consideraremos que el cuantificador \max está bien definido cuando el rango de definición, en este caso $0 \leq i < N$, es no vacío. Por tanto, en este ejemplo N ha de ser > 0 para que el predicado esté bien definido.

Ejemplos

- x es el máximo de las componentes de v .

Aquí vamos a utilizar otro cuantificador, que tiene la siguiente forma general:

$$\max i : R(i) : e(i).$$

Para el ejemplo:

$$x = (\max i : 0 \leq i < N : v[i]).$$

Consideraremos que el cuantificador max está bien definido cuando el rango de definición, en este caso $0 \leq i < N$, es no vacío. Por tanto, en este ejemplo N ha de ser > 0 para que el predicado esté bien definido.

- m es el menor índice de v que contiene el valor x .

$$m = (\min i : 0 \leq i < N \wedge v[i] = x : i)$$

$$m = (\min i : 0 \leq i < N \wedge v[i] = x : i) \text{ ya que queremos el índice, no el } v[i]$$

Ejemplos

- x es el máximo de las componentes de v .

Aquí vamos a utilizar otro cuantificador, que tiene la siguiente forma general:

$$\max i : R(i) : e(i).$$

Para el ejemplo:

$$x = (\max i : 0 \leq i < N : v[i]).$$

Consideraremos que el cuantificador max está bien definido cuando el rango de definición, en este caso $0 \leq i < N$, es no vacío. Por tanto, en este ejemplo N ha de ser > 0 para que el predicado esté bien definido.

- m es el menor índice de v que contiene el valor x .

$$m = (\min i : 0 \leq i < N \wedge v[i] = x : i).$$

Ejemplos

- El valor de cada componente de v es el doble de su índice.

(p.t i: $0 \leq i < N$: $v[i] = 2^i$)

(p.t i: $0 \leq i < N$: $v[i] = 2^i$)

Ejemplos

- El valor de cada componente de v es el **doble de su índice**.

$$\forall i : 0 \leq i < N : v[i] = 2 * i.$$

Ejemplos

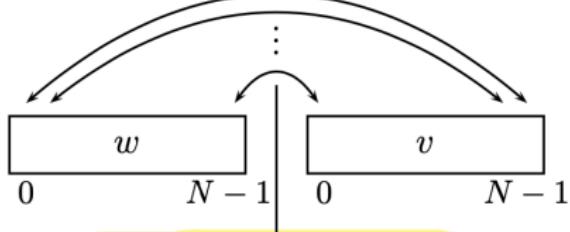
- El valor de cada componente de v es el doble de su índice.

$$(\forall i : 0 \leq i < N : v[i] = 2 * i).$$

- w es la imagen especular de v.

Es como si el vector esté "ordenado" (no tiene por qué estarlo) al revés.

Es para cada elemento



El elemento del final de w es el elemento del principio del otro.

$$(\forall i : 0 \leq i < N : w[i] = v[N - i - 1]).$$

Este caso es muy importante aprenderlo bien

$w[1] = v[48]$
 $w[2] = v[47]$
 $w[3] = v[46]$
 $w[40] = v[9]$
 $w[9] = v[40]$

Para $N=50$ el valor de la posición $w[0] = v[49]$ eso es la imagen especular.

Utilizaremos frecuentemente otras expresiones cuantificadas, **de tipo entero** en lugar de booleano, que se han demostrado útiles en la escritura de predicados breves y legibles:

$\sum w : Q(w) : e(w)$ suma de las $e(w)$ tales que $Q(w)$

$\prod w : Q(w) : e(w)$ producto de las $e(w)$ tales que $Q(w)$

$\max w : Q(w) : e(w)$ máximo de las $e(w)$ tales que $Q(w)$

$\min w : Q(w) : e(w)$ mínimo de las $e(w)$ tales que $Q(w)$

$\# w : Q(w) : P(w)$ número de veces que se cumple $Q(w) \wedge P(w)$

(Q y P son predicados y e es una expresión entera)

Variables libres vs. variables ligadas

Es muy importante saber distinguir los dos tipos de variables que pueden aparecer en un predicado:

Variables ligadas Son las que están afectadas por un cuantificador ($\forall, \exists, \sum, \#, \max$, etc.), es decir se encuentran dentro de su ámbito.

Tienen algo detrás que las afecta. Estas variables las podemos cambiar por otras mientras que no estén libres o sean de programa.

Variables libres vs. variables ligadas

Es **muy importante** saber distinguir los dos tipos de variables que pueden aparecer en un predicado:

Variables ligadas Son las que están afectadas por un cuantificador ($\forall, \exists, \sum, \#, \max$, etc.), es decir se encuentran dentro de su ámbito.

Variables libres Son el resto de las variables que aparecen en el predicado.

- La intención es que estas variables se refieran a variables o parámetros del algoritmo que se está especificando.

Variables libres vs. variables ligadas

Es muy importante saber distinguir los dos tipos de variables que pueden aparecer en un predicado:

Las ligadas tienen cuantificadores, variables de conteo adheridas a si

Variables ligadas Son las que están afectadas por un cuantificador ($\forall, \exists, \sum, \#, \max$, etc.), es decir se encuentran dentro de su ámbito.

Variables libres Son el resto de las variables que aparecen en el predicado.

- La intención es que estas variables se refieran a variables o parámetros del algoritmo que se está especificando.

Variables del programa.

Por ejemplo, en el predicado

$$\forall w : 0 \leq w < n : \text{impar}(w) \rightarrow (\exists u : u \geq 0 : a[w] = 2^u)$$

las variables n y a son libres, mientras que w y u son ligadas.

Una variable ligada la podemos cambiar por otra variable que NO esté siendo usada, ni libre ni de programa.

Las ligadas son las que están afectadas por algún tipo de cuantificador. Mientras que las libres no las afecta ningún tipo de cuantificador.. Ella creo que las llama variables del programa.

Variables libres y ligadas

- Para evitar confusiones conviene que los nombres de las variables ligadas sean distintos de los de las variables libres:

$$V[i] = 7 \vee (\forall i : 0 \leq i < N : V[i] = x)$$

i está libre y ligada. Equivale a:

$$V[i] = 7 \vee (V[0] = x \wedge \dots \wedge V[N - 1] = x)$$

Variables libres y ligadas

- Para evitar confusiones conviene que los nombres de las variables ligadas sean distintos de los de las variables libres:

$$V[i] = 7 \vee (\forall i : 0 \leq i < N : V[i] = x)$$

i está libre y ligada. Equivale a:

$$V[i] = 7 \vee (V[0] = x \wedge \dots \wedge V[N - 1] = x)$$

- Las variables ligadas pueden renombrarse cuando sea necesario

$$V[i] = 7 \vee (\forall j : 0 \leq j < N : V[j] = x)$$

Formalmente, si v no aparece en Q ni en P , entonces:

$$(\partial w : Q(w) : P(w)) \equiv (\partial v : Q(v) : P(v))$$

donde ∂ representa un cuantificador cualquiera. Por ejemplo:

$$(\exists w : 0 \leq w < n : a[w] = x) \equiv (\exists v : 0 \leq v < n : a[v] = x)$$

Usar distintos nombres para las variables libres y las ligadas.

Variables libres y ligadas

- Dado el uso tan diferente de las variables libres y ligadas y que el nombre de estas últimas siempre se puede cambiar, utilizaremos en este curso la siguiente metodología:

*Las variables ligadas de un predicado **NUNCA** tendrán el mismo nombre que una variable del programa que está siendo especificado o verificado. Siempre que sea posible, usaremos las letras u, v, w, ... para nombrar variables ligadas.*

El nombre de una variable ligada debe ser DISTINTO que el de una variable de programa

VARIABLE DE PROGRAMA == VARIABLE LIBRE.

Semántica

- Utilizaremos predicados para definir **conjuntos de estados**.
- Un **estado** es una asociación de las variables del algoritmo con valores compatibles con su tipo.
- Por ejemplo, si x e y son variables de tipo entero, $\sigma = \{x \mapsto 3, y \mapsto 7\}$ representa un estado en el que la variable x vale 3, y la variable y vale 7.
- Un **estado** representa intuitivamente una “fotografía” de las variables de un algoritmo en un **instante concreto**.

Para representar diferentes casos o estados de un predicado definido utilizaremos sigma y asignaremos valores a las variables

X
3

- Utilizaremos **predicados para definir conjuntos de estados.**
- **Un estado** es una asociación de las variables del algoritmo con valores compatibles con su tipo. **Todos los posibles estados que p.ej: hacen $x \geq 0$ se cumpla.**
- Por ejemplo, si x e y son variables de tipo entero, $\sigma = \{x \mapsto 3, y \mapsto 7\}$ representa un estado en el que la variable x vale 3, y la variable y vale 7.
- Un estado representa intuitivamente una “fotografía” de las variables de un algoritmo en un instante concreto.
- Un estado σ **satisface** un predicado P si al sustituir en P las variables libres por sus valores en σ , el predicado se evalúa a **true**. Por ejemplo, $\sigma = \{x \mapsto 3, y \mapsto 7\}$ satisface $P \equiv y - x > 0$, pero no $Q \equiv x \bmod 2 = 0$.

σ \models $\{x \geq 0\}$ satisface

- **Identificaremos** los predicados con el **conjunto de estados que los satisfacen.**

- Suponiendo que x e y sean las únicas variables del algoritmo:
 - $P \equiv y - x > 0$ define los infinitos pares (x, y) en los y es mayor que x
 - $Q \equiv x \bmod 2 = 0$ define todos los pares (x, y) en los x es un número par.
- El predicado **true** define el conjunto de **todos** los estados posibles (i.e. cualquier variable del algoritmo puede tener cualquier valor de su tipo)
- El predicado **false** define el conjunto **vacio**.

(2,0, 2,1)

todos los estados lo satisfacen

vacio porque nunca es cierto

- El significado del predicado $\forall w : Q(w) : P(w)$ es equivalente a $P(w_1) \wedge P(w_2) \wedge \dots$, donde w_1, w_2, \dots son todos los valores de w que hacen cierto $Q(w)$.
 - Si este conjunto es vacío, entonces $\forall w : Q(w) : P(w) \equiv \text{true}$.

- El significado del predicado $\forall w : Q(w) : P(w)$ es equivalente a $P(w_1) \wedge P(w_2) \wedge \dots$, donde w_1, w_2, \dots son todos los valores de w que hacen cierto $Q(w)$.
 - Si este conjunto es **vacío**, entonces $\forall w : Q(w) : P(w) \equiv \text{true}$.
- El significado del predicado $\exists w : Q(w) : P(w)$ es equivalente a $P(w_1) \vee P(w_2) \vee \dots$, donde w_1, w_2, \dots son todos los valores de w que hacen cierto $Q(w)$.
 - Si este conjunto es **vacío**, entonces $\exists w : Q(w) : P(w) \equiv \text{false}$.

Semántica

Q es precondition y P es la postcondición

Para todo, son todos los valores del rango de valores.

- El significado del predicado $\forall w : Q(w) : P(w)$ es equivalente a $P(w_1) \wedge P(w_2) \wedge \dots$, donde w_1, w_2, \dots son todos los valores de w que hacen cierto $Q(w)$.
 - Si este conjunto es vacío, entonces $\forall w : Q(w) : P(w) \equiv \text{true}$.

El rango sobre el que defines el predicado, es cierto si el conjunto es vacío.

- El significado del predicado $\exists w : Q(w) : P(w)$ es equivalente a $P(w_1) \vee P(w_2) \vee \dots$, donde w_1, w_2, \dots son todos los valores de w que hacen cierto $Q(w)$.
 - Si este conjunto es vacío, entonces $\exists w : Q(w) : P(w) \equiv \text{false}$.

- Hay predicados que se satisfacen en todos los estados. Equivalen a **true**. Por ejemplo, $x > 0 \vee x \leq 0$ es cierto en todos los posibles estados.
- Hay otros que no se satisfacen en ninguno. Equivalen a **false**. Por ejemplo $\exists w : 0 < w < 1 : a[w] = 8$

No hay ningún número entre 0 y 1 por lo que es imposible y por tanto es false.

Por lo que tengo yo he entendido, si la postcondición existe, para el conjunto vacío el predicado es false, si es para todo, sería true para el conjunto vacío

- Por definición, el significado del resto de los cuantificadores cuando el rango $Q(w)$ al que se extiende la variable cuantificada **es vacío**, es el siguiente:

$$(\sum w : \text{false} : e(w)) = 0 \quad \text{Si el sumatorio es 0 entonces: false}$$

$$(\prod w : \text{false} : e(w)) = 1 \quad \text{Si el productorio vale 1 entonces false.}$$

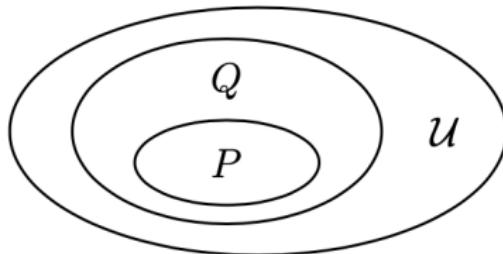
$$(\max w : \text{false} : e(w)) \quad \text{indefinido}$$

$$(\min w : \text{false} : e(w)) \quad \text{indefinido}$$

$$(\# w : \text{false} : P(w)) = 0 \quad \text{El número de veces que se cumple es 0 entonces false.}$$

Fuerza lógica de los predicados

- Diremos que un predicado P es **más fuerte** (resp. **más débil**) que otro Q , y lo expresaremos $P \Rightarrow Q$, cuando en términos de estados se cumpla $P \subseteq Q$, es decir todo estado que satisface P también satisface Q .

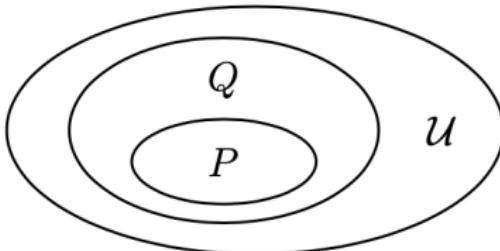


P es más fuerte que Q $p \Rightarrow q$ si todo estado que satisface p también satisface q.

Fuerza lógica de los predicados

- Diremos que un predicado P es **más fuerte** (resp. **más débil**) que otro Q , y lo expresaremos $P \Rightarrow Q$, cuando en términos de estados se cumpla $P \subseteq Q$, es decir todo estado que satisface P también satisface Q .

P = fuerte
 Q = Débil.



EL conjunto de predicados que satisface p , puede estar contenido en otros conjuntos.

Más fuerte->
mas restrictivo

- Ejemplos:

$$x > 0 \Rightarrow x \geq 0$$

$$x \geq 0 \Rightarrow x^2 \geq 0$$

$$P \wedge Q \Rightarrow P$$

$$P \wedge Q \text{ P Y Q} \Rightarrow Q$$

$$P \Rightarrow P \vee Q \text{ P o Q}$$

$$\forall w : 0 \leq w < 10 : a[w] \neq 0 \Rightarrow a[3] \neq 0$$

Los de la izquierda son más restrictivos que los de la derecha.

Fuerza lógica de los predicados

Cual es más restrictivo?

- $P: x \geq 0$

$$Q: x \geq 0 \wedge y \geq 0$$

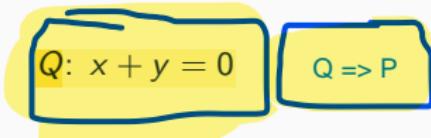
$$Q \Rightarrow P$$

Es más restrictivo Q ya que te dice que se tiene que cumplir $x \geq 0$ Y $y \geq 0$ mientras que el otro solo dice que x es mayor o igual que 0. Por tanto Q es más restrictivo que P: $Q \Rightarrow P$

Fuerza lógica de los predicados

- $P: x \geq 0$ $Q: x \geq 0 \wedge y \geq 0$ $Q \Rightarrow P$

Fuerza lógica de los predicados

- $P: x \geq 0$ $Q: x \geq 0 \wedge y \geq 0 \quad Q \Rightarrow P$
- $P: x \geq 0 \vee y \geq 0$ 

Fuerza lógica de los predicados

- $P: x \geq 0$ $Q: x \geq 0 \wedge y \geq 0$ $Q \Rightarrow P$
- $P: x \geq 0 \vee y \geq 0$ $Q: x + y = 0$ $Q \Rightarrow P$

todos los puntos de la recta están recogidos en cuadrantes de P, luego q es más fuerte que p

Fuerza lógica de los predicados

- $P: x \geq 0$ $Q: x \geq 0 \wedge y \geq 0$ $Q \Rightarrow P$
- $P: x \geq 0 \vee y \geq 0$ $Q: x + y = 0$ $Q \Rightarrow P$
- $P: x < 0$ $Q: x^2 + y^2 = 9$ estas dos no guardan relación luego ninguna es más restrictiva que la otra. Son independientes.

Fuerza lógica de los predicados

- $P: x \geq 0$ $Q: x \geq 0 \wedge y \geq 0$ $Q \Rightarrow P$
 - $P: x \geq 0 \vee y \geq 0$ $Q: x + y = 0$ $Q \Rightarrow P$
 - $P: x < 0$ $Q: x^2 + y^2 = 9$ $P \not\Rightarrow Q$ y $Q \not\Rightarrow P$
situación sin relación

Es importante porque a veces los enunciados no guardan relación

Fuerza lógica de los predicados

- $P: x \geq 0$ $Q: x \geq 0 \wedge y \geq 0$ $Q \Rightarrow P$
- $P: x \geq 0 \vee y \geq 0$ $Q: x + y = 0$ $Q \Rightarrow P$
- $P: x < 0$ $Q: x^2 + y^2 = 9$ $P \not\Rightarrow Q$ y $Q \not\Rightarrow P$
- $P: x \geq 1 \rightarrow x \geq 0$ $Q: x \geq 1$ Q=>P

Fuerza lógica de los predicados

- $P: x \geq 0$ $Q: x \geq 0 \wedge y \geq 0$ $Q \Rightarrow P$
- $P: x \geq 0 \vee y \geq 0$ $Q: x + y = 0$ $Q \Rightarrow P$
- $P: x < 0$ $Q: x^2 + y^2 = 9$ $P \not\Rightarrow Q$ y $Q \not\Rightarrow P$
- $P: x \geq 1 \rightarrow x \geq 0$ $Q: x \geq 1$ $Q \Rightarrow P$

Leyes de equivalencia

Leyes conmutativas, asociativas y de idempotencia de \wedge y \vee

- $P \wedge Q \Leftrightarrow Q \wedge P$
- $P \vee Q \Leftrightarrow Q \vee P$
- $P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R$
- $P \vee (Q \vee R) \Leftrightarrow (P \vee Q) \vee R$
- $P \wedge P \Leftrightarrow P$
- $P \vee P \Leftrightarrow P$

Son las tipicas propiedades que dabamos en mdl

Leyes distributivas, de absorción y de elemento neutro

- $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$
- $P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$
- $P \wedge (P \vee Q) \Leftrightarrow P$
- $P \vee (P \wedge Q) \Leftrightarrow P$
- $P \wedge \text{false} \Leftrightarrow \text{false}$
- $P \vee \text{true} \Leftrightarrow \text{true}$
- $P \wedge \text{true} \Leftrightarrow P$
- $P \vee \text{false} \Leftrightarrow P$

Leyes de equivalencia

Leyes de la negación

- $\neg\neg P \Leftrightarrow P$
- $P \vee \neg P \Leftrightarrow \text{true}$
- $P \wedge \neg P \Leftrightarrow \text{false}$

más propiedades que ya dimos en MDL

Leyes de De Morgan

- $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$
- $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$

Todo esto lo dimos en MMDL por lo tanto leerlo con cuidado cuando lo copiemos en el cuaderno.

Leyes de \rightarrow y \leftrightarrow

- $P \rightarrow Q \Leftrightarrow \neg P \vee Q$
- $P \leftrightarrow Q \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$

Especificación de funciones

Ha dicho que una precondicion sea true implica que la x puede tener cualquier valor.

Si postcondición

si pone {true}

Especificación de funciones

Por eso trabajamos en el Visual Studio.

- Identificaremos un algoritmo con la noción de función C++.
- Las funciones tienen el tipo del resultado devuelto. Si no devuelven ningún valor, se usa el tipo void.
- Los mecanismos de paso de parámetros distinguen entre paso por valor, paso por referencia y paso por referencia constante.
- A efectos de especificación es más ilustrativo saber si los parámetros son de

Lo que nos importa

- por valor **entrada** Su valor inicial es relevante para el algoritmo y éste **no debe** modificarlo. **paso de parámetro por valor.**
- salida** Su valor inicial es irrelevante para el algoritmo y éste **debe** almacenar algún valor en él.
- entrada/salida** Su valor inicial es relevante para el algoritmo, y además **este puede** modificarlo. **paso de parámetro por referencia.**
- referencia

Nos recomienda que primero determinemos (especifiquemos) si es por valor o por referencia. Después ya construimos los argumentos.

```
int f(int x, int & y) <<<=====>>> proc f(E(ntrada) int x, E(ntrada)/S(alida) y, S(alida) int resultado)
```

Especificación de funciones

Mejor utilizar proc, si utilizamos func, los parámetros de la función son todos de entrada y no haría falta especificarlo. Pero si ponemos proc podemos poner el tipo de los parámetros de entrada.

- Por ello, usaremos en la especificación dos tipos de **cabeceras virtuales** que el programador habrá de traducir después a la cabecera C++ que le parezca.

fun todos entrada → **fun nombre (tipo₁ p₁, ..., tipo_n p_n) dev tipo r**
entrada y salida y hay que definir el tipo → **proc nombre (cualif tipo₁ p₁, ..., cualif tipo_n p_n)**

donde *cualif* será vacío si el parámetro es de entrada, **out** si es de salida, o **inout** si es de entrada/salida. Los parámetros de una cabecera **fun** se entienden siempre de entrada. Si ponemos fun entonces todos los parámetros de la función son de entrada

- La primera se usará para algoritmos que devuelvan un solo valor, y la segunda para los que no devuelvan nada, devuelvan más de un valor, o/y modifiquen sus parámetros.

in == E
out == S
innout == E/S

paso de parámetro por valor in == E
función devuelve algo dev == S
paso de parámetro por referencia in/out == E/S

Ejemplos

- Calcular el cociente por defecto y el resto de la división entera de dos naturales.

Antes de realizar la especificación voy a decir lo primero que P es mi precondición y que Q es mi postcondición. A es mi función o procedimiento.

- P: {dividendo $\geq 0 \wedge$ divisor $> 0\}$ Divisor no puede ser 0 porque arithmetical error.
- A: proc{in dividendo, in divisor, in/out cociente, in/out resto} procedimiento porque tiene parámetros de entrada y de salida.
- Q: {dividendo = divisor * cociente + resto}

La postcondición debe cumplirse siempre que se respete la entrada de la precondición.

Ejemplos

- Calcular el cociente por defecto y el resto de la división entera de dos naturales.

Si escribimos

$$\{a \geq 0 \wedge b > 0\}$$

proc divide (int a, int b, out int q, out int r)

$$\{a = q \times b + r\}$$

cociente resto

num y denomin

dividendo = divisor x cociente + resto

el algoritmo

q=0; r=a;

cumpliría con la especificación. Pero no es lo que queremos. El problema es que la

postcondición es demasiado débil.

Tendríamos que añadir que el resto sea más pequeño que el divisor.

Ejemplos

- Calcular el cociente por defecto y el resto de la división entera de dos naturales.

Si escribimos

$$\{a \geq 0 \wedge b > 0\}$$

proc divide (int a, int b, out int q, out int r)

$$\{a = q \times b + r\} \text{ ademas } 0 <= r < \text{divisor.}$$

el algoritmo

Claro, si por ejemplo dividimos 10/5 y decimos que el cociente sea 0 entonces el resto = dividendo y esto no lo queremos, no es posible es una propiedad

$$q=0; \quad r=a; \quad \text{si el cociente es 0 el resto} = a$$

cumpliría con la especificación. Pero no es lo que queremos. El problema es que la postcondición es demasiado débil.

Reforzamos la postcondición:

dividendo = divisor
por cociente + resto.

$$\{a \geq 0 \wedge b > 0\}$$

proc divide (int a, int b, out int q, out int r)

$$\{a = q \times b + r \wedge 0 \leq r < b\}$$

Por conocimientos elementales de matemáticas sabemos que sólo existen dos números naturales que satisfacen lo que exigimos a q y r .

Ejemplos

- Copiar el valor de una variable en otra.

Igual que antes: precondition P y postcondición = Q, función A; si solo tiene parámetros de entrada y procedimiento (proc) si tiene parámetros de entrada/salida.

- P: {true} para cualquier entrada, vale cualquier número.
- A: proc copiaValor{in x, in/out y}
- Q: {x=y}

Ejemplos

No dice que intercambiemos valores de variables.

- Copiar el valor de una variable en otra.

{true} Para cualquier valor de entrada

```
fun copiar (int x) dev int y  
{x = y}
```

Admitimos que está implícito que el valor de la variable x no cambia, ya que la variable x es de entrada, al ser *copiar* una función.

Ejemplos

- Copiar el valor de una variable en otra.

```
{true}  
fun copiar (int x) dev int y  
{x = y}
```

Admitimos que está implícito que el valor de la variable x no cambia, ya que la variable x es de entrada, al ser *copiar* una función.

- Calcular el máximo de tres enteros. Cualesquiera

precondición: {true} para cualquier valor de entrada.

func maximo3 (in int x,in int y, in int z) dev int m

postcondición: {m>=x ^ m>=y ^ m>=z ^ (m ==x ó m==y ó m ==z)}

Ejemplos

- Copiar el valor de una variable en otra.

```
{true}  
fun copiar (int x) dev int y  
{x = y}
```

Admitimos que está implícito que el valor de la variable x no cambia, ya que la variable x es de entrada, al ser *copiar* una función.

- Calcular el máximo de tres enteros.

```
{true}  
fun maximo3 (int x, y, z) dev int m  
{m ≥ x ∧ m ≥ y ∧ m ≥ z ∧ (m = x ∨ m = y ∨ m = z)}
```

Porque tiene que ser mayor que todos pero solo igual que una de las variables.

Ejemplos

- Calcular el máximo de las primeras n posiciones de un vector de enteros no vacío.

precondición: $\{0 \leq n < v.size()\}$ porque son vectores

función: func maxPos(in vector<int> v, in int n) int max

postcondición: $\{(w : 0 \leq w < n : m \geq a[w]) \wedge (w : 0 \leq w < n : m = a[w])\}$

una opción de postcondición más sencilla es: $\{\max w : 0 \leq w < n : v[w]\}$

º P: $\{0 \leq n < v.size()\} //$ queremos el máximo de n posiciones

º A: proc maximoNPosVector{in vct, in n, in/out max}

º Q: max= $\{\max i : 0 \leq i < n : V[i]\}$

Ejemplos

- Calcular el máximo de las primeras n posiciones de un vector de enteros no vacío.

$$\{0 < n \leq longitud(a)\}$$

```
fun maximo (int a[], int n) dev int m
```

$$\{(\forall w : 0 \leq w < n : m \geq a[w]) \wedge (\exists w : 0 \leq w < n : m = a[w])\}$$

utilizando los vectores no habría ese problema.

- La precondición $n > 0$ es necesaria para asegurar que el rango del existencial no es vacío. Una postcondición **false** solo la satisfacen los algoritmos que no terminan.
- Nótese que la segunda conjunción de la postcondición es necesaria. Si no, el implementador malévolos podría devolver un número muy grande pero no necesariamente en el vector. Una postcondición más sencilla sería.

$$\{m = \max w : 0 \leq w < n : a[w]\}$$

- La segunda conjunción de la precondición requiere que el vector parámetro real tenga una longitud de al menos n .

Ejemplos

- Intercambiar los valores de dos variables.

°P: { $x = X \wedge y = Y$ }

°A: proc intercambiarVariable{in/out x, in/out y} dentro debemos de usar una variable aux
°Q:{ $x = Y \wedge y = X$ }

Ejemplos

- Intercambiar los valores de dos variables.

Si escribimos

```
{x = X}  $\wedge$  {y = Y}  
proc intercambia (inout int x, inout int y)  
{x = y}  $\wedge$  {y = x}
```

NO está bien especificado porque con esto estamos diciendo que x e y valen iguales. Lo que nosotros queremos decir es que intercambien valores.

entrada salida
porque
intercambiamos los
valores.

Ejemplos

- Intercambiar los valores de dos variables.

Si escribimos

$$\{x = X \wedge y = Y\}$$

proc *intercambia* (**inout int** *x*, **inout int** *y*)

$$\{x = y \wedge y = x\}$$

estamos expresando que *x* e *y* terminan valiendo lo mismo y eso no es lo que queríamos expresar.

Ejemplos

- Intercambiar los valores de dos variables.

Si escribimos

$$\{x = X \wedge y = Y\}$$

`proc intercambia (inout int x, inout int y)`

$$\{x = y \wedge y = x\}$$

estamos expresando que x e y terminan valiendo lo mismo y eso no es lo que queríamos expresar.

La aparición de una variable en la postcondición representa el valor de la variable al terminar la ejecución del método. Para hacer referencia al valor que tenía antes de la ejecución del método usamos variables auxiliares X , Y :

$$\{x = X \wedge y = Y\}$$

`proc intercambia (inout int x, inout int y)`

$$\{x = Y \wedge y = X\}$$

Es decir, nombramos o le damos un valor anterior al cambio para después poder cambiarlo x vale el valor viejo de y e y vale el valor viejo de x.

Ejemplos

- Positivizar un vector: **consiste en reemplazar los valores negativos por ceros.**
 - P: { $0 \leq i < v.size()$ } //Para todo valor de un vector
 - A: proc cambiarVector{in/out vector} cambia todo el vector
 - Q: {p.t i: $0 \leq i < v.size()$: $v[i] = 0$ }

Ejemplos

- **Positivizar un vector:** consiste en reemplazar los valores negativos por ceros.

Si escribimos

$$\{0 \leq n \leq \text{longitud}(a) \wedge a = A\}$$

proc *positivizar* (**inout int** *a[]*, **int** *n*)

$$\{\forall w : 0 \leq w < n : A[w] < 0 \rightarrow a[w] = 0\}$$

para todas las posiciones del vector: si antes habia un negativo, ahora hay un 0.

Pero falta decir que si, el valor es positivo, sigue valiendo ese valor positivo.

- **Positivizar un vector:** consiste en reemplazar los valores negativos por ceros.

Si escribimos

$$\{0 \leq n \leq longitud(a) \wedge a = A\}$$

proc *positivizar* (**inout int** *a*[], **int** *n*)
 $\{\forall w : 0 \leq w < n : A[w] < 0 \rightarrow a[w] = 0\}$

un implementador malévolo podría escribir este código cumpliendo la especificación:

```
for (int i=0; i<n; i++) {a[i]=0;}
```

Ejemplos

- **Positivizar un vector:** consiste en reemplazar los valores negativos por ceros.

Si escribimos

$$\{0 \leq n \leq longitud(a) \wedge a = A\}$$

proc *positivizar* (**inout int** *a*[], **int** *n*) devuelve el vector con
 $\{\forall w : 0 \leq w < n : A[w] < 0 \rightarrow a[w] = 0\}$ la modificación

un implementador malévolo podría escribir este código cumpliendo la especificación:

for (**int** *i*=0; *i*<*n*; *i*++) {*a*[*i*]=0;} todos los valores son 0, lo cual no queremos. Tenemos que especificar que
Para evitarlo reforzamos la postcondición: pasa si es >0.

$$\{0 \leq n \leq longitud(a) \wedge a = A\}$$

proc *positivizar* (**inout int** *a*[], **int** *n*)

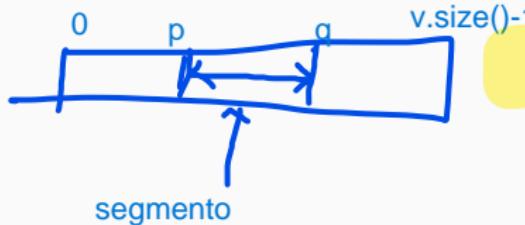
$\{\forall w : 0 \leq w < n : (A[w] < 0 \rightarrow a[w] = 0) \wedge (A[w] \geq 0 \rightarrow a[w] = A[w])\}$
si es positivo

Ejemplos

Estos son ejercicios que en 1º haciamos con coste CUADRATICO. Ahora en segundo los haremos con coste lineal.

- Calcular el máximo de las sumas de los elementos de los segmentos de un vector de enteros dado.

Los segmentos de un vector son los subvectores que se obtienen tomando diferentes subintervalos del intervalo de índices del vector.



Objetivo: representar estos segmentos.

Representarlos:

1: [2,8) sin incluir (profe)

2: [2,8] incluyendo

Un segmento viene definido por un intervalo de índices del vector.

Si $p==q$ es el conjunto vacío.

PARA EL 1: $0 \leq p \leq q \leq v.size()$

$[0, v.size()] \rightarrow$ representación de todos los segmentos de un v.

Ejemplos

- Calcular el máximo de las sumas de los elementos de los segmentos de un vector de enteros dado.

Los segmentos de un vector son los subvectores que se obtienen tomando diferentes subintervalos del intervalo de índices del vector.

- Consideramos los segmentos $[p..q)$ en los que $0 \leq p \leq q \leq n$. Cuando $p = q$ tenemos el segmento vacío.

Ejemplos

- Calcular el máximo de las sumas de los elementos de los segmentos de un vector de enteros dado.

Los segmentos de un vector son los subvectores que se obtienen tomando diferentes subintervalos del intervalo de índices del vector.

- Consideramos los segmentos $[p..q)$ en los que $0 \leq p \leq q \leq n$. Cuando $p = q$ tenemos el segmento vacío.
- Si $n = 0$ el vector es vacío por lo que solo hay un segmento, $[0..0)$, el vacío de suma 0, y el resultado es por tanto 0.

Ejemplos

- Calcular el máximo de las sumas de los elementos de los segmentos de un vector de enteros dado.

Los segmentos de un vector son los subvectores que se obtienen tomando diferentes subintervalos del intervalo de índices del vector.

- Consideramos los segmentos $[p..q)$ en los que $0 \leq p \leq q \leq n$. Cuando $p = q$ tenemos el segmento vacío.
- Si $n = 0$ el vector es vacío por lo que solo hay un segmento, $[0..0)$, el vacío de suma 0, y el resultado es por tanto 0.
- Para poder expresar el segmento $[0..n)$ entonces ha de ser $q \leq n$ para no olvidar ningún valor.

Ejemplos

- Calcular el máximo de las sumas de los elementos de los segmentos de un vector de enteros dado.

Los segmentos de un vector son los subvectores que se obtienen tomando diferentes subintervalos del intervalo de índices del vector.

- Consideramos los segmentos $[p..q]$ en los que $0 \leq p \leq q \leq n$. Cuando $p = q$ tenemos el segmento vacío. El primero cerrado y el segundo abierto
- Si $n = 0$ el vector es vacío por lo que solo hay un segmento, $[0..0)$, el vacío de suma 0, y el resultado es por tanto 0.
- Para poder expresar el segmento $[0..n)$ entonces ha de ser $q \leq n$ para no olvidar ningún valor.

$$\{0 \leq n \leq longitud(a)\}$$

fun maxSumaSeg (int a[], int n) dev int m

$$\{m = (\max p, q : 0 \leq p \leq q \leq n : (\sum i : p \leq i < q : V[i]))\}$$

n son los elementos del vector.

Hay un MONTÓN de ejercicios que sean del palo de este.

Ejemplos

- Calcular el máximo de las sumas de segmentos **NO VACÍOS** de un vector de enteros dado.

Ejemplos

- Calcular el máximo de las sumas de segmentos **NO VACÍOS** de un vector de enteros dado.
 - No podemos admitir $n = 0$, ya que no hay segmentos no vacíos en un vector vacío y por tanto el máximo no está bien definido.

Ejemplos

- Calcular el máximo de las sumas de segmentos **NO VACÍOS** de un vector de enteros dado.
 - No podemos admitir $n = 0$, ya que no hay segmentos no vacíos en un vector vacío y por tanto el máximo no está bien definido.
 - Consideramos los segmentos $[p..q)$ en los que $p < q$. Así, cuando $q = p + 1$ tenemos el segmento unitario.

Ejemplos

- Calcular el máximo de las sumas de segmentos **NO VACÍOS** de un vector de enteros dado.
 - No podemos admitir $n = 0$, ya que no hay segmentos no vacíos en un vector vacío y por tanto el máximo no está bien definido.
 - Consideramos los segmentos $[p..q)$ en los que $p < q$. Así, cuando $q = p + 1$ tenemos el segmento unitario.
 - Para poder expresar el segmento $[0..n)$ entonces ha de ser $q \leq n$ para no olvidar ningún valor.

Ejemplos

Van a haber muchos ejercicios de segmentos. []

menor estricto para los no vacíos.

- Calcular el máximo de las sumas de segmentos **NO VACÍOS** de un vector de enteros dado.

- No podemos admitir $n = 0$, ya que no hay segmentos no vacíos en un vector vacío y por tanto el máximo no está bien definido.
- Consideramos los segmentos $[p..q)$ en los que $p < q$. Así, cuando $q = p + 1$ tenemos el segmento unitario.
- Para poder expresar el segmento $[0..n)$ entonces ha de ser $q \leq n$ para no olvidar ningún valor.

No vacíos.

$$\{0 < n \leq longitud(a)\}$$

Esto es lo que cambia

```
fun maxSumaSegNV (int a[], int n) dev int m
```

$$\{m = (\max p, q : 0 \leq p < q \leq n : S(a, p, q))\}$$

donde

$$S(v, c, f) \equiv (\sum u : c \leq u < f : v[u])$$

LEERME ESTO MUY BIEN EN CASA PORQUE LO HA EXPLICADO MUY RÁPIDO.

Ejemplos

- Segmento de longitud más larga de elementos iguales.

Ejemplos

Imagina un vector: 1 2 2 3 5 7 7 7 8 9

Longitud de un vector abierto $[p,q) = q-p$

Longitud de un vector cerrado $[p,q] = q-p+1$.

- Segmento de longitud más larga de elementos iguales.

```
{0 ≤ n ≤ longitud(a)}  
fun longMaxIguales (int a[], int n) dev int /  
{l = (max p, q : 0 ≤ p ≤ q ≤ n ∧ iguales(a, p, q) : q - p)}
```

donde

$\text{iguales}(v, c, f) \equiv \forall u, w : c \leq u, w < f : v[u] = v[w]$

cada pareja de entre los extremos son iguales.

Si es abierto: $\text{iguales}(v, c, f) = \exists t. \forall u : c \leq u \leq f - 1 : v[u] = v[u-1]$.