

## **INDEX**

|   |           |
|---|-----------|
| <b>ALGORITMICA .....</b>                            | <b>3</b>  |
| <b>TIPUS DE DADES.....</b>                          | <b>3</b>  |
| OBJECTES ELEMENTALS DEL LENGUATGE ALGORÍSMIC.....   | 3         |
| • Constant: .....                                   | 4         |
| • Variable: .....                                   | 4         |
| <b>Tipus elementals .....</b>                       | <b>4</b>  |
| Tipus booleà .....                                  | 5         |
| Tipus caràcter .....                                | 6         |
| Tipus enter .....                                   | 8         |
| Tipus real .....                                    | 9         |
| <b>Declaració d'objectes.....</b>                   | <b>10</b> |
| <b>Expressions .....</b>                            | <b>12</b> |
| <b>Sintaxis .....</b>                               | <b>13</b> |
| <b>Semàntica .....</b>                              | <b>13</b> |
| <b>Avaluació .....</b>                              | <b>14</b> |
| Exemples d'avaluació d'expressions: .....           | 14        |
| Definició de tipus. Tipus enumeratius .....         | 16        |
| <b>Funcions de conversió de tipus .....</b>         | <b>17</b> |
| <b>ESPECIFICACIÓ D'ALGORISMES .....</b>             | <b>18</b> |
| <b>Algorisme i canvi d'estat .....</b>              | <b>18</b> |
| QUÈ VOL DIR ESPECIFICAR? .....                      | 20        |
| <b>Elements del'especificació .....</b>             | <b>23</b> |
| <b>Especificació i comentaris.....</b>              | <b>25</b> |
| Exemples d'especificació .....                      | 26        |
| Intercanvi de dues variables .....                  | 26        |
| Arrel quadrada.....                                 | 26        |
| Arrel quadrada entera .....                         | 26        |
| Quocient i residu .....                             | 27        |
| Nombre de divisors .....                            | 27        |
| <b>ESTRUCTURES ALGORÍSMIQUES .....</b>              | <b>27</b> |
| <b>Estructura general d'un algorisme .....</b>      | <b>27</b> |
| 1) Capçalera: .....                                 | 28        |
| 2) Definició de constants: .....                    | 28        |
| 3) Definició de tipus: .....                        | 28        |
| 4) Declaració de variables: .....                   | 28        |
| 5) Cos de l'algorisme: .....                        | 28        |
| 6) Final de l'algorisme: .....                      | 28        |
| <b>ACCIONS ELEMENTALS .....</b>                     | <b>30</b> |
| <b>L'assignació.....</b>                            | <b>30</b> |
| <b>Composició d'accions.....</b>                    | <b>33</b> |
| <b>Composició seqüencial.....</b>                   | <b>33</b> |
| Intercanvi de valors.....                           | 35        |
| Impostos i salari net.....                          | 36        |
| Interessos .....                                    | 36        |
| <b>Composició alternativa.....</b>                  | <b>37</b> |
| Màxim de dos nombres enters .....                   | 39        |
| <b>Composició iterativa .....</b>                   | <b>40</b> |
| Especificació.....                                  | 42        |
| Exemple.....  | 43        |
| Multiplicació de dos enters .....                   | 43        |
| El factorial.....                                   | 44        |
| <b>Finalització de la composició iterativa.....</b> | <b>45</b> |

---

|   |                  |
|---|------------------|
| <b><i>Variants de la composició iterativa.....</i></b>              | <b><i>46</i></b> |
| <b><i>Accions i funcions.....</i></b>                               | <b><i>47</i></b> |
| <b><i>Accions .....</i></b>   | <b><i>49</i></b> |
| <b><i>En certes aplicacions,.....</i></b>                           | <b><i>50</i></b> |
| <b><i>Paràmetres.....</i></b>                                       | <b><i>51</i></b> |
| <b><i>Funcions .....</i></b>  | <b><i>57</i></b> |
| <b><i>Accions i funcions predefinides.....</i></b>                  | <b><i>59</i></b> |
| <b><i>Funcions de conversió de tipus .....</i></b>                  | <b><i>59</i></b> |
| <b><i>Accions i funcions d'entrada i sortida de dades .....</i></b> | <b><i>59</i></b> |
| <b><i>Algorisme producteNaturals .....</i></b>                      | <b><i>60</i></b> |
| <b><i>Resum .....</i></b>   | <b><i>61</i></b> |

## ALGORITMICA

### Tipus de Dades.

La observació en la vida real podem entendre de què tot gira en accions sobre elements que configurant un objecte.

Podem classificar els tipus de dades estructurats segons diversos paràmetres, que esmentem a continuació:

1) El tipus d'elements que agrupem:

- Tots del mateix tipus: estructura homogènia
- De tipus diferents: estructura heterogènia

2) La manera com accedim als elements:

- Accés seqüencial: l'accés als elements es fa en el mateix ordre segons el qual estan col·locats. Per a obtenir l'element de la  $i$ -èsima posició abans hem d'obtenir els  $i - 1$  elements anteriors.
- Accés directe: podem obtenir directament qualsevol dels elements de l'estructura.

3) El nombre d'elements que pot contenir:

- Mida fixa: el nombre d'elements es defineix en la declaració i no pot variar en temps d'execució.
- Mida variable: en temps d'execució pot variar el nombre d'elements.

### Objectes elementals del llenguatge algorísmic

Tot i que encara no sabem com dissenyar algorismes, el que sí sabem és que hauran de tractar amb dades. Quan dissenyem un algorisme, necessitarem referenciar aquestes dades d'alguna manera i explicitar quins comportaments esperem de les dades. Per a assolir aquest objectiu, introduïm el concepte d'objecte, que serà el suport per a mantenir les dades de què tracti un algorisme.

Sovint, quan pensem en un objecte concret, ens ve al cap, entre altres coses possibles, un ventall d'accions o operacions que s'hi poden fer, i altres accions que no s'hi poden fer.

Un objecte té tres atributs:

- Nom o identificador
- Tipus
- Valor

El nom ens permet identificar unívocament l'objecte. El tipus indica el conjunt de valors que pot tenir i quines operacions s'hi poden aplicar.

El valor d'un objecte no ha de ser necessàriament un número. El valor d'un objecte serà un element del conjunt al qual pertany i que ve indicat pel seu tipus corresponent. A més, hi haurà objectes en què el seu valor podrà ser canviat per un altre del mateix tipus, és a dir, que es pot modificar al llarg del temps d'execució de l'algorisme.

Per tant, un objecte podrà ser:

- **Constant:**

El seu valor no és modificable.

- **Variable:**

El seu valor es pot modificar.

### Tipus elementals

Passem a veure quins tipus té inicialment el llenguatge per a poder declarar els nostres objectes. Són els anomenats *tipus elementals* i són quatre: **booleà**, **caràcter**, **enter** i **real**. Més endavant veurem que el llenguatge ens permetrà de definir els nostres tipus particulars.

Per a descriure cada tipus, exposarem l'identificador o nom amb què el llenguatge el reconeix, quins són els valors que pot prendre, quines operacions admet, i quina és la sintaxi reconeguda pel llenguatge dels seus valors.

## Tipus booleà

El tipus booleà és un dels més utilitzats i el seu origen està en l'àlgebra de Boole. En molts casos, els seus dos valors serveixen per a representar la certesa o falsedat d'un estat concret de l'entorn. També se l'anomena tipus lògic.

El llenguatge algorísmic el reconeix mitjançant l'identificador boolea. Només té dos valors possibles: cert i fals. El llenguatge algorísmic reconeixerà les paraules cert i fals com els valors booleans. Les operacions internes (aquel·les que tornen un valor del mateix tipus) són no (negació), i (conjunció), o (disjunció).

Els operadors lògics no, i, o (negació, "i" lògica, "o" lògic respectivament) es defineixen mitjançant la taula següent:

| <i>a</i> | <i>b</i> | <i>no a</i> | <i>a i b</i> | <i>a o b</i> |
|----------|----------|-------------|--------------|--------------|
| cert     | cert     | fals        | cert         | fals         |
| cert     | fals     | fals        | fals         | cert         |
| fals     | cert     | cert        | fals         | cert         |
| fals     | fals     | cert        | fals         | fals         |

En la taula següent tenim les característiques del tipus booleà:

|                   |                            |
|-------------------|----------------------------|
| Identificador     | boolea                     |
| Rang de valors    | cert, fals                 |
| Operadors interns | no, i, o, =, ≠, <, ≤, >, ≥ |
| Operadors externs |                            |
| Sintaxi de valors | cert, fals                 |

Observeu que en l'apartat d'operadors interns hem introduït altres símbols, els operadors relacionals (=, ≠, <, ≤, >, ≥), que tot seguit us explicarem en l'apartat següent.

## Operadors relacionals

Introduïm uns altres operadors amb els quals possiblement no estigueu familiaritzats.

Són els operadors relacionals i, com veurem més endavant, poden operar sobre altres tipus.

Els operadors  $=$ ,  $\neq$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$  s'anomenen *operadors relacionals* i s'apliquen a dos operands del mateix tipus. A més, els operadors relacionals  $<$ ,  $\leq$ ,  $>$ ,  $\geq$  només es poden aplicar si el tipus té una relació d'ordre. Per conveni, en el tipus booleà es té el següent:

**fals**  $<$  **cert**

Així, doncs, **fals**  $<$  **cert** és una operació que dóna com a valor **cert**. L'operació **cert**  $=$  **fals**, dóna el valor de **fals**, **cert**  $\leq$  **cert** dóna el valor de **cert**, etc.

A continuació teniu una taula pels operadors  $=$ ,  $<$ ,  $>$ :

| <i>a</i> | <i>b</i> | $a = b$ | $a < b$ | $a > b$ |
|----------|----------|---------|---------|---------|
| cert     | cert     | cert    | fals    | fals    |
| cert     | fals     | fals    | fals    | cert    |
| fals     | cert     | fals    | cert    | fals    |
| fals     | fals     | cert    | fals    | fals    |

I les expressions següents cobriran la resta de casos que podem trobar amb els operadors relacionals:

$a \neq b$  és equivalent a **no**( $a = b$ ),

$a \geq b$  és equivalent a  $(a > b) \text{ o } (a = b)$ , i

$a \leq b$  és equivalent a  $(a < b) \text{ o } (a = b)$

## Tipus caràcter

Nosaltres ens podem comunicar mitjançant textos que es construeixen a partir d'uns símbols anomenats *caràcters*. El tipus més elemental perquè un algorisme pugui gestionar informació simbòlica és el tipus caràcter.

L'identificador del tipus és **caràcter**. La sintaxi que reconeix el llenguatge algorísmic dels

seus valors és el mateix caràcter delimitat per cometes simples.

Per exemple, 'e' és el valor de caràcter corresponent a la lletra "e", '=' és el valor corresponent al símbol d'igual, i ' ' és el caràcter corresponent a l'espai.

El rang de valors que pot tenir aquest tipus depèn de la codificació utilitzada pel computador. En tot cas, serà un conjunt finit de valors.

Dins aquest conjunt de caràcters hi ha definida una relació d'ordre. Això ens permet poder aplicar els operadors relacionals que seran externs al conjunt. En el cas dels caràcters que representen les lletres que coneixem, l'ordre segueix un criteri alfabètic. Per exemple, el valor de l'operació 'a' < 'b' és **cert**, 'z' < 'a' és **fals**. Fixeu-vos que el caràcter 'A' és diferent de 'a', i per tant, 'A' = 'a' és **fals**.

Podem parlar, doncs, d'un ordre alfabètic entre els caràcters que representen les lletres minúscules, i un altre ordre entre els caràcters que representen les majúscules.

També podrem parlar d'un ordre entre els caràcters dígit ('0', '1', ..., '9').

Per tant, '0' < '5' és **cert**.

Fixeu-vos que parlem d'ordres entre subconjunts de caràcters que ens poden ser útils. També hi ha un ordre entre aquests conjunts que correspon a un conveni de la codificació utilitzada per a representar els caràcters (vegeu el text al marge). En el cas fer servir la codificació ASCII, les minúscules sempre són més grans que les majúscules, i les majúscules més grans que els caràcters dígit.

En resum:

|                   |  |
|-------------------|--|
| Identificador     | caracter   |
| Rang de valors    | Conjunt finit de valors que depenen d'un codi usat pel computador.                       |
| Operadors interns |  |
| Operadors externs | =, ≠, <, ≤, >, ≥   |
| Sintaxi de valors | Exemples: 'a', 'W', '1'. Observeu que el caràcter '1' no té res a veure amb el número 1. |

### Tipus enter

Hauria estat bé parlar del tipus número com nosaltres l'entenem, que ens permetés representar tant nombres enters com nombres reals, és a dir, amb part fraccionària. En programació, haurem de distingir, però, els enters dels reals, i malgrat que en la vida real considerem els enters com un subconjunt dels reals, en programació els tractarem com a dos conjunts diferents que no tenen cap relació. Això és degut a la representació interna de cada tipus. El tipus enter s'identifica amb el nom reservat **enter**. Tot i això, com que els ordinadors tenen memòria finita, no podrem tenir tots els enters que vulguem, sinó un subconjunt finit que estarà limitat per un enter mínim *ENTERMIN* i un enter màxim *ENTERMAX*. Els valors *ENTERMIN* i *ENTERMAX* dependran del computador que fem servir. La sintaxi dels valors vindrà donada pel seguit de xifres que representa el valor enter precedit del signe menys (-) quan l'enter és negatiu. Així, 4 és un enter, 4876 és un altre enter, i -10 és un altre enter. Noteu que '5' no és un enter, sinó el caràcter "5". Les operacions internes que es poden fer amb els enters són: el canvi de signe (-), la suma (+), la resta (-), el producte (\*), la divisió entera (**div**), i la resta de la divisió o mòdul (**mod**).

Les operacions **div** i **mod** són les que us poden resultar més estranyes. Exemples de l'operació

**div** els teniu a continuació:

$$8 \text{ div } 4 = 2, 7 \text{ div } 2 = 3 \text{ i } 3 \text{ div } 8 = 0$$

Exemples de l'operació **mod**:

$$8 \text{ mod } 4 = 0, 7 \text{ mod } 2 = 1 \text{ i } 3 \text{ mod } 8 = 3$$

Les operacions **div** i **mod** estan definides pel teorema de la divisió entera d'Euclides:

Per tot dividend positiu i divisor positiu i diferent de 0, l'equació:

$$\text{dividend} = \text{divisor} * \text{quocient} + \text{resta}$$

té un únic valor de quocient i un únic valor de resta, si la resta compleix que és més gran o

igual que 0, i a la vegada és més petita que el divisor ( $0 \leq \text{resta} < \text{divisor}$ ) I aquests valors únics són  $\text{quocient} = \text{dividend} \text{ div } \text{divisor}$ , i  $\text{resta} = \text{dividend} \text{ mod } \text{divisor}$



La divisió entera per 0 produirà un error

Evidentment, hi ha una relació d'ordre entre els enters que permet fer servir els operadors booleans ( $4 < 10$  és **cert**,  $235 > 4000$  és **fals**).

En resum:

|                   |   |
|-------------------|---|
| Identificador     | enter   |
| Rang de valors    | Des de <i>ENTERMIN</i> fins <i>ENTERMAX</i> . |
| Operadors interns | - (Canvi signe), +, -, *, div, mod            |
| Operadors externs | Relacionals: =, ≠, <, ≤, >, ≥                 |
| Sintaxi de valors | Exemples: 3, 465454, -899993                  |

## Tipus real

Com hem avançat en el darrer apartat, un altre tipus per a representar números és el tipus real. De la mateixa manera que els enters, estarà limitat per un valor mínim (*REALMIN*), i un valor màxim (*REALMAX*). Aquests valors dependran de la representació interna del computador. A més, només es poden representar un nombre finit de valors del rang. Penseu que entre dos nombres reals qualssevol, hi ha un nombre infinit de reals, i l'ordinador és finit.

Com que només es pot representar un conjunt finit de reals, els programes que treballen amb reals estan sotmesos a problemes de precisió i errors en els càlculs.

Des d'un punt de vista informàtic, els enters no tenen res a veure amb els reals, ja que el comportament en el computador d'ambdós tipus, i també les seves respectives representacions internes en el computador, són diferents.

Els reals comparteixen amb els enters els símbols d'operadors +, -, \* (suma, resta, producte). Malgrat la coincidència de símbols, cal que considereu els símbols com operacions diferents de les que s'apliquen als enters, en el sentit que, per exemple, la suma de reals donarà un resultat real, mentre que la suma d'enters donarà un resultat enter. També tenen l'operació de divisió (/). Fixeu-vos que la divisió per 0 produeix un error.

La sintaxi de valors es fa mitjançant una successió de dígitos on apareix el caràcter "." per a denotar la coma decimal. Per exemple,

49.22, 0.5, 0.0001

Per a representar "10 elevat a" alguna cosa, farem servir el caràcter E. Per exemple, 5.0E-8 correspon a  $5 \cdot 10^{-8}$  o 0.00000005.

Observeu que la sintaxi de reals difereix de la d'enters per la presència del punt. Així, si veiem escrit "5" sabrem que correspondrà a l'enter 5, mentre que si veiem escrit "5.0", sabrem que correspon al real 5. Malgrat que per a nosaltres correspon al mateix número, el computador el tractarà diferent.

En resum:

|                   |  |
|-------------------|--|
| Identificador     | real   |
| Rang de valors    | Des de <i>REALMIN</i> fins a <i>REALMAX</i> i no pas tots els valors que estan en el rang. |
| Operadors interns | – (Canvi signe), +, –, *, /  |
| Operadors externs | Relacionals: =, ≠, <, ≤, >, ≥  |
| Sintaxi de valors | Exemples: 0.6, 5.0E–8, –49.22E + 0.8, 1.0E5, 4.0   |

## Declaració d'objectes

Abans que un algorisme faci servir un objecte, cal que aquest s'hagi declarat prèviament.

Si l'objecte és constant, s'ha de declarar dins un apartat afitat per les paraules clau **const ... fconst**. Si és un objecte variable es declararà dins un apartat afitat per les paraules **var ... fvar**.

En el cas d'un objecte constant, hem d'escriure el seu nom, el seu tipus i el valor constant:

```
const  
    nom: tipus = valor;  
fconst
```

En el cas que l'objecte sigui variable, la declaració pren la forma:

```
var  
    nom: tipus;  
fvar
```

En l'identificador *nom* cal que el primer caràcter sigui sempre un caràcter alfabètic. La resta de caràcters que segueixen poden ser alfabètics, numèrics o el caràcter “\_”. No hi pot haver espais entre caràcters.

Per exemple, el nom *1x1* és incorrecte, el nom *Josep-Maria* també és incorrecte; mentre que *x11* és correcte, *Josep\_Maria* és correcte.

En el cas de variables del mateix tipus, es poden declarar tal i com es veu a continuació:

```
var  
    nom1, nom2, nom3, ...: tipus;  
fvar
```

Com podeu comprovar, hi ha una sèrie de paraules que delimiten les parts de l'algorisme. Aquestes paraules les anomenem **paraules clau** i ens ajuden a identificar fàcilment amb què es correspon cada cosa. Aquestes paraules clau són paraules reservades del llenguatge algorítmic; i no les podem fer servir com a nom dels objectes (constants o variables). Per exemple, no hauríem de posar *var* com a nom d'una constant o variable.

No obstant això, hi ha una excepció a aquesta regla. Tot i que hem definit l'operador lògic *i*, i per tant, com a operador del llenguatge és una paraula reservada, s'accepta la possibilitat de definir una variable amb el nom *i*, que habitualment s'utilitza com a variable entera, especialment com a índex d'una taula, com veureu en altres mòduls.

Exemples

**const**

```
real = 1.4142135623730950;  
ln2: real = 0.6931471805599453;  
ptsEURO: real = 166.386;  
diesSetmana: enter = 7;
```

**fconst**

**var**

```
separador: caracter = '-';  
saldo, preu, nombreArticles: enter;  
diametreCercle: real;  
caracterLlegit: caracter;
```

**fvar**

## Expressions

Quan escrivim algorismes tindrem la necessitat d'operar entre els diversos objectes que hàgim declarat pel nostre entorn de treball per a obtenir algun valor concret. Això ho farem mitjançant les expressions.

Una expressió és qualsevol combinació d'operadors i operands.

Un operand pot ser una variable, una constant o una expressió. Els operadors poden ser unaris o binaris. Un operador unari és aquell que només opera amb un sol operand. És, per exemple, el cas de la negació lògica i el signe menys aplicat a un número. Un operador binari és aquell que opera amb dos operands.

Per exemple, la suma o la resta.

La definició d'expressió anterior no és suficient. Perquè una expressió ens sigui útil, cal que aquesta segueixi unes regles sintàctiques i que es respecti el significat dels símbols utilitzats. En altres paraules, cal que una expressió sigui correcta en sintaxi i semàntica.

## Sintaxis

Una expressió correcta i, per tant, avaluable, ha de seguir les normes de combinació següents:

- Un valor és una expressió.
- Una variable és una expressió.
- Una constant és una expressió.
- Si  $E$  és una expressió,  $(E)$  també ho és.
- Si  $E$  és una expressió i  $\bullet$  és un operador unari,  $\bullet E$  també ho és.
- Si  $E1$  i  $E2$  són expressions i  $\bullet$  és un operador binari,  $E1 \bullet E2$  també ho és.
- Si  $E1, E2, \dots, En$  són expressions i  $f$  és una funció,  $f(E1, E2, \dots, En)$  també ho és.

Les expressions s'escriuen linealitzades, és a dir, d'esquerra a dreta i de dalt cap a baix.

Per exemple, per a  $a, b, c, d, e$  enters,

$$\frac{\frac{a}{b} \times e}{\frac{c}{d}} \text{ equival a } (a \text{ div } b) \text{ div } (c \text{ div } d) * e$$

## Semàntica

Encara que una expressió tingui una sintaxi correcta, aquesta pot no tenir sentit, ja que cal respectar la tipificació de variables, constants, operadors i funcions. Parlarem, doncs, de **correctesa sintàctica** i de **correctesa semàntica**.

Per exemple, si trobem un operador suma aplicat a variables booleanes, l'expressió pot ser correcta sintàcticament, però serà incorrecta semànticament, ja que la suma no està definida

per als booleans. Un altre exemple és  $4.0 + 7$ , la qual no és correcta semànticament perquè barreja dos tipus de dades diferents.

Fixeu-vos que la darrera expressió pot tenir sentit fora del context algorísmic, però no ho és dins l'algorísmica.

## Avaluació

L'avaluació d'una expressió es fa d'esquerra a dreta començant per les funcions (que ja veurem més endavant) i allò que estigui entre els parèntesis més interns.

Dins els parèntesis, s'avaluaran primer les operacions més prioritàries i després les de menys prioritat.

Una constant s'avaluarà en el valor descrit en la seva definició. Una variable s'avaluarà en el valor que guarda en el moment en què s'avalua.

Les prioritats dels operadors (de major a menor prioritat) són les següents:

1.  $-$  (canvi de signe), **no**
2.  $*$ ,  $/$ , **div**, **mod**
3.  $+$ ,  $-$  (resta)
4.  $=$ ,  $\neq$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$
- i
5. **o**

A igualtat de prioritats, s'avalua d'esquerra a dreta.

### Exemples d'avaluació d'expressions:

1)  $c \text{ div } d * e$ . Primer s'avalua  $c \text{ div } d$  i el resultat d'aquesta expressió es multiplica

per  $e$ . L'expressió és equivalent a  $(c \text{ div } d) * e$ , i no té res a veure amb  $c \text{ div } (d * e)$

2)  $3 * 1 \text{ div } 3 = 1$  perquè és equivalent a  $(3 * 1) \text{ div } 3 = 1$

3)  $3 * 1 \text{ div } 3 = 0$

4)  $(a * b) - ((c + d) \text{ div } a) \text{ mod } (2 * a)$

L'ordre d'avaluació és el següent:

1.  $a * b$
2.  $c + d$
3.  $(c + d) \text{ div } a$
4.  $2 * a$
5. El valor del pas 3 mod el valor del pas 4
6. El valor del pas 1 menys el valor del pas 5

5)  $a + b = a * d \text{ div } c$

1. S'avalua  $a * d$
2. S'avalua el resultat del pas 1 **div** c
3. S'avalua  $a + b$
4. S'avalua = amb el resultat del pas 2 i el resultat del pas 3. El resultat d'avaluar aquesta expressió és un booleà.

6)  $a > b = c$

L'ordre d'avaluació d'aquesta expressió és:

1. S'avalua  $a > b$
2. S'avalua = amb el resultat del pas 1 i c. El resultat d'avaluar aquesta expressió és un booleà.

7)  $a * b > c + d \text{ o } e < f$

L'ordre d'avaluació d'aquesta expressió és:

1. S'avalua  $a * b$
2. S'avalua  $c + d$
3. S'avalua  $>$  amb el resultat del pas 1 i el resultat del pas 2
4. S'avalua el resultat  $e < f$
5. S'avalua **o** amb el resultat del pas 3 i el resultat del pas 4. El resultat d'avaluar aquesta expressió és un booleà.

8) **no**  $a = b \wedge c \geq d$

1. S'avalua **no**  $a$
2. S'avalua  $=$  amb el resultat del pas 1 i  $b$
3. S'avalua el resultat  $c \geq d$
4. S'avalua  $\wedge$  amb el resultat del pas 2 i el resultat del pas 3. El resultat d'avaluar aquesta expressió és un booleà.

### Definició de tipus. Tipus enumeratius

La notació algorítmica ens permet de definir tipus nous per a poder treballar amb objectes que siguin més propers al problema que estem tractant.

Un **constructor de tipus** és una construcció de la notació algorítmica que permet de definir tipus nous.

En aquest apartat, veurem el **constructor de tipus per enumeració**, que permet de definir un tipus nou de manera que els valors possibles que pugui tenir s'enumeren en la

La sintaxi és:

```
tipus  
    nom = { valor1, valor2, ..., valorn };  
ftipus
```

Els únics operadors que es poden aplicar als tipus enumeratius són els operadors relacionals externs. L'ordre en què s'enumeren els valors, és el que es fa servir per a establir les relacions de és petit a més gran.

#### tipus

```
color = { verd, blau, vermell, groc, magenta, cian, negre, blanc };  
dia = { dilluns, dimarts, dimecres, dijous, divendres, dissabte, diumenge };  
mes = { gener, febrer, març, abril, maig, juny, juliol, agost, setembre, octubre, novembre, desembre };
```

#### ftipus

L'expressió `febrer < març` ens tornarà el valor **cert**.



## Funcions de conversió de tipus

A vegades, ens caldrà passar d'un tipus de valor a un altre tipus. Per a fer-ho, disposarem d'unes funcions predefinides que ens permetran de convertir un valor d'un cert tipus elemental a un altre. Les funcions són les següents:

- *realAEnter*: aquesta funció accepta un argument de tipus real i el converteix al tipus enter. Cal tenir en compte que el nou valor resultant de la conversió perd tots els decimals que pugui tenir el valor real. Per exemple, la funció

*realAEnter*(4.5768) ens torna el valor de tipus enter 4, i *realAEnter*(-3.99) ens retorna el valor -3.

- *enterAREal*: aquesta funció accepta un argument de tipus enter i el converteix al tipus real. Per exemple, *enterAREal*(6) ens torna el valor de tipus real 6.0. Recordeu que, des del punt de vista algorísmic, 6.0 no té res a veure amb 6.

- *caracterACodi*: aquesta funció accepta un argument de tipus caràcter i el converteix al tipus enter. L'enter té a veure amb la codificació que el caràcter té internament en el computador. La funció la farem servir en casos molt especials i poc freqüents.

Per exemple, *caracterACodi*('A') ens tornarà

**tipus**  
*nom* = { *valor1*, *valor2*, ..., *valorn* };  
**ftipus**

el valor 65 si utilitzem la codificació ASCII, o 193 si utilitzem la codificació EBCDIC. Amb altres

codis, tindrà altres valors.

- *codiACaracter*: aquesta funció accepta un argument del tipus enter i el converteix al tipus caràcter. No obstant això, l'efecte de la funció està limitat a un subconjunt del tipus enter que té a veure amb la codificació de caràcters que utilitza el computador. Així, doncs, la funció no té un efecte definit per a aquells enters que no corresponguin a un codi de caràcter. Per exemple, en la codificació ASCII utilitzada en els PC, el subconjunt d'enters és l'interval entre 0 i 127. Per exemple, *codiACaracter*(10) ens retornarà un caràcter no visualitzable en la

codificació ASCII i que correspon al salt de línia en un text. En canvi, per la mateixa codificació, *codiACaracter(48)* ens tornarà el caràcter "0".

## Especificació d'algorismes

### Algorisme i canvi d'estat

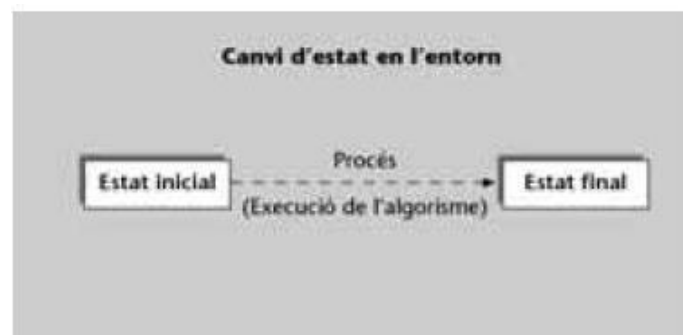
Un algorisme és una descripció d'un mètode sistemàtic per a resoldre un problema. La manera com un algorisme resol un problema forma part de l'algorisme mateix; però en tot cas, la resolució del problema s'aconsegueix si s'actua sobre l'entorn en què està immers l'algorisme, i es modifica convenientment.

Així, per exemple, una rentadora disposa d'un mètode sistemàtic per a rentar roba. Per tant, podem concloure que les rentadores disposen d'un algorisme per a rentar la roba, com hem vist en el mòdul anterior.

- Quin és en aquest cas l'entorn sobre el qual actua la rentadora? Doncs d'una manera simplificada podem dir que és la roba que hi ha dins la cubeta de la rentadora.
- I com hi actua? Bé, el procés que fa servir la rentadora el teniu descrit en el mòdul anterior. Però el que realment ens importa és que inicialment la roba està (més o menys) bruta; i després que la rentadora apliqui l'algorisme de rentat, la roba està neta.

És a dir, hi ha hagut un **canvi d'estat en l'entorn** sobre el qual actua la rentadora.

L'**estat inicial** era que la roba estava bruta i l'**estat final** és que la roba està neta. I, per tant, hem solucionat el nostre problema, que era rentar la roba.



Aquest canvi d'estat entre l'estat inicial i l'estat final no s'ha de produir de cop necessàriament. Hem vist en el mòdul anterior que l'algorisme que fa servir la rentadora per a rentar roba és l'execució d'un seguit d'accions. Cadascuna d'aquestes accions modifica l'estat d'alguna manera determinada. És a dir, entre l'estat inicial i l'estat final hi pot haver molts **estats intermedis**.

Així, la rentadora en primer lloc omple la seva cubeta d'aigua barrejada amb sabó. Per tant, la roba passa d'estar bruta a estar bruta i en remull amb aigua i sabó. Posteriorment, la cubeta es mou durant vint minuts. Això fa que la brutícia que hi ha a la roba, com que està en remull en aigua i sabó, passi, en gran part, a estar a l'aigua. Després d'aquests vint minuts en què la roba està en remull en aigua i sabó, la roba està més neta i l'aigua està més bruta.

Podríem continuar analitzant la resta d'accions que fa la rentadora per a rentar la roba. Totes aquestes accions produirien canvis d'estat sobre l'entorn que, si ens baséssim en l'estat immediatament anterior, ens anirien acostant cap a l'estat final desitjat.



Traduïm ara tot això al món del llenguatge algorísmic, que és on treballarem en aquesta assignatura.

Com ja es diu en el mòdul anterior, l'**entorn** d'un algorisme és el conjunt d'objectes que intervenen en aquest. En l'apartat anterior heu vist amb quins objectes treballa el llenguatge algorísmic.

De tots aquests, els únics objectes que l'execució d'un algorisme podrà modificar són les **variables**. Així, doncs, un procés (l'execució d'un algorisme) únicament pot modificar l'entorn si

Per això, diem que l'estat en un moment determinat de l'execució d'un algorisme ens ve determinat pel valor de les variables que hi intervenen.

canvia el valor de les variables que hi ha en aquest.

### Què vol dir especificar?

- El primer pas en la construcció d'un algorisme consisteix a saber clarament què és el que volem resoldre; quin és el problema que tenim entre mans.

D'aquesta manera, podrem intentar dissenyar un algorisme que el resolgui.

Això ho fem mitjançant l'especificació.

*Especificar* consisteix, doncs, a donar informació necessària i suficient per a definir el problema a resoldre de la manera més clara, concisa i no ambigua possible.

Heu de distingir clarament entre *especificació* i *algorisme*:

- L'**especificació** ens diu quin és el problema que hem de resoldre.
- L'**algorisme** ens diu com resolem el problema.

Tal com hem vist en l'apartat anterior, donat un problema que volem resoldre i un algorisme que el resol, aquest modifica l'entorn convenientment de manera que obtinguem el resultat desitjat.

Així, doncs, especificarem si donem una descripció la més clara, concisa i no ambigua possible de:

- L'estat inicial en què es troba l'entorn i del qual partim.
- L'estat final de l'entorn al qual hem d'arribar per a resoldre el problema.

**La descripció de l'estat inicial l'anomenarem precondició. I la descripció de l'estat final l'anomenarem postcondició.**

Fixeu-vos que veiem l'algorisme com una caixa negra que, a partir d'un estat inicial determinat per la precognició, obté l'estat final descrit a la post condició.

Quan especifiquem un problema, ens importen únicament els estats inicial i final, i no tot el conjunt d'estats intermedis pels quals es passa fins a arribar a l'estat final. És més, atesa una especificació d'un problema, hi pot haver més d'un algorisme que solucioni el problema (de la mateixa manera que per a anar d'un lloc a un altre ho podem fer per diversos camins).

Heu de tenir ben clar què forma part de *què* hem de resoldre i què forma part del *com* ho resolem. A vegades no és tan fàcil decidir què ha de contenir l'especificació d'un algorisme. En principi, atès un problema que cal resoldre, podem seguir les indicacions següents:

- En la precondició descriurem tot allò que tingui a veure amb les condicions en les quals el nostre algorisme ha de poder resoldre el problema.
- En la postcondició posarem allò que volem obtenir, el resultat que ha d'obtenir el nostre algorisme per a resoldre el problema. Pràcticament sempre haurem d'expressar la postcondició en termes de què hi ha a la precondició; és a dir, haurem de relacionar l'estat final amb l'estat inicial.

Si continuem amb l'exemple de la rentadora, si posem la rentadora en marxa i la roba està apilada al costat de la rentadora, l'algorisme s'executarà igualment però no resoldrà el

problema, ja que quan la rentadora hagi acabat, la roba continuarà apilada i igual de bruta al costat de la rentadora (a més, la rentadora potser s'haurà espatllat, però això és un altre tema).

Així, doncs, hi ha certes condicions que l'estat inicial ha de complir perquè l'algorisme que fa servir la rentadora per a rentar roba resolgui el problema.

Podríem tenir, per exemple, la precondició següent:

*{ Pre: La roba és dins la rentadora i la rentadora està connectada al subministrament de llum i d'aigua. A més, el caixonet destinat al sabó és ple de sabó }*

Si, a més, volem que la roba surti suau, s'hi hauria d'afegir que "el caixonet destinat al suavitzant és ple de suavitzant".

Un cop es compleix tot això, podem posar en marxa la rentadora. Aquesta executarà l'algorisme que donarà com a resultat que la roba quedi neta (després d'un cert temps, és clar!). Si alguna de les condicions de la precondició no es compleix, podem posar en marxa la rentadora (o almenys intentar-ho), però no tenim cap garantia que la roba es renti.

En la descripció de l'estat final, haurem de dir que la roba que inicialment teníem bruta a la cubeta ara està neta. Fixeu-vos que no n'hi ha prou de dir que la roba que hi ha dins la rentadora està neta. Hem de dir que "la roba és la mateixa que hi havia a l'estat inicial". En cas contrari, fixeu-vos en l'algorisme següent:

*"Obrir la porta de la rentadora. Agafar la roba bruta i tirar-la per la finestra. Anar a l'armari, agafar roba neta. Si no en queda, anar a la botiga i comprar-ne (en podríem comprar per a diverses vegades i desar una part a l'armari). Posar aquesta roba dins la rentadora. Tancar la porta de la rentadora."*

Aquest algorisme resol el problema en què passem de tenir roba bruta dins la rentadora a tenir-hi roba neta. Però no volem pas això. Volem rentar la roba bruta que tenim inicialment.

Hem d'anar sempre amb cura de relacionar la descripció que donem de l'estat final amb la descripció de l'estat inicial.

Podríem donar una postcondició com aquesta:

*{ Post: La roba que inicialment era dins la rentadora, continua a dins, però ara és neta. La roba no està estripada }*

---

Si, a més, especifiquéssim l'algorisme d'una rentadora assecadora (i, per tant, estaríem parlant d'un altre problema que cal resoldre), hi hauríem d'afegir: "i la roba està seca".

El comentari "la roba no està estripada" és important, ja que si no hi fos, una rentadora que estripés la roba (a més de netejar-la) seria una rentadora vàlida per a l'especificació (però evidentment no ho seria per a rentar roba en la vida real, que és el que estem intentant d'especificar!).

### Elements de l'especificació

L'especificació d'un algorisme consta de quatre parts: la declaració de variables, la preconditionió, el nom de l'algorisme i la postcondició.

La declaració de variables defineix el conjunt de variables que formen part de l'entorn de l'algorisme i ens diu de quin tipus són; la preconditionió i la postcondició són la descripció de l'estat inicial i final d'aquest entorn; i el nom de l'algorisme es correspon amb el nom que li donem a l'algorisme que estem especificant i hauria de ser el més significatiu possible.

D'aquesta manera, amb l'especificació diem el següent:

- 1) Donades unes variables amb les quals hem de ser capaços de representar un problema i la seva solució
- 2) i donades les condicions de l'estat inicial (indicades en la preconditionió)
- 3) hem de dissenyar un algorisme tal que,
- 4) ha de ser capaç d'obtenir l'estat final desitjat, descrit en la postcondició.

Ara ja tenim tots els ingredients necessaris per a poder especificar algorismes.

Així, doncs, vegem-ne un exemple: l'especificació d'un algorisme que calcula el factorial d'un nombre enter.

En primer lloc hem de decidir com representem el problema i la seva solució.

En aquest cas, farem servir una variable entera per a representar el nombre del qual volem calcular el factorial i una altra (també entera) per a guardar-hi el factorial d'aquest nombre. Les anomenarem respectivament *n* i *fact*.

En la precondició haurem d'explicar quines són les condicions en què l'algorisme es podrà executar i ens donarà un resultat correcte. En aquest cas hauríem de dir que el nombre a partir del qual calculem el factorial (és a dir, *n*) és més gran o igual que 0. És a dir, en la precondició definim sobre quin domini el nostre algorisme ha de ser capaç de resoldre el problema.

En la postcondició haurem de dir quines condicions compleix l'estat final; és a dir, aquell en què ja hem solucionat el problema. En aquest cas, la condició.

Vegem-ho, doncs, en el format amb el qual expressarem a partir d'ara les especificacions:

*n, fact: enter*

*{ Pre: n té com a valor inicial N i  $N \geq 0$  }*  
*factorial*  
*{ Post: fact és el factorial de N }*

En l'especificació anterior fem servir *N* per a designar el valor inicial de la variable *n*. Aquest "truc" és necessari perquè el valor de la variable *n* pot canviar mentre s'executa l'algorisme (ningú no ens garanteix que l'algorisme factorial no modifiqui *n*). Per aquest motiu, per a les variables per a les quals en la postcondició hem de saber quin és el seu valor a l'estat inicial farem servir aquest

"truc". Per convenció, sempre farem servir com a nom del valor inicial d'una variable el nom de la mateixa variable en majúscules.

En la postcondició diem simplement que la variable *fact* té com a valor el factorial de *N*. No cal dir que el factorial d'un nombre és el producte dels nombres naturals des d'1 fins a aquest nombre, ja que se suposa que nosaltres ja ho sabem això (i, per tant, no cal que ens ho diguin perquè puguem resoldre el problema).

I si ho diguéssim no estaria pas malament; simplement estaríem donant més informació de



la necessària.

Per a cada especificació, hi ha un conjunt d'algorismes que la compleixen o s'adapten. És a dir, que donat un problema, normalment el podrem resoldre de diverses maneres. Totes aquestes seran correctes i, per tant, assoliran almenys aquest objectiu dels quatre marcats en el document d'introducció de l'assignatura.

Però no totes assoliran els altres tres objectius: intel·ligible, eficient, general.

Per tant, quan dissenyeu un algorisme, haureu de ser capaços de dissenyar un algorisme correcte però alhora que assoleixi els altres tres objectius en la mesura que sigui possible.

### **Especificació i comentaris**

Fins ara hem parlat de l'especificació com a pas previ (i necessari) al disseny d'un algorisme. L'especificació també ens pot servir, un cop fet l'algorisme, per a recordar què fa clarament el nostre algorisme sense haver de repassar tot el codi. Això pren importància a mesura que la complexitat dels algorismes que dissenyem creix. Deduir què fa un algorisme a partir de l'algorisme mateix

pot no ser ràpid i ens podem equivocar fàcilment.

A banda de tot això, també és molt recomanable afegir comentaris dins l'algorisme que ens permetin d'entendre'l fàcilment. Aquests comentaris poden:

- 1) Especificar en quina situació o estat ens trobem en un moment determinat de l'algorisme. Això ens permetrà de fer el seguiment dels estats pels quals vapassant un algorisme d'una manera còmoda.
- 2) Limitar-se a ser comentaris purament aclaridors que fan referència a les construccions algorísmiques que hem fet servir per a resoldre un problema concret, de manera que posteriorment puguem entendre ràpidament com ho fa l'algorisme per a resoldre un problema. Aquests comentaris no tenen res a veure amb l'especificació, però sovint són força útils i és molt recomanable de fer-ne servir.

Posarem aquesta mena de comentaris entre claus, igual que la preconditionió i la postcondició. De totes maneres, per convenció, afegirem 'Pre:' i 'Post:' al principi de la preconditionió i la postcondició per a distingir-les, tal com ja hem fet anteriorment. Heu de ser capaços de distingir clarament entre l'especificació i els comentaris purament aclaridors.

Els comentaris, tot i que no formen part de l'algorisme pròpiament, són força importants per a fer més comprensibles els algorismes. D'aquesta manera, poden ajudar força en la fase de manteniment, on nosaltres mateixos o bé altres persones hàgim d'entendre un algorisme desenvolupat fatemps.

### Exemples d'especificació

En aquest subapartat podeu trobar exemples d'especificació de problemes amb una complexitat creixent.

### Intercanvi de dues variables

Volem especificar un algorisme que intercanviï els valors de dues variables. Per això, el nostre entorn estarà constituït per aquestes dues variables i hem d'expressar per a les dues variables que el valor final d'una variable és el valor inicial de l'altra variable. L'especificació és com segueix:

```
x, y: enter
{ Pre:  $x=X$  i  $y=Y$  }
intercanviar
{ Post:  $x=Y$  i  $y=X$  }
```

### Arrel quadrada

```
x, arrel: real
{ Pre:  $x=X$  i  $X \geq 0$  }
arrelQuadrada
{ Post:  $arrel^2 = X$  i  $arrel \geq 0$  }
```

Observeu que en la precondició indiquem que no té cap sentit fer l'arrel quadrada d'un nombre negatiu. D'altra banda, un nombre positiu té una arrel negativa i una de positiva. En la postcondició establim que el resultat correspongui a la positiva.

### Arrel quadrada entera

```
x, arrel: enter
{ Pre:  $x=X$  i  $X \geq 0$  }
arrelQuadradaEntera
{ Post:  $arrel^2 \leq X < (arrel + 1)^2$  }
```

Fixeu-vos que en la postcondició no cal especificar que  $arrel \geq 0$ , ja que la postcondició actual ja ho contempla implícitament.

### Quocient i residu

$x, y, q, r$ : **enter**  
{ Pre:  $x = X \wedge y = Y \wedge X, Y > 0$  }  
divisioEntera  
{ Post:  $X = q * Y + r$  i es compleix que  $0 \leq r < Y$  }

### Nombre de divisors

Volem especificar un algorisme que calculi el nombre de divisors positius d'un nombre positiu. En aquest cas, l'entorn estarà constituït per dues variables enteres.

Una, on tindrem el nombre, i l'altra, on a l'estat final hi haurà d'haver el nombre de divisors d'aquell.

$x, nDivisors$ : **enter**  
{ Pre:  $x = X \wedge X > 0$  }  
nombreDivisors  
{ Post:  $nDivisors$  és el nombre de nombres entre 1 i  $X$  que divideixen  $X$  }

Fixeu-vos que com que ens demanen el nombre de divisors sense especificar cap restricció, tenim en compte també l'1 i  $X$  que segur que són divisors de  $X$ .

### Estructures algorísmiques

En aquest apartat presentem les construccions bàsiques del llenguatge algorísmic que farem servir al llarg de tota l'assignatura per a expressar els algorismes.

### Estructura general d'un algorisme

Vegem, en primer lloc, quina estructura té un algorisme descrit en llenguatge algorísmic.

En tot algorisme apareixeran les parts que enumerem a continuació en l'ordre següent:

**1) Capçalera:**

Ens serveix per a identificar on comença la descripció del nostre algorisme i per a donar-li un nom.

**2) Definició de constants:**

Immediatament després de la capçalera definirem les constants que farem servir en el nostre algorisme. En cas que no en fem servir cap, aquesta part no hi apareixerà.

**3) Definició de tipus:**

En el primer apartat d'aquest mòdul heu vist que les variables tenen un tipus determinat i que el mateix llenguatge algorítmic ens en proporciona uns de bàsics. A vegades, però, el nostre algorisme requerirà tipus més complexos que els que ens proporciona el llenguatge algorítmic. Si això és així, haurem de donar la seva definició just després de la de les constants. En cas que no ens calgui definir nous tipus, no hi haurem de posar res en aquesta part.

**4) Declaració de variables:**

Aquí direm quines variables fa servir el nostre algorisme i de quin tipus són. La declaració de constants i tipus s'ha de fer abans perquè així aquí podrem fer servir els tipus i constants que hàgim definit.

**5) Cos de l'algorisme:**

Descripció en llenguatge algorítmic de les accions que fa l'algorisme.

**6) Final de l'algorisme:**

Ens serveix per a tenir clar on acaba l'algorisme.

A continuació, teniu un esquelet d'algorisme on apareixen les diferents parts d'aquest.

```
algorisme nom

  const
    nomConstant1: tipus = valorConstant1;
    nomConstant2: tipus = valorConstant2;
    ...
  fconst

  tipus
    nomTipus1 = definicióTipus1;
    nomTipus2 = definicióTipus2;
    ...
  ftipus

  var
    nomVariable1: tipusVariable1;
    nomVariable2: tipusVariable2;
    ...
  fvar

  acció1
  acció2
  ...

falgorisme
```

La paraula clau **algorisme** s'ha de posar sempre al principi d'un algorisme; i junt amb el nom de l'algorisme conforma la capçalera. La paraula clau **falgorisme** la posem un cop ja hem descrit totes les accions de l'algorisme.

Les constants sempre han d'aparèixer enmig de les paraules clau **const** i **fconst**. D'aquesta manera, sempre que veiem aquestes paraules sabrem que tot el que hi ha al mig són definicions de constants.

El mateix passa amb les definicions de tipus i les paraules clau **tipus** i **ftipus**, i també amb les declaracions de variables i les paraules clau **var** i **fvar**. Si alguna d'aquestes parts no té cap definició; és a dir, no hi ha cap constant o cap tipus o cap variable, tampoc no cal que posem les paraules clau corresponents.

Aquestes no són totes les paraules clau que farem servir; a continuació en trobareu més que

---

ens ajudaran, entre altres coses, a definir les estructures de control per a combinar accions senzilles amb accions més complexes.

### Accions elementals

El llenguatge algorísmic té únicament una acció elemental: l'acció d'assignació. Més endavant, en l'apartat següent, veurem com podem definir les nostres pròpies accions i estudiarem una sèrie d'accions predefinides (i no elementals) que ens ofereix el llenguatge per a poder comunicar l'execució de l'algorisme amb l'exterior. D'aquesta manera, podrem obtenir les dades necessàries per a l'execució de l'algorisme o bé mostrar les dades corresponents al resultat donat per l'algorisme.

### L'assignació

L'assignació és la manera que tenim en llenguatge algorísmic per a donar un valor a una variable.

L'expressem de la manera següent:

*nom de variable* := *expressió*;

*nom de variable* és l'identificador d'una variable i *expressió* és una expressió.

S'ha de complir sempre que la variable i l'expressió siguin del mateix tipus. No podem, per exemple, assignar valors booleans a variables enteres o valors enters a variables de tipus caràcter.

L'execució de l'assignació consisteix a avaluar l'expressió, cosa que dona com a resultat un valor; i posteriorment es dona a la variable aquest valor. És a dir, després de fer l'assignació, la variable té com a valor el resultat d'avaluar l'expressió.

Atès que l'assignació (i en general qualsevol acció) modifica l'entorn de l'algorisme, podem parlar d'un "estat previ" a l'execució de l'acció i també d'un "estat posterior". Per tant, podem aplicar també els conceptes de precondition i postcondition a una acció. Així podrem descriure d'una manera clara i sistemàtica l'efecte que té l'execució d'una acció sobre

l'entorn.

Vegem-ho per a l'assignació:

```
{ El resultat d'avaluar l'expressió  $E$  és  $V$  }  
 $x = E$ ;  
{  $x$  té com a valor  $V$  }
```

A partir d'ara abreujaem sempre la frase “ $x$  té com a valor  $V$ ” per un “ $x = V$ ” força més curt.

Exemples

En el cas que tinguem una variable anomenada  $a$ , que aquesta sigui de tipus caràcter i que li

```
{ }  
 $a := 'e'$ ;  
{  $a = 'e'$  }
```

vulguem assignar el valor “e”, podem fer:

Si tenim una altra variable de tipus enter (posem per cas  $d$ ), l'expressió haurà de ser de tipus enter.

```
{ }  
 $d := 3 * 4 + 2$ ;  
{  $d = 14$  }
```

També podem fer assignacions on en les expressions corresponents intervinguin variables; i, per tant, depenguin de l'estat de l'entorn.

Així, si  $x$  i  $y$  són dues variables reals, tenim que:

```
{  $(y + 3) / 2 = V$  }  
 $x := (y + 3.0) / 2.0$ ;  
{  $x = V$  }
```

on  $V$  és el valor de l'expressió, amb la mateixa idea dels valors inicials de les variables que posàvem en majúscules i que hem introduït en l'apartat anterior.

---

Aquesta és l'aplicació directa de l'especificació general de l'assignació que hem vist abans; però estareu d'acord que també ho podem expressar de la manera següent:

$$\begin{aligned} & \{ (y + 3) / 2 = V \} \\ & x := (y + 3.0) / 2.0; \\ & \{ x = V \} \end{aligned}$$

No podem fer servir aquesta forma per a l'especificació genèrica de l'assignació, ja que no sabem quines variables intervenen en l'expressió. Però sempre que hàgim d'especificar una assignació concreta ho farem així, ja que resulta més clar.

D'altra banda, el valor de la variable  $y$  no ha variat després de l'assignació.

Aquest fet no queda reflectit en la postcondició i és possible que ens interressi (segons l'algorisme concret del qual formi part l'assignació). Així, doncs, si ens interessés també aquest coneixement, l'hauríem de posar de la manera següent:

$$\begin{aligned} & \{ y = Y \} \\ & x := (y + 3.0) / 2.0; \\ & \{ x = (Y + 3) / 2 \} \end{aligned}$$

És important de tenir en compte que l'expressió  $E$  s'avalua abans de donar el valor a la variable

$x$ . Així, si la variable  $x$  apareix també dins l'expressió, el valor que haurem de fer servir per a calcular l'expressió serà el que tenia en l'estat previ a l'assignació.

Vegem-ho en l'exemple següent. Suposem que tenim una variable entera anomenada  $n$ :

$$\begin{aligned} & \{ y = Y \} \\ & x := (y + 3.0) / 2.0; \\ & \{ y = Y \wedge x = (Y + 3) / 2 \} \end{aligned}$$



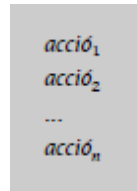
## Composició d'accions

Vegem ara com podem combinar accions elementals per tal de poder construir algorismes útils, ja que fins ara els algorismes que podem construir consisteixen en una única acció elemental (assignació).

## Composició seqüencial

La primera forma de composició i la més òbvia que veurem és la composició seqüencial d'accions. Consisteix a executar ordenadament una seqüència d'accions: primer la primera, després la segona, a continuació, la tercera; i així fins a la darrera.

Ho expressem de la manera següent:



acció<sub>1</sub>  
acció<sub>2</sub>  
...  
acció<sub>n</sub>

Fins ara havíem parlat d'accions individuals. Aquestes tenien un estat previ i un estat posterior (i, per tant, podíem descriure'ls amb el que hem anomenat *precondició* i *postcondició* de l'acció). Aquest estat posterior s'assolia després de l'execució d'una acció bàsica indivisible. És a dir, l'entorn canvia en un únic pas de l'estat inicial a l'estat final.

Una seqüència d'accions també té un estat previ i un estat posterior a la seva execució que podem descriure mitjançant la precondició i postcondició de la seqüència d'accions. Però ara la transformació de l'estat inicial en l'estat final no es produeix en un sol pas:

- L'estat previ a l'execució de la seqüència d'accions es correspon amb l'estat previ a l'execució de la primera acció de la seqüència.
- Un cop executada la primera acció, obtenim un nou estat que és l'estat posterior a l'execució de la primera acció. Aquest estat és a la vegada l'estat previ a l'execució de la segona acció de la seqüència.

- Un cop executada la segona acció, obtenim un nou estat que és l'estat posterior a l'execució de la segona acció i a la vegada l'estat previ de l'execució de la tercera acció.
- D'aquesta manera, i si fem el mateix per a la resta d'accions de la seqüència, arribem a l'estat previ de la darrera acció. Un cop executada aquesta acció, obtenim l'estat posterior a l'execució de la darrera acció de la seqüència. Aquest estat es correspon amb l'estat posterior a l'execució de la seqüència d'accions. Per a arribar a l'estat posterior a l'execució de la seqüència d'accions, passem per una sèrie d'estats intermedis. Donat el significat de cadascuna de les accions que formen part de la seqüència (determinat per la seva precondició i postcondició) i donat també l'estat previ a l'execució de la seqüència, podem determinar cadascun d'aquests estats intermedis i també l'estat posterior a l'execució de la seqüència d'accions.

Com que no tenim coneixement de les accions que formaran part de la seqüència d'accions, no podem dir gaire cosa respecte a la seva especificació.

Únicament el que ja hem dit respecte als estats intermedis.

Ho representem així:

```
{ P }  
acció1  
{ R1 }  
acció2  
{ R2 }  
...  
{ Rn-1 }  
acción  
{ Q }
```

$P$  és la descripció de l'estat previ a l'execució de la seqüència d'accions (precondició de la seqüència d'accions);  $R_i$  és la descripció de l'estat posterior a l'execució de  $acció_i$ ; i  $Q$  és la descripció de l'estat posterior a l'execució de la successió d'accions (postcondició de la successió d'accions).

La forma com siguin  $P$ ,  $Q$  i  $R_i$  dependrà per complet de quines accions formin part de la

successió. L'important és que cadascuna d'aquestes accions ens determinarà la relació entre  $R_{i-1}$  i  $R_i$ , i això ens determinarà la relació entre  $P$  i  $Q$ .

Si reviseu el mòdul "Introducció a la programació", podeu comprovar que l'algorisme de la rentadora consisteix en una composició seqüencial d'accions.

Així, doncs, ara ja podem fer els primers algorismes útils.

### Exemples

#### Intercanvi de valors

Volem donar una successió d'accions que intercanviï els valors de dues variables enteres  $x$  i  $y$ . Suposem que tenim una tercera variable també entera ( $aux$ ) que ens servirà per a fer

```
{ Pre:  $x = X$  i  $y = Y$  }  
 $aux := x$ ;  
{  $x = X$  i  $y = Y$  i  $aux = X$  }  
 $x := y$ ;  
{  $x = Y$  i  $y = Y$  i  $aux = X$  }  
 $y := aux$ ;  
{  $x = Y$  i  $y = X$  i  $aux = X$  }  
{ Post:  $x = Y$  i  $y = X$  }
```

l'intercanvi.

Com que és la primera seqüència d'accions que veiem i perquè tingueu clar quina part correspon a especificació i quina correspon pròpiament a l'algorisme, a continuació veurem la mateixa seqüència d'accions, però ara sense la part corresponent a l'especificació/descripció d'estats:

```
 $aux := x$ ;  
 $x := y$ ;  
 $y := aux$ ;
```

Així, podríem usar aquesta seqüència d'accions per a donar un algorisme (hi faltaria la capçalera, declaració de variables, etc.) que solucioni el problema especificat a 2.5.1. Com podeu comprovar, la descripció de l'estat inicial d'aquesta seqüència d'accions coincideix amb la precondició d'aquell problema. I la descripció de l'estat final, tot i que no coincideix (té més

coses, concretament  $aux=X$ ), en satisfà la postcondició. Observeu que la variable *aux* ens fa falta per a resoldre el problema, però no pren part en l'especificació del problema.

És el que es coneix com *variable temporal*, o també, *variable auxiliar*.

### Impostos i salari net

Donat el salari brut d'una persona, cal proporcionar una seqüència d'accions que calculi el salari net i els impostos que ha de pagar, sabent que aquesta persona paga un 20% d'impostos. Suposem que disposem de tres variables reals anomenades: *brut*, *net* i *impostos*.

```
{ Pre: brut = BRUT }  
net := brut * 0.8;  
{ brut = BRUT i net = BRUT * 0.8 }  
impostos := brut * 0.2;  
{ Post: brut = BRUT i net = BRUT * 0.8 i impostos = BRUT * 0.2 }
```

Hem resolt aquest problema amb una successió de dues accions: primer calculem el salari net i després els impostos. En la postcondició, tenim que el salari net és el 80% del brut i els impostos, el 20%, tal com estableix l'enunciat.

A l'algorisme hi hem incorporat la precondition i postcondició, i la descripció dels estats intermedis (en aquest cas només un) perquè veieu com les accions de la seqüència determinen aquests estats intermedis fins que arribem a l'estat final. Aquestes descripcions d'estats, sempre entre claus, no formen part de l'algorisme, les incorporem en forma de comentari.

### Interessos

Donat un capital inicial invertit en un dipòsit financer que dóna un 5% anual, doneu una seqüència d'accions que calculi quin capital tindrem després de quatre anys (cada any els interessos obtinguts passen a engrossir el capital). Suposeu que tenim aquest capital inicial en una variable real anomenada *capital*; i que volem tenir el capital final també en aquesta variable.

```
{ Pre: capital = CAPITAL }  
capital := capital * 105.0/100.0;  
{ capital té el capital que tenim després d'un any, on el capital inicial és CAPITAL }  
capital := capital * 105.0/100.0;  
{ capital té el capital que tenim després de dos anys, on el capital inicial és CAPITAL }  
capital := capital * 105.0/100.0;  
{ capital té el capital que tenim després de tres anys, on el capital inicial és CAPITAL }  
capital := capital * 105.0/100.0;  
{ Post: capital té el capital que tenim després de quatre anys, on el capital inicial és CAPITAL }
```

Aconsegüim el nostre propòsit si repetim quatre cops la mateixa acció. Aquesta acció calcula el capital que tenim al final d'un any tenint en compte el capital que teníem al principi d'any. Si la repetim quatre cops, obtenim el capital que tindrem després de quatre anys.

### Composició alternativa

Amb el que hem vist fins ara podem construir accions més complexes a partir d'accions més simples. Però les accions que executem sempre seran les mateixes, independentment de quin sigui l'estat inicial. Amb una restricció com aquesta no podem, per exemple, calcular el màxim de dos valors.

La composició alternativa ens permet de decidir quines accions executarem segons quin sigui l'estat. A continuació en teniu la sintaxi:

```
si expressió llavors  
    accióa  
sino  
    acciób  
fsi
```

Aquí, *expressió* és una expressió de tipus booleà, i per això de vegades també s'anomena

*condició*. D'altra banda, *acció<sub>a</sub>* i *acció<sub>b</sub>* són dues accions (o successions d'accions, tant se val). La seva execució consisteix a avaluar l'expressió; si aquesta és certa, s'executarà *acció<sub>a</sub>*; i si és falsa s'executarà *acció<sub>b</sub>*. S'anomena *alternativa doble*. Les paraules **si**, **llavors**, **sino** i **fsi** són paraules clau del llenguatge algorísmic i ens serveixen per a delimitar fàcilment cadascuna de les parts de la composició alternativa.

Hi ha una versió més reduïda, on no apareix la part corresponent al **sinó**. En aquest cas,

---

s'avalua l'expressió; i si és certa s'executa *acció<sub>a</sub>* i si és falsa no es fa res. La seva sintaxi la teniu a continuació. S'anomena *alternativa simple*.

```
si expressió llavors
    accióa
fisi
```

Farem servir la composició alternativa quan vulguem executar una acció o una altra dependent de si es compleix una determinada condició o no, és a dir, quan vulguem variar el flux d'execució del nostre algoritme.

Vegem com s'expressa el significat de la composició alternativa en l'especificació:

```
[ P i el resultat d'avaluar expressió és un cert valor B (que pot ser cert o fals) ]
si expressió llavors
    [ P i B ]
    accióa
    [ Qa ]
sino
    [ P i no(B) ]
    acciób
    [ Qb ]
fisi
[ Es compleix Qa o bé es compleix Qb ]
```

Igual que passava a la composició seqüencial, *Q<sub>a</sub>*, *Q<sub>b</sub>* i *P* dependran de quin sigui l'entorn de l'algorisme i de quines siguin *acció<sub>a</sub>* i *acció<sub>b</sub>*. Després d'avaluar l'expressió booleana triem una branca o l'altra del **si**; però l'estat no es modifica.

L'estat únicament es veu modificat quan passem a executar *acció<sub>a</sub>* o *acció<sub>b</sub>*.

Quant a l'especificació de la versió sense **sino**, s'obté fàcilment a partir de l'especificació que acabem de veure eliminant la part del **sino** i substituint a la postcondició final *Q<sub>b</sub>* per *P*. Això és així perquè la composició alternativa sense **sino** seria equivalent a una amb **sino**, on *acció<sub>b</sub>* fos una acció que no fa res.

Exemple

### Màxim de dos nombres enters

Volem fer el cos d'un algorisme que calculi el màxim de dos nombres enters. Suposem que tindrem aquests dos nombres enters en dues variables enteres  $x$  i  $y$  i que volem deixar el resultat a una altra variable entera  $z$ .

Es tracta d'assignar a  $z$  el màxim de tots dos valors ( $x$  i  $y$ ). El problema és que quan fem l'algorisme no sabem quin serà el valor més gran:  $x$  o  $y$ .

- Si sabéssim que  $x$  és el més gran podríem fer directament  $z := x$ .
- Si, en canvi, sabéssim que el valor més gran és  $y$ , podríem fer  $z := y$ .
- Si tots dos valors són iguals, qualsevol de les dues assignacions ens serveix.

Llavors, segons si  $x$  és més gran que  $y$  o bé al revés, haurem d'actuar d'una manera o d'una altra. Per a poder donar un algorisme que resolgui el problema hem de poder triar des de l'algorisme mateix entre executar una acció o l'altra.

Podem aconseguir això amb la composició alternativa doble. Vegem com:

```
{ Pre:  $x = X$  i  $y = Y$  }  
si  $x > y$  llavors  
  {  $x = X$  i  $y = Y$  i  $x > y$  }  
   $z := x$ ;  
  {  $x = X$  i  $y = Y$  i  $X > Y$  i  $z = X$ ; per tant:  $z = \text{màxim}(X, Y)$  }  
sino  
  {  $x = X$  i  $y = Y$  i  $X \leq Y$  }  
   $z := y$ ;  
  {  $x = X$  i  $y = Y$  i  $X \leq Y$  i  $z = Y$ ; per tant:  $z = \text{màxim}(X, Y)$  }  
fsi  
{ Post:  $z = \text{màxim}(X, Y)$  }
```

Igual que en els exemples del subapartat anterior, hem afegit al tros d'algorisme una descripció de tots els estats intermedis pels quals pot passar l'execució de l'algorisme. Quan vosaltres dissenyeu els algorismes, no cal que ho feu, però, com a pas previ, cal que especifiqueu el problema. És a dir, cal que doneu una precondition i postcondició globals que l'algorisme haurà de complir.

El mateix algorisme, ara sense la descripció dels estats intermedis (però amb la precondition

i la postcondició), resultaria de la manera següent:

```
{ Pre:  $x = X$  i  $y = Y$  }  
si  $x > y$  llavors  $z := x$ ;  
sino  $z := y$ ;  
fsi  
{ Post:  $z = \text{màxim}(X, Y)$  }
```

Fixeu-vos que el més important en la composició alternativa és que tant d'una branca com de l'altra se'n pugui deduir la postcondició que volíem assolir; en aquest cas, que  $z$  tingués com a valor el màxim de  $x$  i  $y$ .

### Composició iterativa

Amb la composició alternativa ja podem resoldre un nombre més gran de problemes, però encara ens cal anar més enllà. Així, per exemple, imagineu que el llenguatge algorítmic o l'ordinador no saben multiplicar dos enters (en el supòsit que no disposessin de l'operador  $*$ ) i en canvi sí que els saben sumar.

Quan hàgim de multiplicar dos enters, li haurem d'explicar com ho ha de fer.

Així, per a aconseguir que un enter  $z$  sigui igual a  $10 * x$ , ho podem fer d'aquesta manera:

$z := x + x + x + x + x + x + x + x + x + x$

Si l'expressió que cal assignar a  $z$  fos  $25 * x$ :

$z := x + x$ ;



Una altra manera de fer-ho per al cas de  $z = 10 * x$  seria:

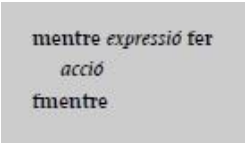
```
z := 0;  
z := z + x;  
z := z + x;  
z := z + x;  
z := z + x;  
z := z + x;  
z := z + x;  
z := z + x;  
z := z + x;  
z := z + x;  
z := z + x;
```

Aquesta manera de resoldre-ho és molt feixuga i, a més, només es resolen casos particulars del producte, no pas un cas general com  $x * y$ , on hem de ser capaços de determinar quants cops cal executar l'acció  $z := z + x$  des del mateix algorisme, segons el valor que tingui la variable  $y$  en aquell moment.

Podeu trobar un altre exemple en l'exercici "Interessos" sobre la composició seqüencial on, a partir d'un capital inicial, s'acumulen interessos durant quatre anys. Què passa si ho volem calcular per a vint anys? I per a un nombre indeterminat d'anys que guardem en una altra variable?

Per a resoldre aquest tipus de situacions ens servirà la composició iterativa.

L'expressarem de la manera següent:



```
mentre expressió fer  
    acció  
fmentre
```

Aquí, *expressió* és una expressió de tipus booleà i *acció* és una acció o successió d'accions. Les paraules **mentre**, **fer** i **fmentre** són paraules clau del llenguatge algorísmic.

L'execució de la construcció **mentre** consisteix a executar l'acció mentre l'avaluació de l'expressió tingui com a resultat **cert**. En el moment en què l'expressió passi a ser falsa, no es fa res més. Si inicialment *expressió* és falsa, no s'executa *acció* cap vegada. A cadascuna de les execucions de l'acció en direm **iteració**.

## Especificació

Per a especificar la composició iterativa es fa servir una propietat especial anomenada *invariant* (que representarem per  $I$ ).

L'invariant és una descripció de quina és l'evolució dels càlculs acumulats al llarg de totes les iteracions que hem fet fins a un moment determinat.

Aquesta descripció ens ha de servir per a descriure l'estat en què ens trobem quan encara no hem fet cap iteració, quan n'hem fet una, quan n'hem fet dues... i quan les hem fet totes.

La propietat invariant s'anomena d'aquesta manera per aquesta raó: una mateixa descripció ens serveix per a descriure l'estat abans i després de cada iteració.

Així, doncs, l'invariant s'ha de complir:

- 1) Just abans de la construcció iterativa; és a dir, en l'estat previ a la composició iterativa (encara no s'ha fet cap iteració).
- 2) En l'estat previ a l'execució de l'acció en totes les iteracions.
- 3) En l'estat posterior a l'execució de l'acció en totes les iteracions.
- 4) Un cop ja s'ha acabat l'execució de la composició iterativa; és a dir, en l'estat posterior d'aquesta (s'han fet ja totes les iteracions).

No us preocupeu gaire per l'invariant, més tard en comentarem alguna cosa més. Atès que l'invariant s'ha de complir abans i després de cada iteració i que, a més, l'avaluació de l'expressió booleana no pot mai modificar l'estat, l'especificació de la composició iterativa ens queda:

```
{ I i el resultat d'avaluar expressió és un cert valor booleà B  
  (que pot ser cert o fals) }  
mentre expressió fer  
  { I i B }  
  acció  
  { I }  
fmentre  
{ I i no(B) }
```

Un cop acabat el **mentre**, sabem que no es compleix la condició *B*. Això, juntament amb l'invariant, ens ha de permetre d'afirmar que hem arribat a l'estat que desitjàvem. Ho veurem a continuació amb un exemple.

### Exemple

#### Multiplicació de dos enters

Tornem a l'exemple motivador de l'inici d'aquest subapartat. Ara ja tenim les eines per a poder multiplicar dos enters fent servir únicament sumes i d'una manera general. Aquí resoldrem l'algorisme per al cas en què la *y* és positiva.

Fàcilment podem ampliar l'algorisme si fem servir l'operador de canvi de signe per a obtenir un algorisme que funcioni per a qualsevol *y*. Com en els exemples anteriors, suposarem que ja tenim els valors a les variables *x* i *y*, i que volem guardar el resultat en una variable ja definida anomenada *z*.

```
{ Pre: x = X i y = Y i Y ≥ 0 }  
z := 0;  
mentre y ≠ 0 fer  
  z := z + x;  
  y := y - 1;  
fmentre  
{ Post: z = X * Y }
```

Ens hem estalviat la descripció dels estats intermedis expressament per a poder formular la pregunta següent: quina és aquí la propietat invariant?

Bé, sabem que la propietat invariant s'ha de complir al principi i al final de cada iteració. Podríem triar per exemple  $x = X$ . Aquesta propietat es compleix abans i després de cada

---

iteració (de fet es compleix sempre, ja que no modifiquem el seu valor).

Però ens descriu la propietat  $x = X$  els càlculs fets? No, en absolut. No hi diem res respecte a la multiplicació que fem pas a pas. L'invariant ha de contenir prou informació perquè ens serveixi i per a deduir, a partir de  $I$  i  $\text{no}(B)$ , la postcondició a la qual volem arribar (en aquest cas  $z = X * Y$ ). Evidentment amb  $x = X$  no ho aconseguim.

Una propietat que sí ens descriu els càlculs fets és:  $x = X \wedge z = x * (Y - y)$ ; a més, es compleix abans i després de cada iteració. Podeu comprovar que aquesta sí que ens permet d'arribar a la postcondició esperada. Tornem a veure la part central de l'algorisme anterior però ara incorporem l'invariant i el valor de l'expressió  $B$  a la descripció dels estats intermedis:

```
{ Pre:  $x = X \wedge y = Y \wedge Y \geq 0$  }  
z := 0;  
{  $x = X \wedge z = x * (Y - y) \wedge y \geq 0$  }  
mentre  $y \neq 0$  fer  
  {  $x = X \wedge z = x * (Y - y) \wedge y > 0$  }  
  z := z + x;  
  y := y - 1;  
  {  $x = X \wedge z = x * (Y - y) \wedge y \geq 0$  }  
fmentre  
{ Post:  $x = X \wedge z = x * (Y - y) \wedge y = 0$ ; per tant:  $z = X * Y$  }
```

Aquesta descripció de l'invariant només pretén que entengueu tan bé com sigui possible l'especificació de la composició iterativa; què significa exactament "propietat invariant" i el perquè d'aquesta. Però en cap cas no haureu de trobar vosaltres mateixos els invariants de les composicions iteratives que dissenyeu.

De totes maneres, al llarg dels apunts us trobareu amb invariants molts cops com a complement de l'explicació dels algorismes que veurem.

## El factorial

Vegem un altre exemple en què farem servir la composició iterativa. Volem fer un tros d'algorisme que donat un nombre enter positiu (que suposarem a la variable  $n$ ) en calculi el factorial, i el deixi en una variable anomenada *fact*.

Hem vist l'especificació d'aquest problema en l'apartat anterior. Per a solucionar-lo, hem de multiplicar tots els nombres naturals entre 1 i el número del qual volem calcular el factorial.

Per a fer això, hem de fer servir la composició iterativa.

Necessitarem una variable auxiliar (que suposarem que tenim definida) per a saber quin és el nombre que multipliquem en cada moment. Vegem-ho:

```
{ Pre:  $n = N$  i  $N \geq 0$  }  
fact := 1;  
i := 1;  
{  $n = N$  i  $N \geq 0$  i fact és el factorial de  $i - 1$  i  $i \leq n + 1$  }  
mentre  $i \leq n$  fer  
  {  $n = N$  i  $N \geq 0$  i fact és el factorial de  $i - 1$  i  $i \leq n$  }  
  fact := fact * i;  
  i := i + 1;  
  {  $n = N$  i  $N \geq 0$  i fact és el factorial de  $i - 1$  }  
fmentre  
{ Post:  $n = N$ ,  $N \geq 0$ , fact és el factorial de  $i - 1$  i  $i = n + 1$ , per tant: fact és el factorial de  $N$  }
```

En aquest exemple hem tornat a descriure els estats intermedis mitjançant l'invariant. Això ho hem fet amb una intenció purament didàctica i ja no ho farem tan detalladament a partir d'ara. Fixeu-vos, però, que després de la composició iterativa, del fet que es compleix l'invariant i que la condició del **mentre** és falsa, en podem deduir la postcondició a la qual volíem arribar: que *fact* és el factorial de *N*.

### Finalització de la composició iterativa

Un punt força important quan fem servir la composició iterativa és saber del cert que s'acabarà en algun moment. Sempre que fem una iteració, correm el risc de fer-la malament i que no acabi mai. Així, si a l'exemple de la multiplicació de dos nombres ens oblidem l'acció  $y := y - 1$ , *y* valdrà sempre el mateix i, per tant, un cop entrem dins el mentre, l'acció  $z := z + x$  s'executarà indefinidament.

Intuïtivament, heu de ser capaços de veure que les modificacions fetes per una iteració sobre l'entorn ens porten necessàriament al fet que a la iteració següent ens queden menys iteracions per fer.

Aquest fet intuïtiu es pot demostrar formalment mitjançant la funció de fita.

La **funció de fita** és una funció matemàtica que té com a domini el conjunt de valors vàlids per al conjunt de variables que intervenen a la iteració i com a rang els nombres enters. És a dir, donat un estat, si li apliquem la funció de fita obtindrem un enter.

Aquesta funció ha de complir que el resultat d'aplicar-la abans d'una iteració ha de ser estrictament més gran que el resultat d'aplicar-la després de la iteració.

A més, si es compleix l'invariant i  $B$  és certa, ha de ser més gran que zero.

Si donada una composició iterativa som capaços de trobar una funció de fita que compleixi aquests requisits, haurem demostrat que la construcció iterativa acaba.

Així, en l'exemple del producte que hem vist anteriorment, una funció de fita vàlida (n'hi pot haver moltes) podria ser:  $fita = y + 1$ . En el cas del factorial, una funció de fita correcta és:  $fita = n - i + 1$ .

En aquesta assignatura no us demanarem mai que trobeu l'invariant ni la fita d'una construcció iterativa. Però sí és important que entengueu les explicacions anteriors i que sapigueu que tant el comportament com la finalització de l'execució d'una construcció iterativa es poden definir formalment mitjançant els conceptes d'*invariant* i *fita*.

### Variants de la composició iterativa

La construcció **mentre** ens permet expressar qualsevol composició d'accions on hàgim de repetir una sèrie de càlculs. De totes maneres, moltes vegades aquesta repetició d'accions segueix un patró concret on fem servir una variable que anem incrementant (o decrementant) a cada iteració fins que arriba a un valor donat. Per a aquests casos disposem d'una variant de la construcció iterativa que expressarem així:

```
per índex := valor inicial fins valor final [pas increment] fer  
    acció  
fper
```

On tenim que la part "**pas increment**" és opcional i si no hi apareix es considera que increment és igual a 1. Les paraules **per**, **fins**, **pas**, **fer** i **fper** són paraules clau del llenguatge algorísmic.

Aquesta construcció es pot expressar amb la sentència **mentre** que hem vist anteriorment. Així que vosaltres mateixos en podeu deduir el significat i l'especificació:

```
    índex := valor inicial;  
    mentre índex ≤ valor final fer  
        acció;  
        índex := índex + increment;  
fmentre
```

Fixeu-vos que si l'increment fos negatiu, caldria canviar la condició del bucle **mentre** a *índex* ≥ *valor final*.

Amb això, podríem reformular l'algorisme de la factorial de la manera següent:

```
{ Pre: n = N i N ≥ 0 }  
fact := 1;  
per i := 1 fins n fer  
    fact := fact * i;  
fper  
{ Post: fact és el factorial de N }
```

Únicament ens hem estalviat dues línies de codi (la inicialització de la *i* i l'increment d'aquesta). Però a vegades ens resultarà més còmode fer servir aquesta construcció; especialment per a treballar amb taules, element que s'introdueix en altres mòduls. En general, sempre que el nombre d'iteracions sigui conegut, farem servir la construcció **per**.

## Accions i funcions

A vegades ens trobarem amb problemes que serien més fàcils d'expressar i resoldre si disposéssim d'accions o funcions més convenients que la simple assignació (l'única acció que ens proporciona el llenguatge algorísmic) i les funcions de conversió de tipus.

De la mateixa manera que la notació algorísmica ens permet definir els nostres tipus (el tipus enumeratiu, per exemple), podem definir les nostres accions i funcions pròpies.

Imagineu, per exemple, que ens dediquem a resoldre problemes de caire geomètric i que sovint cal calcular el sinus d'un angle i el seu cosinus. Si cada cop que necessitem calcular el sinus d'un angle, hem de posar totes les accions necessàries per a poder fer els càlculs amb les variables corresponents, els algorismes serien llargs, gairebé repetitius, i amb massa detalls que caldria tenir en compte quan els llegim. En canvi, si per a expressar l'algorisme disposéssim d'unes funcions que calculessin els sinus i cosinus d'un angle donat, l'algorisme seria possiblement més curt, i sobretot més entenedor, ja que ens concentraríem més en el

---

problema de geometria en si, sense pensar en el problema del càlcul del sinus o del cosinus.

Com que la notació algorítmica no té les funcions de sinus i cosinus, però ens permet de definir funcions, podem suposar que aquestes existeixen, i més endavant dissenyar-les i preocupar-nos només una vegada de com es calcula un sinus i un cosinus a partir d'un angle donat. També podríem necessitar accions com girar punts a l'espai, rectes a l'espai, etc., que farien més comprensible un algorisme que resolgués un problema més general.

El fet que la notació algorítmica ens permeti de definir accions i/o funcions ens serà de gran ajuda, ja que ens permetrà d'enfrontar-nos amb problemes de més complexitat. Els algorismes que solucionen aquests problemes es formularan en termes d'accions més convenients per a la intel·ligibilitat dels algorismes en si, i del seu seguiment. Més endavant desenvoluparem les accions fins a arribar al detall que requereix la notació algorítmica.

Podem considerar la possibilitat de crear accions i funcions noves com un enriquiment o ampliació del repertori d'accions disponibles del llenguatge algorítmic per a enfrontar-nos a problemes més complexos amb més comoditat.

Tot i que es pot pensar que les accions i funcions ens serviran per a estalviar un grapat d'instruccions quan dissenyem un algorisme, la importància de les accions i funcions rau en el fet que són eines del llenguatge algorítmic que ens facilitarà la metodologia que cal seguir més endavant en el disseny de programes.

D'altra banda, quan desenvolupem els algorismes podem comptar amb algunes accions i funcions que es consideren predefinides, i que la majoria de llenguatges de programació proporcionen per a facilitar la tasca de la programació; especialment les accions referents a l'entrada i sortida de dades.

A vegades també podríem desenvolupar algorismes a partir d'una biblioteca d'accions i funcions ja desenvolupades per algun equip de desenvolupament de programari. Així, doncs, les accions i funcions permeten de dividir el desenvolupament d'un algorisme entre diverses persones si se segueix una metodologia i criteris adequats al fi que es persegueix.



## Accions

Intuïtivament, una acció ve a ser una mena de subalgorisme que es pot utilitzar des de qualsevol punt d'un altre algorisme o acció. De fet, les accions, tenen la mateixa estructura que els algorismes però la capçalera es delimita per les paraules clau **accio ... faccio** en comptes de **algorisme ... falgorisme**.

Per tant, dins l'acció podem definir també un entorn local o propi de l'acció (seccions **const ... fconst, tipus ... ftipus, var ... fvar** definits dins l'acció).

S'entén per *entorn local* aquell conjunt d'objectes que només l'acció que es desenvolupa pot utilitzar. És a dir, que altres parts de l'algorisme no podran fer servir aquest entorn.

Per exemple, imagineu que tenim l'enunciat següent:

Es desitja fer l'intercanvi entre els valors de les variables enteres  $x$  i  $y$ ,  $z$  i  $w$ , i  $v$  i  $u$ , respectivament.

És a dir, ens demana que (especificant el problema...):

$x, y, z, w, v, u$ : **enter**

$\{ \text{Pre: } (x = X \wedge y = Y) \wedge (z = Z \wedge w = W) \wedge (v = V \wedge u = U) \}$

tresIntercanvis

$\{ \text{Post: } (x = Y \wedge y = X) \wedge (z = W \wedge w = Z) \wedge (v = U \wedge u = V) \}$

El problema de l'intercanvi entre dues variables s'ha estudiat abans, i per tant, podríem escriure la solució següent:

```
{ Pre: (x = X i y = Y) i (z = Z i w = W) i (v = V i u = U) }  
    aux := x;  
    x := y;  
    y := aux;  
{ (x = Y i y = X) i (z = Z i w = W) i (v = V i u = U) }  
    aux := z;  
    z := w;  
    w := aux;  
{ (x = Y i y = X) i (z = W i w = Z) i (v = V i u = U) }  
    aux := v;  
    v := u;  
    u := aux;  
{ Post: (x = Y i y = X) i (z = W i w = Z) i (v = U i u = V) }
```

**En certes aplicacions,...**

... podem disposar d'una biblioteca de programes.

En aquests casos, les accions i/o funcions les han desenvolupat altres persones perquè nosaltres les puguem fer. L'algorisme resultant conté tres subalgorismes semblants que es diferencien només per algunes de les variables utilitzades. De fet, fem tres intercanvis de dues variables donades, repetim el mateix procés d'intercanvi però primer aplicat a les variables *x* i *y*, després a les variables *z* i *w*, i finalment a les variables *u* i *v*.

Hauria estat més fàcil poder-ho expressar de la manera següent:

```
{ Pre: (x = X i y = Y) i (z = Z i w = W) i (v = V i u = U) }  
intercanvia x i y  
{ (x = Y i y = X) i (z = Z i w = W) i (v = V i u = U) }  
intercanvia z i w  
{ (x = Y i y = X) i (z = W i w = Z) i (v = V i u = U) }  
intercanvia u i v  
{ Post: (x = Y i y = X) i (z = W i w = Z) i (v = U i u = V) }
```

I després definir en què consisteix "intercanvia ... i ...". Suposem que podem accedir des de l'acció a les variables de l'algorisme que l'invoca; aleshores, podríem dissenyar aquesta acció de la manera següent:

accio intercanvia

```
var
    aux: enter;
fvar
    aux := x;
    x := y;
    y := aux;
facció
```

Suposem també que les variables  $x$  i  $y$  estan declarades en l'algorisme que invoca l'acció; aleshores se'ns planteja un altre problema: com intercanviarem les variables  $z$  i  $w$ ,  $v$  i  $u$ ? Posant-les en  $x$  i  $y$  abans de la crida? Això seria massa complicat!

A més, per a dissenyar aquesta acció ens cal saber quina declaració de variables  $x$  i  $y$  ha fet l'algorisme que l'utilitza.

Hi ha, per tant, una dependència massa gran entre l'algorisme o acció que la cridi ( $x$  i  $y$ ) i l'acció intercanvia. Hauríem de dissenyar sempre conjuntament l'acció i qualsevol algorisme que la vulgui utilitzar. Volem definir una acció d'intercanvi més general que es pugui aplicar a dues variables qualssevol i que no depengui de l'entorn particular des d'on s'invoca.

En l'exemple en què desenvolupem *tresIntercanvis*, l'algorisme informal en què hem posat accions del tipus "intercanvia ... i ...", hem expressat què volíem fer i sobre quines variables afectava què es vol fer. El primer cop volem intercanviar  $x$  i  $y$ , el segon l'intercanvi es fa sobre  $z$  i  $w$ , etc. Per tant, volem poder dissenyar una acció general d'intercanvi de dos valors que es pugui aplicar de manera directa a qualsevol entorn que la necessiti.

Aquests problemes exposats se solucionen mitjançant l'ús de paràmetres.

## Paràmetres

Quan construïm i definim una acció cal poder expressar, dins el cos de l'acció, accions que actuen sobre objectes genèrics i, més tard, poder associar aquests objectes genèrics a objectes concrets definits en l'entorn des del qual es faci servir l'acció.

En l'exemple anterior ens aniria bé de poder indicar que  $x$  i  $y$  no són objectes reals sinó que s'utilitzen només per a formular com es fa l'intercanvi de dues variables, i deixem pendent per a més endavant què serà realment  $x$  i què serà realment  $y$ . Aquests objectes genèrics

---

(x, y en l'exemple) s'anomenen **paràmetres formals de l'acció**. Llavors diem que l'acció està parametritzada, és a dir que per a poder-la fer hem d'associar abans aquests paràmetres a objectes concrets.

D'altra banda, els paràmetres formals ens aniran bé per a expressar allò que volfem de manera genèrica, que després serà aplicable a l'entorn que convingui en cada cas concret.

A la capçalera de l'acció indicarem quins són els paràmetres formals amb els quals treballa. Per tant, la sintaxi de la capçalera d'una acció serà la següent:

```
accio nom(param1, param2, ... , paramn)  
    ... cos de l'acció  
faccio
```

*nom* és el nom que identifica l'acció, i *param*<sub>i</sub> són paràmetres de l'acció.

Si seguim el nostre exemple, l'acció que ens cal té l'aspecte següent:

```
accio intercanvia(entsor x: enter, entsor y: enter)  
var  
    aux: enter;  
fvar  
    aux := x;  
    x := y;  
    y := aux;  
faccio
```

Els paràmetres formals de l'acció són  $x$  i  $y$ . Malgrat que s'anomenin  $x$  i  $y$ , no tenen res a veure amb les variables  $x$  i  $y$  de l'algorisme. Es tracta d'una coincidència de noms que no tenen relació entre si. Fixeu-vos que estan definits com a paràmetres d'entrada/sortida; més endavant els definirem.

Un cop definida l'acció sobre uns paràmetres formals, només cal indicar a l'acció a l'hora de cridar-la des de l'algorisme els objectes reals sobre els quals cal que actuï.

Per tant, per a fer servir una acció ja definida dins un algorisme o dins una altra acció haurem d'escriure el seu nom seguit dels objectes de l'entorn sobre els quals volem que actuï l'acció. Llavors direm que invoquem o cridem una acció.

Per a invocar l'acció dins un algorisme o una altra acció, ho expressarem així:

*nom de l'acció*( $obj_1, obj_2, \dots, obj_n$ );

On  $obj_i$  és una variable, constant, o una expressió de l'entorn definit en l'algorisme.

Mitjançant els paràmetres, l'acció es pot comunicar amb els objectes de l'entorn que l'ha invocat .

Si seguim l'exemple, en l'algorisme, la invocació formal de les accions per tal d'indicar a l'acció parametritzada *intercanvia* els objectes reals que pot utilitzar es farà com segueix:

```
{ Pre: (x = X i y = Y) i (z = Z i w = W) i (v = V i u = U) }  
intercanvia(x, y);  
intercanvia(z,          w);  
intercanvia(v, u);
```

```
{ Post: (x = Y i y = X) i (z = W i w = Z) i (v = U i u = V) }
```

En la primera invocació,  $x$  i  $y$  són els **paràmetres actuals** de l'acció *intercanvia*, en contrast amb els paràmetres formals, ja que aquests són els objectes concrets sobre els quals s'aplicarà l'acció. En la segona,  $z$  i  $w$  són els paràmetres actuals. Per tant, els paràmetres formals són els paràmetres que necessitem per a definir (formalitzar) l'acció. Els paràmetres actuals o reals són els objectes que s'utilitzen en la invocació.

La invocació *intercanvia*(*z*,*w*) és equivalent a:

```
aux := z;  
z := w;  
w := aux;
```

O, dit d'una altra manera,

```
{ z = Z i w = W }  
intercanvia(z, w);  
{ z = W i w = Z }
```

Com podeu observar, la correspondència entre paràmetres actuals i formals segueix l'ordre textual de definició. En el nostre exemple, *z* es correspon amb el paràmetre formal *x*, i *w* amb el paràmetre formal *y*.

El tipus definit per a cada paràmetre formal ha de coincidir amb el que tingui cada paràmetre actual. Així, doncs, en l'exemple d'intercanvi, no es poden posar com a paràmetres actuals, variables de tipus caràcter, ni qualsevol altre tipus que no sigui l'enter en el nostre exemple. En cada acció, però, es poden definir paràmetres de diferents tipus (el nostre exemple, no ho fa).

Els paràmetres poden ser de diferents tipus segons l'ús que se n'hagi de fer des de l'acció.

Els paràmetres es poden classificar en:

- **Entrada:** només ens interessa consultar el seu valor.
- **Sortida:** només interessa assignar-li un valor.
- **Entrada/sortida:** ens interessa consultar i modificar el seu valor.

Per a indicar que un paràmetre és d'entrada farem servir la paraula **ent**, si és de sortida, usarem la paraula **sor**, i si és d'entrada/sortida, la paraula **entsor**.

- Si un paràmetre és d'entrada, només ens interessa el seu valor inicial. Posteriorment, podem modificar aquest valor dins l'acció, però això no afectarà en cap cas els paràmetres actuals.
- Si un paràmetre és de sortida, el seu possible valor inicial no podrà ser llegit per l'acció. Això vol dir que la inicialització d'aquest paràmetre s'ha de fer en l'acció. Un cop executada l'acció, el valor del paràmetre actual corresponent es correspondrà amb el valor final dels seus paràmetres formals.
- Si el paràmetre és d'entrada/sortida, el valor del paràmetre podrà ser llegit i tota modificació del seu valor tindrà efecte amb els paràmetres actuals.

La sintaxi dels paràmetres *param<sub>i</sub>* en la sintaxi de la capçalera d'una acció) és la següent:

- per a un paràmetre d'entrada és:            *ent nom: tipus*
- per a un paràmetre de sortida és:            *sor nom: tipus*
- per a un paràmetre d'entrada/sortida és: *entsor nom: tipus*

Si un paràmetre formal és d'entrada, el paràmetre actual corresponent podrà ser una variable, una constant o una expressió. Si un paràmetre formal és de sortida o d'entrada/sortida el paràmetre actual corresponent haurà de ser una variable.

Podem per tant definir que un Objecte basant-nos en la vida real té una sèrie de característiques que fan del mateix un ens únic i que el seu funcionament de rol està supeditat en el seu conjunt de propietats i les accions pròpies que interactuen amb el entorn.

Per exemple, si tenim definida la capçalera d'acció següent:

**accio** miraQueFaig(**ent** valorEntrada: **enter**, **sor** resultat: **caracter**)

Podem invocar-la de les maneres següents:

var

nombre: enter;

elMeuCaracter: caracter;

fvar

...

miraQueFaig(nombre, elMeuCaracter);

...

o bé:

var

x, y, comptador: enter;

elMeuCaracter: caracter;

fvar

...

miraQueFaig(comptador \* (x - y) div 100, elMeuCaracter);

...

o bé:

var

x, y, comptador: enter;

elMeuCaracter: caracter;

fvar

...

miraQueFaig(50, elMeuCaracter);

...

En canvi, el següent seria incorrecte:

...

miraQueFaig(50, 'A');

...

Ja que "A" és una constant del tipus caràcter. Si és una constant, per definició no pot variar el seu valor, i això contradiu el fet que el paràmetre formal corresponent és de sortida (es pot modificar el valor del paràmetre).



## Funcions

També tenim la possibilitat de definir funcions. Les funcions sempre retornen un valor i només poden aparèixer en les expressions. Per tant, no es poden invocar per si soles.

A més, els paràmetres formals (arguments) de la funció que es defineixi només poden ser d'entrada. L'efecte d'invocar una funció dins una expressió és executar un conjunt d'accions que calculen un valor i, finalment, retornar aquest valor que ha de ser del tipus que s'hagi declarat en la capçalera.

La seva sintaxi és la següent:

```
funcio nom(param1, param2, ..., paramn): tipus  
    ...  
    ...  
    retorna expressió;  
ffuncio
```

On *parami* és *nom*: *tipus*. O sigui, com que els paràmetres han de ser sempre d'entrada, ens estalviem especificar que ho són amb la paraula reservada **ent**.

L'expressió *tipus* indicarà el tipus de valor que retornarà la funció.

Tota funció ha d'acabar en una sentència com "**retorna expressió**" on el valor resultant de l'avaluació d'*expressió* ha de ser del tipus declarat a la capçalera.

A l'igual que les accions, una funció pot tenir un entorn local.

Observeu que la diferència entre les accions i les funcions està en la forma d'invocar-les i en les restriccions dels seus paràmetres. La invocació d'una funció sempre ha de formar part d'una expressió, mentre que la d'una acció no en forma part mai. En una funció, els seus paràmetres sempre seran d'entrada i el valor que retorna és l'únic efecte de la funció. En una acció, els paràmetres no tenen aquestes restriccions.

Per exemple, podríem definir com a funció el producte basat en sumes que s'ha tractat en un subapartat anterior com:

```
{ Pre:  $x = X \wedge y = Y \wedge x > 0 \wedge y > 0$  }  
funcio producte(x: enter, y: enter): enter var  
    z: enter;  
fvar  
    z := 0;  
mentre y ≠ 0 fer  
    z := z + x;  
    y := y - 1;  
fmentre  
    retorna z;  
ffuncio  
{ Post: producte(x,y) = X * Y }
```

Si en algun punt d'un algorisme haguéssim de fer el producte de  $a$  i  $b$ , i posar el resultat en  $c$ :

```
var  
a, b, c: enter;  
fvar  
...
```

```
c := producte(a, b);
```

Si haguéssim de fer el producte de  $a * b * c * d$  i posar el resultat en  $f$ , cridaríem la funció de la manera següent:

```
var  
fvar    a, b, c, d, f: enter;  
  
    ...  
    f := producte(producte(producte(a, b), c), d);  
    ...
```

També podríem utilitzar la funció dins una expressió com ara:

```
var  
    a, b, c: enter;  
fvar  
    ...  
    c := 4 + 3 * producte(a, b);  
    ...
```

## Accions i funcions predefinides

En la notació algorísmica tindrem algunes accions i/o funcions ja predefinides.

### Funcions de conversió de tipus

Ja coneixem algunes funcions predefinides del llenguatge algorísmic: les funcions de conversió de tipus que hem vist quan parlàvem dels objectes en el primer apartat. Aquí teniu les declaracions de les capçaleres d'aquestes funcions.

- **funcio** *realAEnter*(*x: real*): **enter**
- **funcio** *enterAREal*(*x: enter*): **real**
- **funcio** *caracterACodi*(*x: caracter*): **enter**
- **funcio** *codiACaracter*(*x: enter*): **caràcter**

### Accions i funcions d'entrada i sortida de dades

Ara que ja heu vist com es dissenya un algorisme, potser us pregunteu com un algorisme es pot comunicar amb el seu entorn. El llenguatge algorísmic disposa també d'un conjunt d'accions i funcions predefinides que permeten als algorismes de rebre dades des del dispositiu d'entrada i enviar dades al dispositiu de sortida. D'aquesta manera, els algorismes poden rebre les dades amb què han de treballar i retornar els resultats obtinguts.

Aquestes funcions i accions són les següents:

- Funció que retorna un enter que s'ha introduït pel teclat del computador:

**funcio** *llegirEnter*(): **enter**

Funció que retorna un real que ha estat introduït pel teclat del computador:

**funcio** *llegirReal*(): **real**

- Funció que retorna un caràcter que ha estat introduït pel teclat del computador:

**funcio** llegirCaracter(): **caracter**

- Acció que visualitza per pantalla el valor de l'enter e:

**accio** escriureEnter(**ent** e: **enter**)

- Acció que visualitza per pantalla el valor del real r:

**accio** escriureReal(**ent** r: **real**)

- Acció que visualitza per pantalla el valor del caràcter c:

**accio** escriureCaracter(**ent** c: **caracter**)

Per tant, vegeu ara un algorisme complet que ja incorpora l'entrada i sortida de dades sobre el producte de naturals:

```
Algorisme producteNaturals
var
    x, y, z: enter;
fvar
    x := llegirEnter();
    y := llegirEnter();
    { Pre: x = X i y = Y i x > 0 i y > 0 }
    z := 0;
    mentre y ≠ 0 fer
        z := z + x; y := y - 1;
    fmentre
    { Post: z = X * Y }
    escriureEnter(z);
falgorisme
```

O bé també es podria escriure com:

```
algorisme producteNaturals
var
    x, y, z: enter;
fvar
    x := llegirEnter(); y := llegirEnter();
```

```
{ Pre:  $x = X \wedge y = Y \wedge x > 0 \wedge y > 0$  }  
z := producte(x, y);  
{ Post:  $z = X * Y$  }  
escriureEnter(z);  
falgorisme
```

```
funcio producte(x, y: enter): enter var  
z: enter;  
fvar  
{ Pre:  $z = Z$  }  
z := 0;  
mentre y ≠ 0 fer  
z := z + x; y := y - 1;  
fmentre  
{ Post:  $z = X * Y$  }  
retorna z;
```

**ffuncio**

### **Resum**

En aquest apartat s'han introduït els conceptes bàsics (algorisme, entorn, estat, processador, etc.) que utilitzarem al llarg de l'assignatura. Amb la introducció de la notació algorítmica, ara serem capaços d'expressar els nostres algorismes amb una notació formal i rigorosa per tal d'evitar a qualsevol ambigüitat.

Una idea central del mòdul és veure l'algorisme com l'agent que transforma un estat inicial de l'entorn de l'algorisme a un estat desitjat (i final) de l'entorn, passant progressivament per estats intermedis.

Ara ja podeu construir algorismes per a problemes a petita escala. En concret, si seguiu les pautes que us hem recomanat, caldrà fer el següent:

- 1) Entendre l'enunciat: especificarem l'algorisme que cal dissenyar. D'alguna manera, és una reformulació de l'enunciat amb termes més propers a l'algorítmica i eliminarem els dubtes que inicialment puguem tenir. És important no preocupar-nos de com ho farem, sinó de què hem de fer.

{ Pre: ... } → Comentem amb precisió (especifiquem) quin és l'estat inicial de partida.

Nom de l'algorisme... → Posem el nom de l'algorisme que cal dissenyar.

{ Post: ... } → Comentem amb precisió (especifiquem) quin és l'estat final després que

---

l'algorisme s'ha executat.

- 2) Plantejar el problema: encara no hem parlat gaire de com fer-ho des d'un punt de vista metodològic. Pel que sabem fins ara, la construcció d'un algorisme passarà per descobrir l'ordre temporal en què es compondran les accions amb les estructures escollides per tal d'assolir els objectius plantejats. El seguiment dels diferents estats intermedis pels que passarà l'algorisme abans d'arribar a l'estat final, ens guiarà l'ordre en què es compondran les diverses accions.

Els comentaris que fem per a especificar amb precisió l'estat de l'entorn en un instant donat, entre les accions que cal desenvolupar, ens ajudarà a raonar i construir la composició correcta d'accions.

- 3) Formular la solució: ja coneixem la sintaxi i significat de cadascun dels elements del llenguatge algorísmic que ens permetrà d'expressar amb precisió i rigor l'algorisme que resol el problema.

{ Pre: ... } → Comentem amb precisió (especifiquem) quin és l'estat inicial de partida.

**algorisme nom** → Descriu l'algorisme mitjançant el llenguatge algorísmic.

**const ... fconst** → Declarem els objectes constants (**enter**, **real**, **caracter**, **boolea**).

**tipus ... ftipus** → Declarem els tipus que necessitem (enumerats).

**var ... fvar** → Declarem les variables (**enter**, **real**, **caracter**, **boolea**).

... → Descriu la seqüència d'accions amb les tres estructures que coneixem:

seqüencial (;), alternativa (**si ... fsi**), i iterativa (**mentre ... fmentre**, **per ... fper**).

falgorisme

{ Post: ... } → Comentem amb precisió (especifiquem) quin és l'estat final després que l'algorisme s'ha executat.

A vegades tindrem la necessitat d'encapsular certes accions dins una acció i/o funció parametritzada. A aquest efecte disposem de:

{ Pre: ... } → Comentem amb precisió sota quines condicions l'acció farà l'efecte desitjat.

**accio** *nom(param1, ..., paramn)* → Definirem la capçalera de l'acció amb els seus paràmetres formals, i indicarem de quin tipus són amb **ent**, **sor** o **entsor**.

<Declaració entorn> → Declarem els objectes que cal utilitzar localment mitjançant **const** ...  
**fconst**, tipus ... **ftipus**, var

... **fvar**.

... → Descrivim la seqüència d'accions que cal fer amb

“,”, **si ... fsi**, **mentre ... fmentre**, **per... fper**.

**faccio**

{ Post: ... } → Comentem amb precisió l'efecte de l'acció. o { Pre: ... } → Comentem amb precisió sota quines condicions la funció farà l'efecte desitjat .

**funció** *nom(param1, ..., paramn): tipus* →

Definirem la capçalera de la funció amb els seus paràmetres formals d'entrada.

<Declaració entorn> → Declarem els objectes que cal utilitzar localment mitjançant **const** ...  
**fconst**, **tipus ... ftipus**, **var**

... **fvar**.

... → Descrivim la seqüència d'accions que cal fer usant

“,”, **si ... fsi**, **mentre ... fmentre**, **per... fper**.

**ffuncio**

{ Post: ... } → Comentem amb precisió l'efecte de la funció.

- 4) Avaluar la solució: per ara, hem de reflexionar a partir de la semàntica de cadascun dels elements del llenguatge algorísmic i, pas a pas, veure que la seva composició és adient. Els comentaris inserits entre accions ens ajudaran a reflexionar i raonar sobre la correctesa de l'algorisme.