Introducció a la metodologia de disseny descendent

Josep Vilaplana Pastó
M. Jesús Marco Galindo

PID_00149887

Universitat Oberta de Catalunya
www.uoc.edu

Índex

Introducció

En aquest mòdul aprendrem a desenvolupar algorismes per a problemes més complicats dels que hem vist fins ara. No introduirem cap element més del llenguatge algorísmic, sinó que, en tot cas, madurarem l'ús que farem d'alguns dels elements com la declaració d'accions i de funcions i la definició de tipus nous. És a dir, aquí proposem una metodologia que, si se segueix, ens ha de portar al disseny d'algorismes correctes i intel·ligibles que solucionen enunciats més complexos.

Penseu que...

... el seguiment de la metodologia que proposem ha de servir perquè els nostres dissenys es facin en un temps de desenvolupament mínimament raonable i augmenti el nostre grau d'eficàcia a l'hora de produir algorismes correctes. Recordeu els objectius que ens hem proposat en el mòdul "Introducció a la programació".

Fins ara ens hem dedicat, d'una banda, a aprendre la sintaxi i la semàntica del elements del llenguatge algorísmic. Amb la resolució de problemes senzills hem desenvolupat la destresa per a expressar els nostres algorismes de manera clara i no ambigua. Tot i això, per a dissenyar els nostres primers algorismes, hem estudiat unes pautes metodològiques basades en l'aplicació dels esquemes de tractament seqüencial. L'ús d'aquests esquemes ens ajuda a evitar errors típics en què podem caure, i també a formular algorismes intel·ligibles.

Així, doncs, la metodologia que s'imparteix en aquesta assignatura no és exclusiva d'aquest mòdul, sinó que en els altres mòduls ja es proposa progressivament una metodologia que s'ha de seguir per a problemes més senzills que els que oferirem a partir d'ara. Tot allò que hem après per als problemes senzills, però, ens ha de servir també per als problemes complexos que ens trobarem a partir d'aquest moment.

Els problemes senzills que hem resolt es caracteritzen pel fet que tenen una solució que utilitza un nombre raonable d'elements bàsics del llenguatge algorísmic.

Els **enunciats dels problemes** senzills fan referència a objectes que són molt propers als tipus elementals del llenguatge o que són directament representables per aquests.

El tractament que s'ha d'aplicar a aquests objectes es pot implementar amb unes quantes assignacions i algunes construccions algorísmiques. El resultat final és un algorisme comprensible i llegible en termes dels elements bàsics del llenguatge algorísmic.

Quan desenvolupem aplicacions més reals ens trobarem amb problemes grans i complexos.

Els **enunciats de problemes complexos** fan referència a objectes que no tenen una correspondència directa amb els tipus elementals de la notació algorísmica.

El tractament d'aquests objectes pot no ser elemental i pot portar a un nombre de sentències i declaracions excessivament gran, si expressem tot el que cal fer com una composició seqüencial d'assignacions, iteracions i/o alternatives.

El resultat és un algorisme de moltes pàgines difícil de seguir (i, per tant, de saber si funciona) ple de detalls (alguns cops repetits) que el fan pràcticament il·legible. El nombre de detalls que s'han de tenir en compte és excessiu i la nostra capacitat de ser-ne conscients de tots és limitada. Pensar aquests enunciats en termes dels elements bàsics del llenguatge sense unes pautes que s'hagin de seguir ens pot portar molts maldecaps i dispersió mental, la nostra capacitat de comprensió disminuirà i, per tant, també la nostra seguretat sobre la correctesa i legibilitat (manteniment) de l'algorisme. Amb aquests arguments arribem a la conclusió que la notació algorísmica és tan elemental que fa incòmode i poc pràctic el desenvolupament d'aplicacions reals, i que caldria un llenguatge més còmode que comprengués objectes més reals.

Us imagineu...

... què aconseguiríem si exposéssim el funcionament d'un programa en termes del moviment d'electrons que hi ha en el silici del vostre processador i per les pistes de coure de la placa de circuit imprès que teniu al vostre ordinador? Acabaríem confosos i dispersos per l'excés d'informació en l'exposició i perdríem tota referència sobre allò de què parlem.

D'altra banda, si aquest llenguatge existís, hauria de tenir un nombre tan gran d'accions i tipus per a poder resoldre la gran quantitat i diversitat de problemes que hi ha, que mai no l'acabaríem de conèixer a fons. Es poden trobar alguns llenguatges específics per a algunes aplicacions concretes, però quan no trobem allò que busquem, què fem?

En realitat, la notació algorísmica té prou elements per a construir qualsevol algorisme que vulguem a partir dels tipus elementals, l'assignació i les composicions algorísmiques. De la mateixa manera, el llenguatge algorísmic ens permet ampliar el llenguatge propi perquè ens permet definir i construir tipus més complexos, i definir les accions i/o funcions que més ens convinguin.

Amb les característiques del llenguatge algorísmic, tot allò que ens falta és saber quina metodologia cal seguir per a portar a bon terme el disseny d'un problema complex. La metodologia consisteix a descompondre un problema complex en subproblemes més senzills i independents entre si. La solució d'aquests subproblemes ha d'implicar la solució del problema. La descomposició del problema es fa mitjançant l'abstracció.

L'**abstracció** és el resultat d'aïllar determinats aspectes (qualitat, atributs, etc.) d'un tot per a poder raonar o pensar de manera més còmoda i menys dispersa (en el nostre cas, pensar en la solució de problemes).

En el fons, es tracta de simplificar alguna realitat i reduir-la a només alguns aspectes d'aquesta que tinguin una rellevància o interès especial per al que pensem o fem.

De fet, hem fet servir l'abstracció gairebé des de l'inici de l'assignatura (per exemple, el model de seqüència, els esquemes –que podríem considerar com algorismes abstractes–, etc). Però a partir d'aquest mòdul haurem d'aprendre a abstreure de manera adequada problemes més complexos que els que hem treballat fins ara. Haurem de descompondre el problema en subproblemes més senzills. Aquest mòdul us donarà les tècniques necessàries per a fer-ho (eines de la notació per a abstreure dades i abstreure codi) i una metodologia per a seguir (disseny descendent).

Objectius

Quan haureu estudiat aquest mòdul, haureu assolit els objectius següents:

1. Ser capaços d'analitzar problemes de certa complexitat i descompondre'ls en subproblemes més senzills amb l'aplicació de la tècnica de disseny descendent per refinaments successius.

2. Adonar-se del fet que a l'hora de descompondre els problemes per al disseny, hi ha maneres de fer-ho més convenients i efectives que d'altres. L'elecció d'abstraccions convenients al problema i el seguiment de les pautes indicades en aquest mòdul us han de portar a descomposicions efectives.

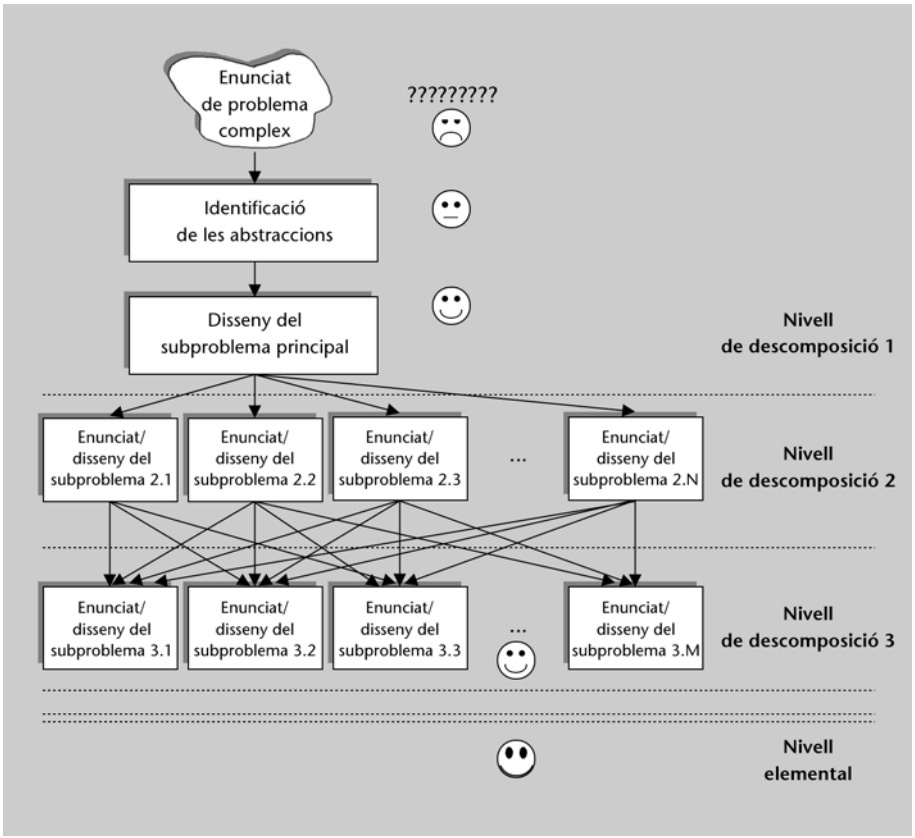
3. Saber aplicar els conceptes estudiats en els altres mòduls per a resoldre cadascun dels subproblemes senzills mitjançant l'abstracció de codi i dades.

4. L'assoliment dels objectius anteriors us permetrà desenvolupar algorismes de manera efectiva i en un temps raonable que resolguin problemes més reals.

1. Disseny descendent

1.1. Anàlisi de problemes complexos

Com ja hem avançat, la manera d'encarar-se a un problema complex consisteix a descompondre'l en una col·lecció de subproblemes més senzills i independents de manera que la solució d'aquests impliqui la solució del problema en qüestió. Alhora, si els subproblemes no són prou senzills, es poden descompondre en altres subproblemes. Per aquest motiu parlarem de *nivells de descomposició*. O sigui, el primer nivell descompondrà el problema en els subproblemes que calgui. En un segon nivell, descompondrem cadascun dels subproblemes resultants de la descomposició del primer nivell, etc. El nombre de nivells de descomposició en subproblemes dependrà de la complexitat del problema. En el darrer nivell trobarem els subproblemes més senzills de resoldre, és a dir, els més concrets i elementals. La solució de cadascun dels subproblemes de cada nivell contribueix parcialment a la solució global del problema. Vegeu la figura següent:



Només mitjançant la pràctica serem capaços de fer bones descomposicions. No qualsevol descomposició d'un problema en subproblemes és bona per a desenvolupar un bon disseny. La descomposició s'hauria de basar en algun criteri guiat per les abstraccions del problema que creiem convenients.

Per a il·lustrar els conceptes que es donen en aquest apartat us proposem el problema següent: una editorial està interessada a analitzar els errors d'un text presentat per un autor de materials. Ens encarrega de dissenyar un algorisme que, un cop feta l'anàlisi d'errors d'un text, ens digui el percentatge de frases, la mitjana de paraules per frase, i el màxim de paraules per frase que s'han de corregir. Si no cal corregir el text, a la sortida no es diu res.

A l'entrada de l'algorisme tindrem la seqüència de caràcters del text. Cada frase acaba amb una paraula finalitzada amb un punt.

Cada paraula està separada de la següent per un únic espai. En el cas que la paraula acabi en punt, l'espai és a continuació del punt. A l'inici del text no hi ha espais. Les paraules que cal corregir contenen un o més signes d'asterisc (*). Quan una paraula consisteix només en el caràcter asterisc, és una correcció que s'ha de fer en la frase (per exemple, una paraula que s'ha d'afegir).

S'ha convingut que el final del text original està indicat amb una frase que no té cap paraula: simplement hi ha d'haver el caràcter punt de final de frase.

Per exemple, si a l'entrada tenim el següent:

Hola qu* tal. Com B*a ai*o. .

A la sortida tindrem això:

100 1.5 2

El disseny pot arribar a ser complicat si pensem l'aplicació a partir de la seqüència de caràcters. En canvi, si proposem un conjunt d'abstraccions naturals i convenients per al problema, el desenvolupament ha de ser més senzill de fer. Aquí ens és útil pensar que el text és una seqüència de frases i que s'ha de proposar un algorisme que vagi obtenint i tractant cada frase. Després treballaríem el nivell següent, que consisteix a plantejar el subproblema de com s'obté una frase, el subproblema de com se l'ha de tractar i com s'identifica que es tracta de la darrera frase. La consideració d'una frase com una seqüència de paraules ha de ser convenient i ha de donar lloc a la creació dels subproblemes dedicats a les paraules que es resolen en el darrer nivell com a problemes de seqüència de caràcters.

L'ús de les abstraccions *frase* i *paraula* són pròpies i naturals al problema, i la seva correspondència en un nivell de descomposició fa que ens puguem concentrar en els detalls més concrets d'una abstracció, i deixar per a més endavant els detalls de les abstraccions que estiguin pendents de resoldre. Hem de procurar que cada nivell tingui un nombre de detalls que el faci fàcil de seguir i assegurar la correctesa de les solucions plantejades.

1.2. Anàlisi ascendent enfront d'anàlisi descendent

El primer nivell de descomposició és el més abstracte, els següents són menys abstractes o més concrets, fins que s'arriba al darrer, que és el més concret. En el cas de l'exemple, el nivell més concret és la seqüència elemental de caràcters.

Quan es planteja un problema complex, hi ha dues direccions possibles per a la seva anàlisi: la primera consisteix a resoldre els problemes més concrets per a resoldre després els més abstractes. És a dir, anem del més concret al més abs-

tracte. Aquesta manera de plantejar-ho s'anomena **anàlisi ascendent**. La segona direcció consisteix a resoldre primer el nivell més abstracte, i continuar després amb els nivells menys abstractes fins que s'arriba al nivell més concret. Anomenem aquesta segona direcció **anàlisi descendent**. És el que hem suggerit en l'exemple.

L'anàlisi ascendent està plena de dificultats i no és adequada per a portar a bon terme un disseny en un temps de desenvolupament raonable. La dificultat principal rau en el fet que quan no es resolen en primer lloc els nivells més abstractes, no sabem realment quins problemes concrets cal solucionar. Així, podem resoldre problemes concrets que després no es faran servir o que no estan convenientment definits per a ser usats en els nivells més abstractes. Això portaria a redissenyar el nivell més concret i, en conseqüència, retrocedir en el disseny per a després tornar a l'altre nivell. En l'exemple que treballem, imagineu que no teniu gaire clar el tractament que s'ha de fer i us dediqueu a desenvolupar una acció que llegeixi una paraula i la guardi en una estructura de dades, per a descobrir després en el nivell més abstracte que no cal el contingut exacte d'aquella paraula, sinó que només cal saber si cal corregir-la i comptar-la en el cas que així sigui. L'estructura de dades pot sobrar i servir de molt poc per al problema, sobretot si posa limitacions a l'enunciat, com veurem més endavant.

En l'anàlisi descendent, les coses van millor. Quan es comença en el nivell més abstracte, identifiquem i definim els subproblemes que cal resoldre en el nivell menys abstracte següent. D'aquesta manera, fixem els objectius per al següent nivell menys abstracte i no farem més feina que la que realment cal per a resoldre el problema global, escurçarem el temps de desenvolupament, i evitarem retrocessos. L'exemple que treballarem seguirà aquesta línia. L'inconvenient d'aquesta estratègia és que, quan es té poca experiència pràctica en el mètode i en el disseny d'algorismes, hi ha la inseguretat de si més endavant es podrà resoldre algun dels subproblemes proposats en el nivell abstracte, i sovint es prefereix plantejar el problema de manera ascendent per a assegurar-se que té una base de problemes concrets resolts que han d'ajudar a resoldre els nivells més abstractes. Si ho fem així, saltem de nivells i arribem a confusions innecessàries. La pràctica del mètode descendent en diversos problemes és l'única manera d'assimilar el mètode i superar aquest inconvenient inicial, ja que l'anàlisi ascendent o una anàlisi combinada ascendent i descendent fa que el desenvolupament sigui un procés més lent, sovint confós, dispers i amb incoherències en els resultats.

El disseny descendent és un mètode més eficaç i ràpid per a desenvolupar problemes complexos que el mètode ascendent o un mètode que els combini tots dos.

1.3. Orientacions de les abstraccions

Disposem de dues orientacions per a fer abstraccions d'un problema o subproblema, i així guiar-ne la descomposició en subproblemes:

- **Orientació funcional**: allò que es demana en l'enunciat del problema consisteix en un tractament molt ampli en detalls i passos. Aquest tractament ampli

es pot descompondre en diverses etapes conceptualitzades de manera que cadascuna faci una funció parcial que contribueixi a la solució global del problema. Un cop identificades, expressem la composició algorísmica d'aquestes etapes i, més endavant, desenvolupem cada etapa per separat.

- **Orientació a l'objecte**: un objecte real del problema no és directament representable pels tipus elementals del llenguatge algorísmic. Definim un tipus convenient per a l'objecte i, al llarg de tot el disseny, per a fer qualsevol canvi d'estat o accés de l'objecte, caldrà dissenyar expressament unes accions i/o funcions per al tipus d'objecte definit. És a dir, associem unes accions i/o funcions al tipus, i només aquestes accions i/o funcions modificaran o consultaran l'estat d'un objecte del tipus definit.

Podem trobar un exemple d'abstracció funcional en el problema següent:

Es disposa de les temperatures mitjanes diàries d'un mes en una taula. Es vol dissenyar un algorisme que faciliti quin dia del mes té la temperatura més propera a la temperatura mitjana d'aquest mes.

Per a calcular la temperatura mitjana del mes cal recórrer tota la taula. Un cop calculada, s'ha de cercar el valor de temperatura diària que més s'hi apropa. D'aquesta manera, hem dividit el problema en dos subproblemes més concrets segons la seva funció, que són els següents:

1) Càlcul de la mitjana de temperatures d'un mes.
2) Cerca del dia en què la temperatura s'apropa a la temperatura mitjana del mes.

Tots dos problemes treballen sobre el mateix objecte, que és una taula de temperatures mitjanes diàries d'un mes.

En l'exemple que hem proposat per a treballar al llarg de l'apartat, s'han fet abstraccions per objectes. Hem interpretat part de la informació implícita que guarda la seqüència de caràcters com frases i paraules. Fixeu-vos que podem considerar un text de diverses maneres, per exemple, com una seqüència de capítols d'un llibre, els capítols com una seqüència d'apartats, etc. Passem per alt el fet que el text d'entrada s'organitzi en paràgrafs o no, ja que no té cap utilitat en la resolució del nostre problema. La força de l'abstracció rau aquí: només considerem allò que és rellevant per al nostre problema. En cada nivell hem de definir la representació que cal que tingui cada abstracció escollida i les accions i/o funcions s'associaran a cada abstracció.

1.4. Maneres de treballar i presentar el disseny descendent

Necessitem expressar d'alguna manera el desenvolupament mitjançant la metodologia del disseny descendent. Una forma podria consistir a proposar en el primer nivell un esborrany d'algorisme informal.

En altres paraules...

... podem imaginar que el nivell és un processador virtual (o un llenguatge virtual) que sap què és una seqüència de frases i com s'obtenen i es tracten.

Vegem què passa en el nostre exemple si resolem el primer nivell. Primer de tot, hem d'establir els objectius del disseny mitjançant l'especificació i mantenir el nivell d'abstracció que ens hàgim proposat. En el nostre cas, ens hem d'imaginar el text com una seqüència de frases.

Un cop s'ha examinat l'enunciat, escrivim l'especificació següent:

{ Pre: a l'entrada tenim una seqüència de frases, *T*, que pot ser buida. Cada frase de *T* conté informació per a saber si cal corregir-la o no }
AnalisiCorreccions
{ Post: en el cas que no calgui corregir *T*, no s'escriu res. En el cas que *T* sigui un text que s'ha de corregir, a la sortida s'ha de generar una seqüència *R* formada per tres elements: *R1 R2 R3*.
R1 representa la mitjana de paraules que s'han de corregir per frase de *T*,
R2 representa el percentatge de frases que s'han de corregir a *T*, i
R3 representa el màxim de paraules que s'han de corregir per frase de *T* }

Es demana dur a terme una anàlisi del text per a fer una estadística de les correccions que s'hi han de fer. El disseny de la solució es basarà en un esquema de recorregut, atès que les dades de l'anàlisi del text han de dependre de les dades parcials que ha d'aportar cadascuna de les frases del text. Les accions de l'anàlisi i també l'obtenció de la seqüència anirien a càrrec del nostre processador virtual o el nostre llenguatge virtual intel·ligent. Així, doncs, si apliquem l'esquema de recorregut, obtenim informalment el següent:

algorisme analisiCorreccions
  var ... **fvar**

  "Inicialitzar els resultats de l'anàlisi del text."
  "Obtenir la primera frase."
  mentre "la frase obtinguda no sigui la darrera frase" **fer**
    "Actualitzar els resultats de l'anàlisi del text a partir de la frase actual."
    "Obtenir la frase següent."
  fmentre
  "Escriure els resultats de l'anàlisi del text."
falgorisme

En el nivell següent refinaríem cadascuna de les accions o funcions expressades informalment amb possibles nous noms informals. Així continuaríem fins que es fa un algorisme completament formal i concret. Podeu veure que aquesta manera d'expressar la resolució té l'inconvenient que en cada nivell hem de reescriure l'algorisme, i al final del desenvolupament, si tenim molts nivells, podem tenir un algorisme llarg i difícil de comprendre pels nombrosos detalls que comprèn.

Aquesta tècnica de refinament...

... és la que hem fet servir per a aplicar els esquemes. Els esquemes es poden veure com un esborrany d'algorisme, que concretem en el nivell següent.

Una manera millor d'expressar el nostre disseny es basa en les eines que ens proporciona la notació algorísmica. Cada subproblema que s'ha de resoldre es pot encapsular mitjançant accions i/o funcions parametritzades. Cada objecte complex es pot encapsular mitjançant els constructors de tipus o amb la declaració de tipus nous (com ara enumeratius o d'altres). D'aquesta manera, a cada nivell escrivim un algorisme, acció i/o funció amb total precisió i formalitat i no reescrivim les parts del disseny ja resoltes. L'ús de paràmetres ens ha de permetre d'assolir la màxima independència entre nivells i entre els subproblemes plantejats, la qual cosa permet que ens concentrem en una parcel·la limitada del problema global.

L'especificació d'accions i/o funcions tindrà una doble utilitat: la primera és fer de pont entre nivells per a verificar la correctesa de les solucions plantejades assumint que les accions i/o funcions emprades en el nivell d'estudi compleixen l'especificació donada. La segona utilitat és que l'especificació d'una acció i/o

funció és en si l'enunciat formalitzat del subproblema que s'ha de resoldre en el nivell que correspongui. Si en el disseny de l'acció i/o funció es respecta l'especificació, el disseny global ha de funcionar com s'espera que ho faci.

Així, doncs, reflectirem l'anàlisi descendent del problema mitjançant els mecanismes d'encapsulació que ens ofereix la notació algorísmica (definició de tipus, constructors de tipus, accions i/o funcions).

En el nostre exemple, el resultat del primer nivell resta de la manera següent:

```
algorisme analisiCorreccions

  var
    resultatsAnalisiText: tDadesAnalisi;
    frase: tFrase;
  fvar

  { Pre: en l'entrada tenim una seqüència de frases, T, que pot ser buida.
    Cada frase de T conté informació per a saber si cal corregir-la o no }

  iniciResultatsAnalisi(resultatsAnalisiText);
  obtenirFrase(frase);

  mentre no darreraFrase(Frase) fer
    actualitzaResultatsAnalisi(frase, resultatsAnalisiText);
    obtenirFrase(Frase);
  fmentre

  escriureResultatsAnalisi(resultatsAnalisiText);
  { Post: en el cas que no calgui corregir T, no s'escriu res.
    En el cas que T sigui un text que s'ha de corregir,
    en la sortida s'ha de generar una seqüència R formada per tres elements: R1 R2 R3.
    R1 representa la mitjana de paraules que s'han de corregir per frase de T,
    R2 representa el percentatge de frases que s'han de corregir a T, i
    R3 representa el màxim de paraules que s'han de corregir per frase de T }
falgorisme
```

En la solució present...

... només hi ha els comentaris de la precondició i la postcondició. L'elecció d'uns noms apropiats per a les accions, funcions i variables fa que no calguin més comentaris si després especifiquem cadascuna de les accions i/o funcions. Això ho veurem més endavant i completarà el nivell actual de treball.

Observeu que la darrera solució presentada per al primer nivell és sintàcticament correcta, però no està completa perquè falta per definir el tipus *tFrase*, i un conjunt d'accions i/o funcions (*obtenirFrase* i *darreraFrase*), i el tipus *tDadesAnalisi* amb les seves accions (*iniciResultatsAnalisi*, *actualitzaResultatsAnalisi* i *escriureResultatsAnalisi*) que s'han de definir en un segon nivell.

Malgrat que la solució fins ara formulada no està completa, hi ha un altre avantatge que podem aprofitar: en podem raonar la correctesa sense necessitat de conèixer els detalls dels tipus que encara no s'han definit, ni de com les accions i/o funcions han de fer l'efecte que s'espera d'aquestes i que encara s'han de desenvolupar. L'especificació de cadascuna d'aquestes ens indica quin és l'efecte que té cada acció o funció i, d'acord amb això, podem raonar sobre la correctesa de la solució formulada. En desenvolupar el nivell següent, si les accions o funcions respecten les precondicions i postcondicions, la correctesa del nivell s'ha de mantenir. Aquí, l'especificació fa un paper de pont entre nivells important, ja que ajuda a separar el què del com.

La correctesa de la solució està assegurada pel fet que hem aplicat l'esquema de recorregut a una seqüència de frases. En la solució proposada hem considerat una darrera frase que actua com a marca de final de seqüència. Cal esbrinar si aquesta frase és només una marca o bé és a la vegada una frase que cal tractar i una marca.

Un dels punts forts dels esquemes...

... rau en el fet que coneixem tots els detalls del seu funcionament o comportament, i ens serveixen de base per a raonar còmodament sobre la correctesa de l'algorisme o del tros d'algorisme resultant de l'aplicació dels esquemes.

Si examinem l'enunciat, observem que al final del text tindrem dos caràcters ".". El primer punt pertany a la darrera frase "real" del text. El segon punt es pot veure com un punt que tanca una frase buida (no té cap paraula) i serveix només per a marcar el final del text. La frase buida és la que ha d'indicar la fi de la seqüència de frases i no cal que la tractem (en aquest cas, que l'analitzem). Per tant, només cal analitzar cada frase llegida no buida, i acumular els resultats de l'anàlisi en l'objecte *resultatsAnalisiText*. Quan sortim de la iteració **mentre**, haurem analitzat totes les frases "reals" del text (és a dir, amb caràcters i, per tant, amb contingut), i *resultatsAnalisiText* tindrà el resultat de l'anàlisi global del text.

A més, el fet que l'algorisme no entri en detalls del tipus no definit, sinó que la responsabilitat d'aquests rau en les accions i/o funcions que tracten el tipus, fa que la definició concreta de *tFrase* sigui irrellevant per a raonar la correctesa d'aquesta etapa de disseny. En el segon nivell, haurem de definir els tipus *tFrase* per a treballar-hi amb els detalls que creguem convenients. Podrem estudiar la millor alternativa, i no tindrà cap efecte sobre el primer nivell sempre que es respecti el significat, implícit en aquests moments, de les accions i/o funcions que utilitza l'algorisme del primer nivell i que es desenvoluparan en el segon nivell. Podrem comprovar que cada acció o funció fa allò que ha de fer sense haver de retrocedir a etapes anteriors del disseny.

L'orientació a l'objecte ens facilita una separació molt efectiva dels problemes per a poder-los descompondre en subproblemes independents.

Abans de continuar amb el problema, farem algunes observacions. La possibilitat que la notació algorísmica ens ofereix definir accions i/o funcions permet encapsular subproblemes, tal com hem vist en la darrera solució de l'exemple. Els constructors de tipus i la declaració de nous tipus ens permet encapsular les abstraccions de dades. Així, doncs, la notació ens dóna tots els elements necessaris per a poder reflectir en el disseny les abstraccions i la metodologia emprada en les descomposicions. A més, si seguim les pautes, podem evitar de reescriure o modificar cadascuna de les etapes prèvies.

1.5. Eines de la notació per a l'abstracció de dades

Comencem per parlar de com s'encapsulen les abstraccions de dades. Per exemple, intentem definir l'abstracció de frase del nostre exemple. Podem veure la frase com una seqüència de paraules o mots i també com una seqüència de caràcters. Imaginem que en l'aplicació que desenvolupem ens demanen de detectar frases capicua. Quan arribarem al nivell que tracta l'abstracció de frase, ens

adonarem que ens cal guardar tota la frase per a comparar el darrer caràcter amb el primer, el penúltim amb el segon, etc., aplicant un esquema de cerca dins una funció que ha de dir si una frase és capicua o no. Per a definir els tipus podem fer ús dels constructors de tipus que ens ofereix el llenguatge: taules i tuples.

Per exemple, la frase "Senén té sis nens i set nenes" és capicua.

En el nostre cas, com que la frase consisteix en dades homogènies (caràcters), utilitzarem les taules. Per exemple:

```
tipus
   tFrase = taula[maxCarFrase + 1] de caracter;
ftipus
```

on *maxCarFrase* és el màxim de caràcters que preveiem que tindrà cada frase. L'element de més és per al sentinella, ja que *a priori* no sabem el nombre de caràcters. Com que ens interessa de comparar el darrer amb el primer, etc., ens pot ser més útil tenir la longitud de la frase en comptes del sentinella. Però si declarem el següent:

```
var
   nCaractersFrase: enter;
   frase: tFrase;
fvar
```

dispersem l'abstracció de frase en dos conceptes: la seva longitud i la frase en qüestió. Aquesta dispersió pot semblar ridícula, però si després ens veiem obligats a guardar més d'una frase, haurem de declarar més comptadors de caràcters per a les diferents taules. A més, hauríem de canviar els paràmetres de les accions i/o funcions en què no ho hàgim previst per a tenir en compte els comptadors nous. És a dir, anem d'un nivell a l'altre i reescrivim el codi ja resolt, i això no ens interessa. El millor és encapsular totalment tot allò que faci referència a l'abstracció que implementem: 

```
tipus
   tFrase = tupla
               nCaracters: enter;
               c: taula[maxCarFrase] de caracter;
            ftupla
ftipus
```

D'aquesta manera, tenim els detalls de l'abstracció que ens interessen dins el tipus declarat, i en el nivell més abstracte no cal fer cap canvi, ja que els detalls s'han de treballar en el nivell de l'abstracció de frase.

Imaginem ara que ens demanen esbrinar si una frase del text té les mateixes paraules d'una frase donada, però amb ordre diferent. Més breument, podem dir que una frase és permutació d'una altra. Sense entrar en detalls del problema, sembla clar que cal veure la frase com una seqüència de paraules i, per tant, l'anterior definició de *tFrase* no ens serviria, ja que hem d'estructurar la

Per exemple, la frase "la mar estava serena" és una permutació de la frase "serena estava la mar".

frase en una col·lecció de paraules per a comparar-les entre si. Una estructura gens útil és la següent:

```
tipus
   tFrase = tupla
               nMots: enter;
               c: taula[maxMotsFrase, maxCarMot+1] de caracter;
            ftupla
ftipus
```

on *maxMotsFrase* és el nombre màxim de paraules que pot tenir una frase, *maxCarMot* és el màxim de caràcters que pot tenir una paraula, i cadascuna de les files de la matriu de caràcters guarda un sentinella per a cada paraula. Malgrat que la informació estigui estructurada, és críptic de tractar aquesta estructura perquè no reflecteix del tot l'abstracció de mot que ens convé per a resoldre més còmodament el problema nou sense dispersió d'abstraccions. En canvi, la definició següent és molt més apropiada i menys problemàtica de tractar:

```
tipus
   tFrase = tupla
               nMots: enter;
               m: taula[maxMotsFrase] de tMot;
            ftupla
ftipus
```

On *tMot* s'ha de definir en el nivell corresponent amb les accions i funcions associades que calguin.

Així, doncs, podem observar que l'abstracció de dades es pot fer mitjançant els constructors de tipus, però no cal oblidar que en l'abstracció recollim allò que ens interessa i ens convé perquè és rellevant al problema que resolem. Teniu ja la solució que s'ha d'aplicar al tipus *tFrase* del nostre problema inicial?

Penseu la solució abans de continuar.

Doncs, aquí us la presentem:

```
tipus
   tFrase = tupla
               nMotsC: enter;
               final: boolea;
            ftupla
ftipus
```

Tot allò que ens interessa d'una frase per al nostre exercici inicial, a més de saber si es tracta de la frase final, és saber el nombre de mots que té per a corregir. Així, doncs, la tupla definida compleix els objectius que ens hem proposat per al problema.

Potser podeu pensar que algun dels tipus on es guarda el contingut de la frase també serveixen per al nostre objectiu, ja que després podem desenvolupar les accions i/o funcions que comptabilitzen el nombre de caràcters i de mots a partir del qual està guardat. Però si pensem així, ens oblidem dels objectius que perseguim quan fem un disseny. Primer, tenim una raó de sentit comú: per què hem de guardar

coses que després no farem servir? L'abstracció de dades consisteix a simplificar els detalls de l'objecte en estudi per a posar aquelles dades que són rellevants al problema, i descartar tot allò que no és útil. Si fem un estudi que consisteix a fer una estadística d'alçades d'un grup d'individus, ens serveix d'alguna cosa el color dels seus ulls per a la finalitat de l'estudi?

També hi ha raons tècniques que ajuden als objectius de disseny que perseguim. Primer, cal fer un algorisme que compleixi els objectius proposats (postcondició) per l'enunciat a partir de les condicions inicials que estableixi (precondició). A l'entrada tenim un text que és una seqüència de caràcters. Quants caràcters hi pot haver en una seqüència, quants mots hi pot haver? En principi, no ho sabem. Si declarem taules, hem de definir màxims, i en conseqüència restringir l'entrada (text) de l'enunciat a condicions, que si se satisfan, l'algorisme funcionarà correctament. Si no se satisfan, hem de modificar les constants de l'algorisme. Tot aquest enrenou és per a formular un algorisme que no serà del tot correcte, ja que no admet qualsevol frase com demana l'enunciat. A més, per a comptar els mots d'una frase no ens cal tenir prèviament la frase guardada i és un caprici posar restriccions a l'enunciat que en el fons no calen. Ens hem d'ajustar a l'enunciat, i no pas al revés.

Una altra raó és la qüestió de l'eficiència: guardar una frase per al nostre problema ocupa un espai de memòria innecessari, de manera que abusem d'aquest recurs sense que això tingui una justificació.

En els problemes de la frase capicua i la permutació de frases té sentit que la frase es guardi, ja que ens cal l'accés directe que ens proporcionen les taules. En aquests problemes té sentit d'exigir a l'enunciat límits allà on calgui, i l'espai que ocupen les dades està plenament justificat per la seva necessitat.

1.6. Eines de la notació per a l'abstracció de codi

Un cop s'ha resolt l'abstracció de dades en el nivell corresponent, cal dissenyar les accions i funcions que s'han proposat en el nivell anterior en fer una abstracció de codi. De fet, el primer nivell encara s'ha de completar donant significat a cadascuna de les accions i funcions que s'han proposat. El mitjà és l'especificació i així establim un pont entre nivells que assegura la coherència entre si. Per a la frase, tenim el següent:

accio obtenirFrase(**sor** f: tFrase)
  { Pre: a l'entrada tenim un text *T* representat com una seqüència de frases *t*.
    A la part dreta de *t*, *DT*, hi ha una seqüència de frases o la frase buida }

  { Post: a *f* tenim representada la primera frase de *DT*.
    La frase representada està a la part esquerra de *t* }

funcio darreraFrase(f: tFrase): **boolea**
  { Pre: *f* = *F* }
  { Post: *darreraFrase*(*F*) és **cert** si *F* és una frase buida.
    Ha de retornar **fals** en cas contrari }

Per a l'anàlisi del text a partir de l'objecte de tipus *tAnalisiDades,* ens hem de centrar en la postcondició del problema. Recordeu que hem d'obtenir el percentatge de frases, la mitjana de paraules per frase, i el màxim de paraules per frase que s'han de corregir. Les dues primeres dades s'obtenen a partir dels elements següents:

- nombre de frases del text (*nFrases*),
- el nombre de frases que s'han de corregir (*nFrasesC*) del text, i
- el nombre de paraules que s'han de corregir del text (*nMotsC*),

on hem posat entre parèntesis uns noms que ens han de servir per a abreviar l'especificació. Abreviarem també el màxim de paraules per frase mitjançant el nom *mx*. Si seguim aquests noms, l'especificació resulta de la manera següent:

accio iniciResultatsAnalisi(**sor** a: tDadesAnalisi)
{ Pre: }
{ Post: a *a* tenim *nFrases*, *nFrasesC*, *nMotsC* i *mx* d'un text buit }

accio actualitzaResultatsAnalisi(**ent** f: tFrase; **entsor** a: tDadesAnalisi)
{ Pre: $a = A$ i $f = F$ }
{ Post: a *a* tenim *nFrases, nFrasesC, nMotsC* i *mx* del text
resultant d'afegir l'anàlisi de lafrase *F* al text ja analitzat a *A* }

accio escriureResultatsAnalisi(**ent** A: tDadesAnalisi)
{ Pre: $a = A$ }
{ Post: si les dades de *A* són d'un text que s'ha de corregir,
pel canal de sortida s'escriu una seqüència *R* formada per tres elements: *R1 R2 R3*.
R1 representa la mitjana de paraules que s'han de corregir
per frase present en el text analitzat en *A*,
R2 representa el percentatge de frases que s'han de corregir
en el text analitzat en *A*, i
R3 representa el màxim de paraules
que s'han de corregir per frase existent en el text analitzat en *A*.
En el cas que *A* tingui les dades d'un text buit, no s'escriu res }

Fixeu-vos que amb l'especificació establim la separació entre nivells:

1) Amb l'especificació a la mà podem comprovar si el primer nivell és correcte sense haver de conèixer els detalls interns de cada acció o funció.

2) Cada tripleta (capçalera acció o funció)-(precondició)-(postcondició) és l'enunciat formalitzat d'un subproblema que s'ha de resoldre en el segon nivell, i marca els objectius que s'han de treballar.

D'aquesta manera, les accions i funcions de la notació algorísmica permeten encapsular els subproblemes que necessitem sense necessitat de reescriure l'algorisme, ja que un cop s'hagin dissenyat aquestes, el disseny estarà complet. Com a eines d'abstracció de codi, les accions i funcions permeten separar el què i el com en el plantejament d'un nivell. És a dir, escriure en l'indret adequat del text de l'algorisme què volem mitjançant el nom d'una acció o funció amb els paràmetres adequats sense pensar en com l'acció o funció ha d'arribar a fer el seu propòsit. D'aquesta manera, podem mantenir les abstraccions que considerem convenients en el nivell que treballem.

Observeu que...

... l'especificació també ens permet dividir el treball de l'aplicació que s'ha de desenvolupar entre un grup de persones, de manera que cada persona no depengui de l'altra per a fer el desenvolupament.

Adoneu-vos també que la parametrització d'accions i funcions és la responsable del fet que els entorns de cada problema o subproblema estiguin convenientment separats. De fet, els paràmetres ens ajuden a formalitzar les entrades i sortides del subproblema en qüestió. Amb els paràmetres establim el contacte amb l'exterior de l'acció o funció. Amb els paràmetres actuals usem l'acció a l'entorn que hem decidit que cal modificar i/o consultar. Amb els paràmetres formals dissenyem l'acció o funció sense pensar qui la utilitzarà, i simplement decidim quins paràmetres ens calen per a arribar a l'objectiu proposat pel subproblema.

Amb els paràmetres...

... les accions o funcions es poden veure com capses negres que interactuen amb l'exterior a partir d'una entrada de dades i generen dades a la sortida. Allò que hi ha dins la capsa no importa quan la fem servir. A més, la podem moure a qualsevol lloc perquè no depèn de res més extern que dels seus propis paràmetres per al seu funcionament intern.

1.7. Aprofundiment en l'exemple

Ara resoldrem el nivell següent de descomposició del problema. En aquest nivell ens enfrontem a dues abstraccions: les frases i el resultats de l'anàlisi del text que hem conceptualitzat com a objectes del problema resolt en el primer nivell.

Comencem pels resultats de l'anàlisi del text. Per a saber el percentatge de frases que s'han de corregir en el text ens cal conèixer quantes frases hi ha , i quantes d'aquestes cal corregir. Necessitarem, doncs, dos comptadors (*nFrases*, *nFrasesCor*, respectivament) per a calcular el percentatge quan hàgim acabat de processar el text.

Per a conèixer la mitjana de mots per frase corregida necessitarem un comptador de les paraules que s'han de corregir, que juntament amb el nombre de frases que s'han de corregir, ens proporcionaran la mitjana. També necessitarem enregistrar el màxim nombre de paraules que s'han de corregir per frase. Encapsulem les dades:

```
tDadesAnalisi = tupla
                    nFrases: enter;
                    nFrasesCor: enter;
                    nMotsPerCorregir: enter;
                    maximMotsPerCorregir: enter;
                ftupla
```

El disseny de les accions, d'acord amb les especificacions que perseguim és el següent:

```
accio iniciResultatsAnalisi(sor A: tDadesAnalisi)
  A.nFrases := 0;
  A.nFrasesCor := 0;
  A.nMotsPerCorregir := 0;
  A.maximMotsPerCorregir := 0;
faccio

accio actualitzaResultatsAnalisi(ent F: tFrase; entsor A: tDadesAnalisi)
  var
    nMotsPerCorFrase: enter;
  fvar
  A.nFrases := A.nFrases + 1;
  nMotsPerCorFrase := motsPerCorregir(F);
  si nMotsPerCorFrase > 0 llavors
    A.nFrasesCor := A.nFrasesCor + 1;
    A.nMotsPerCorregir := A.nMotsPerCorregir + nMotsPerCorFrase;
  fsi
```

```
    si nMotsPerCorFrase > A.maximMotsPerCorregir llavors
      A.maximMotsPerCorregir := nMotsPerCorFrase;
    fsi
  faccio

  accio escriureResultatsAnalisi(ent A: tDadesAnalisi)
    si A.nFrasesCor > 0 llavors
      escriureReal(enterAReal(A.nFrasesCor) / enterAReal(A.nFrases) * 100.0));
      escriureReal(enterAReal(A.nMotsPerCorregir)/enterAReal(A.nFrasesCor));
      escriureEnter(A.maximMotsPerCorregir);
    fsi
  faccio
```

Observeu que en la darrera acció hem de preveure l'error de dividir per 0. De fet, l'enunciat demana que no es generi sortida si no hi ha cap error en el text.

En l'acció *actualitzaResultatsAnalisi* s'usa la funció *motsPerCorregir* que afegirem al grup d'accions i/o funcions del tipus *tFrase*. Tot i que ja havíem avançat per raons d'exposició, el tipus *tFrase*, la funció ens és útil per a mantenir la independència entre les dues abstraccions: per a resoldre els resultats de l'anàlisi del text no ens cal els detalls de com es representa una frase, i d'aquesta manera, resolem l'abstracció de frase amb independència d'allò que s'ha fet en l'abstracció de dades d'anàlisi. Si fem modificacions a *tFrase* (amb els respectius canvis que cal fer en el seu conjunt d'accions i funcions), no haurem d'examinar el codi referent a l'abstracció de dades d'anàlisi. L'especificació de la funció serà la següent:

```
funcio motsPerCorregir(f: tFrase): enter
  { Pre: f = F }
  { Post: motsPerCorregir(F) ens dóna el nombre de mots que s'han de corregir
    en la frase F }
```

Resolem ara el segon nivell de descomposició: l'abstracció de frases. Començarem pel subproblema d'obtenir una frase. La millor manera de plantejar l'obtenció és considerar que, a l'entrada, la frase que s'ha d'obtenir és una seqüència de mots:

$$m_1\ m_2\ m_3\ m_4\ ...$$

Decidim, doncs, de fer l'abstracció de mot per a resoldre aquest nivell. Per tant, expressarem totes les solucions d'aquest nivell en termes de mots, si cal, i més endavant ja treballarem els detalls dels mots en una altra etapa.

En reconèixer una seqüència del problema, sigui aquesta real o abstracta, cal identificar el primer element, com s'obté el següent i quin element marca el final de la seqüència o és el darrer. Si repassem l'enunciat, observem que no hi ha cap diferència entre obtenir el primer mot i l'obtenció del següent, i que la propietat que identifica el darrer mot és que finalitza amb un punt. En el nostre cas, el manteniment de l'abstracció de mot converteix en abstraccions de codi la identificació de les operacions en la seqüència, com veurem a continuació.

Reconeixem l'obtenció d'una frase com un problema de recorregut, que consisteix a comptar el nombre de mots que s'han de corregir i ens porta a l'esborrany següent, que s'ha de completar:

```
accio obtenirFrase(sor f: tFrase)
   var m: tMot; fvar
   "Iniciar el tractament"
   obtenirPrimerMot(m);
   mentre no darrerMot(m) fer
      "Tractar l'element" { tractament del mot m }
      obtenirMot(m);
   fmentre
   "Tractament final"
faccio
```

Observem també que el darrer mot no sols marca el final de la frase, sinó que és un mot vàlid que es pot tractar igual que els altres. Aquesta possibilitat no està prevista en l'esquema de recorregut, que haurem de modificar convenientment perquè es tingui en compte.

Hem de preveure la possibilitat que la frase estigui buida, és a dir, que només contingui el punt. En aquest cas, haurem de comptar amb el mot buit: un mot que no té cap caràcter i acaba en punt.

El tractament que s'ha d'aplicar per cada mot consisteix a saber si cal corregir el mot al qual s'ha accedit o no, i en el cas que així sigui, comptabilitzar-lo per a saber quants mots cal corregir a la frase. D'altra banda, hem de comptar el nombre de mots que hi ha a la frase per a saber si aquesta és buida o no. Cal recordar que la frase buida (i, per tant, que conté un mot buit) representa el final del text i no forma part del text. Òbviament el mot buit no és un mot real de la frase i no cal comptar-lo. En conseqüència, haurem de distingir si el darrer mot és buit o no per a fer-li un tractament diferent. La solució resulta així:

```
accio obtenirFrase(sor f: tFrase)
   var m: tMot; nMots: enter; fvar

   { Pre: a l'entrada tenim un text T representat com una seqüència de frases t.
     A la part dreta de t, DT, hi ha una seqüència de frases o la frase buida }

   nMots := 0;
   f.nMotsC := 0;
   obtenirMot(m);
   mentre no darrerMot(m) fer
      si calCorregirMot(m) llavors f.nMotsC := f.nMotsC + 1; fsi
      nMots := nMots + 1;
      obtenirMot(m);
   fmentre
   si no motBuit(m) llavors
      si calCorregirMot(m) llavors f.nMotsC: = f.nMotsC + 1; fsi
      nMots := nMots + 1;
   fsi
   f. final := nMots = 0;
   { Post: a f tenim representada la primera frase de DT.
     La frase representada és a la part esquerra de t }
faccio
```

Fixeu-vos que...

... en la darrera estructura condicional (**si** … **fsi**) tractem el mot que actua com a marca.

La funció *calCorregirMot* ens ha de dir si el mot *m* ha de patir correccions o no. De moment, la feina que s'ha de fer en el nivell següent és aquesta:

```
accio obtenirMot(sort m: tMot)
   { Pre: a l'entrada tenim la seqüència de mots, t, del text T,
     de la qual només podem obtenir la part dreta DT i aquesta no ha arribat
     al final del text }
   { Post: m representa el primer mot que hi ha a DT.
     El mot obtingut estarà a la part esquerra de la seqüència t }

funcio darrerMot(m: tMot): boolea
   { Pre: m = M }
   { Post: darrerMot(M) és cert si m representa el darrer mot d'una frase }

funcio motBuit(m: tMot): boolea
   { Pre: m = M }
   { Post: motBuit(M) és cert si m representa un mot buit }

funcio calCorregirMot(m: tMot): boolea
   { Pre: m = M }
   { Post: calCorregirMot(M) és cert si el mot M s'ha de corregir.
     calCorregirMot(M) és fals si no cal corregir M }
```

La resta de subproblemes de l'abstracció de frase són evidents, i a continuació es mostren les solucions respectives:

```
funcio darreraFrase(f: tFrase): boolea
retorna f.final;
ffuncio
   { Pre: f = F que és una frase vàlida }
   { Post: darreraFrase(F) és cert si F és una frase buida. Retornarà fals en cas contrari }

funcio motsPerCorregir(f: tFrase): enter
retorna f.nMotsC;
ffuncio
{ Pre: f = F }
{ Post: motsPerCorregir(F) dóna el nombre de mots
  que s'han de corregir per a la frase F }
```

Us pot semblar exagerat definir aquestes funcions tan breus, però l'avantatge és clar: com que el primer nivell no es preocupa dels detalls de *tFrase*, no s'ha de modificar mai si en el segon nivell escollim una altra representació per a *tFrase* i dissenyem les accions i funcions d'acord amb la nova representació. Només hem canviat el nivell on concretem l'abstracció sense canviar els altres nivells, on hi ha les altres abstraccions sense concretar. Per tant, les possibles modificacions que s'han de fer en l'algorisme passen només en l'abstracció o les abstraccions de les quals ens hem proposat canviar la representació.

Passem ara al tercer nivell corresponent a l'abstracció de mot. Podem considerar un mot com una seqüència de caràcters acabat en blanc, excepte el darrer mot d'una frase, que acaba amb un punt abans d'arribar a l'espai. En aquest nivell, doncs, expressem el mot en termes de caràcters, que afortunadament el nostre llenguatge algorísmic entén; per tant, es tracta del darrer nivell d'abstracció, ja que el disseny serà prou concret.

Fixeu-vos que...

... expressar les solucions que tracten un mot en termes de caràcters és convenient per a aquest problema concret, però això no sempre és cert. Per exemple, si haguéssim de comptar les síl·labes d'una paraula o mot, pot ser convenient de veure el mot com una seqüència de síl·labes abans que de caràcters.

Aquí, ens interessa saber si cal corregir el mot o no. Per tant, hem de detectar la presència del caràcter "*" un cop per a saber si el mot s'ha de corregir o no. El fet que el mot consti només del caràcter "*" s'ha d'interpretar com que cal afegir un mot per a corregir la frase. D'altra banda, també ens interessa saber si es tracta d'un mot final vàlid o és buit. Per tant:

```
tipus
   tMot = tupla
             final: boolea;
             buit: boolea;
             calCorregir: boolea;
          ftupla
ftipus

{ Pre: a l'entrada tenim la seqüència de mots, t, del text T,
  de la qual només podem obtenir la part dreta DT
  i aquesta no ha arribat al final del text }

accio obtenirMot(sor m: tMot)
   var
      c: caracter;
      caractersLlegits: enter;
   fvar
   caractersLlegits := 0;
   c := llegirCaracter();
   mentre c ≠ '*' i c ≠ ' ' i c ≠ '.' fer { cerquem l'asterisc }
      caractersLlegits := caractersLlegits + 1;
      c := llegirCaracter();
   fmentre
   m.calCorregir := c = '*';
   mentre c ≠ ' ' i c ≠ '.' fer { passem la resta de la paraula }
      caractersLlegits := caractersLlegits + 1;
      c := llegirCaracter();
   fmentre
   m.buit := caractersLlegits = 0;
   m.final := c = '.';
faccio

{ Post: m representa el primer mot que hi ha a DT.
  El mot obtingut estarà en la part esquerra de la seqüència t }
```

La resta de funcions són òbvies:

```
funcio darrerMot(m: tMot): boolea
retorna m.final o m.buit;
ffuncio

funcio motBuit(m: tMot): boolea
retorna m.buit;
ffuncio

funcio calCorregirMot(m: tMot): boolea
retorna m.calCorregir;
ffuncio
```

Amb això arribem al final de la solució de l'exercici. Pareu esment a la quantitat de detalls que han calgut per a arribar a la solució. L'anàlisi ha estat complicada, sobretot en l'etapa en què treballàvem amb frases com a seqüències de mots, amb tants detalls com calia tenir en compte. Però la separació dels problemes ens ha ajudat a concentrar-nos-hi més còmodament que si haguéssim tingut en compte tots els detalls del problema alhora.

1.8. Valoració final de la metodologia

Realment és tant útil el mètode proposat? Si haguéssim pensat el problema només en termes de caràcters, sense utilitzar cap abstracció ni separació de problemes, podríem haver arribat a una solució com aquesta:

```
algorisme nyap

  var
    c: caracter;
    N1, N2, N3, N4, Mx: enter;
    FF, A, DP: boolea;
  fvar

  N1: = 0;
  Mx: = 0;
  N2: = 0;
  N3: = 0;
  N4: = 0;
  FF: = cert;
  A := fals;
  c: = llegirCaracter();
  mentre (no FF) o (c ≠ '.') fer
    si (c ≠ ' ') llavors
      FF := fals;
    si no (A) llavors A := c = '*'; fsi
    c := llegirCaracter();
    si (c = ' ') llavors
      si A llavors
        N2 := N2 + 1;
        A := fals;
      fsi
      DP := c= '.';
      c := llegirCaracter();
    sino
      DP := c = '.';
      c := llegirCaracter();
    fsi
    si DP llavors
      FF := cert;
      N3 := N3 + 1;
      si N2 > 0 llavors N4 := N4 + 1; fsi
      N1 := N1 + N2;
      si N2 > Mx llavors Mx := N2; fsi
      N2 := 0;
    fsi
  fmentre
  si N3 > 0 llavors
    escriureReal((enterAReal(N4) / enterAReal(N3) * 100.0));
    escriureReal((enterAReal(N1) / enterAReal(N4));
    escriureEnter(Mx);
  fsi
falgorisme
```

Ara, la discussió és si val la pena d'arribar a aquest tipus de solució. Pregunteu-vos el següent:

- Us és fácil de veure si aquesta solució funciona sense provar-la a l'ordinador, i fins i tot així?
- Creieu que, com que és un text més curt, ha de ser més eficient quan es processi?
- Creieu que, com que és un text més curt, han calgut menys hores de desenvolupament?
- Creieu que amb molts comentaris seria més intel·ligible que la solució anterior basada en el mètode que hem proposat?
- Creieu que és fàcil de modificar, mantenir, o entendre per algú o per vosaltres mateixos després d'un temps?

Si teniu en compte el que s'ha dit al llarg de tota l'assignatura, aquest grapat de preguntes tenen resposta única: no.

Hi ha tants detalls en l'algorisme que el fan molt poc pràctic per a convèncernos del fet que funciona. Cal estar molt concentrats i atents per a estar-ne segurs: avança bé la seqüència?, acabarà bé?, estan bé les inicialitzacions?, etc. Qui proposi un algorisme així, possiblement l'ha provat en l'ordinador, l'ha modificat contínuament a partir d'una versió inicial fins que algunes proves li donen la confiança que és correcte, però no n'està segur del tot. El dubte és saber quina prova li falta. Se n'han fet lectures i relectures (entre d'altres coses, podria ser més explícit en el nom de les variables, per exemple) per a intentar de veure'l parcialment a partir d'algunes abstraccions que no s'expressen.

A més,...

... penseu que és més fàcil de fer proves si el problema està descompost en accions i/o funcions perquè puguem comprovar quina tasca fa cada funció.

No obstant això, la confusió pot existir pel fet que les abstraccions no estan separades. Quan més endavant (una setmana o més) calgui revisar-lo pel que sigui, s'ha de tornar a fer aquest exercici olímpic de concentració. Si s'ha comentat àmpliament, de ben segur que trobareu que alguns comentaris despisten, perquè no s'ha seguit un ordre concret i no s'han separat ni mantingut les abstraccions que en el seu moment es van fer per a entendre l'algorisme. A més l'algorisme "nyap" amb comentaris podria ser tan llarg com el disseny fet metodològicament. La majoria de cops, un desenvolupament d'aquest tipus és molt lent, malgrat que el text final de l'algorisme sigui relativament curt.

A vegades, la intuïció ens fa males passades. La llargada del text en què s'expressa un algorisme no té correlació amb la possible eficiència en temps que pugui tenir l'algorisme quan es processa. El cost d'introduir encapsulació (accions i/o funcions, constructors, etc.) és mínim i negligible. Recordeu que un altre dels objectius dels algorismes és que es puguin llegir fàcilment, ja que el cost de manteniment i millores en una empresa es pot disparar si altres programadors han de perdre el temps llegint algorismes confusos de programadors no metòdics. Per exemple, l'editorial ens acaba d'avisar que s'ha introduït un caràcter corrector nou, el "/", que serveix per a indicar que les paraules que el continguin no es comptin com a paraules que s'han de corregir, malgrat que tinguin asterisc. On fareu els canvis més còmodament? En l'algorisme "nyap", o en el que hem desenvolupat pel mètode de disseny descendent?

Al llarg de tot el subapartat hem treballat un exemple. Una aplicació real presenta una complexitat molt més gran. En casos reals, totes les observacions fetes són encara més evidents i encara es fa més necessari treballar amb la metodologia aquí proposada.

1.9. Darreres observacions

Per a assimilar la metodologia cal practicar força i vèncer les pors inicials. En cada exercici que feu heu de potenciar la vostra capacitat intel·lectual per a trobar les abstraccions més convenients al problema, i també heu de potenciar la

vostra capacitat d'expressió (l'elecció d'un bons noms en les definicions també forma part del joc). La resta és verificar que la composició de sentències i expressions de l'algorisme, accions i/o funcions que dissenyeu segueixin la lògica correcta i prevista amb l'aplicació de les tècniques vistes al llarg de l'assignatura.

Resum

En aquest darrer mòdul hem introduït la metodologia de disseny descendent per refinaments successius que ens permet dissenyar algorismes que resolen problemes més complexos i propers a la realitat.

Com hem vist, la base d'aplicació d'aquesta metodologia consisteix a descompondre el problema complex, mitjançant l'abstracció, en subproblemes independents i més senzills de resoldre.

Ara, per tant, podem construir adequadament algorismes que resolguin problemes complexos. Ja tenim tots els recursos necessaris per a fer-ho: d'una banda, tenim les eines –el llenguatge algorísmic– i, de l'altra, la metodologia que ens ensenya com s'han de fer servir.

Tan bon punt aconseguim prou destresa en l'aplicació de la metodologia i l'ús de les eines del llenguatge algorísmic, haurem assolit un dels objectius bàsics que ens hem proposat al principi de l'assignatura: aprendre a dissenyar i implementar algorismes correctes, intel·ligibles, eficients i generals que resolguin els problemes plantejats, o dit d'una altra manera, aprendre a programar de manera professional, és a dir, amb eficiència –en un temps raonable– i eficàcia –que el resultat respongui als objectius demanats. A mesura que resolem problemes adquirim aquesta destresa.

A tall de resum,...

... atès el caràcter incremental de la matèria, les pautes que us recomanem aquí són alhora una síntesi de tot allò que hem après al llarg dels mòduls i, per tant, un bon resum final.

Cada cop que fem front al disseny d'un algorisme que ha de resoldre un problema, hem de seguir les pautes recomanades i que ara recordem: entendre l'enunciat, plantejar la solució, formular la solució i, finalment, avaluar-la.

1) Entendre l'enunciat

Llegirem l'enunciat d'un problema per primera vegada. Reconeixerem si cal aplicar-hi la metodologia de disseny descendent quan els objectes del problema o els objectius que l'enunciat proposa assolir no es poden expressar fàcilment en termes de tipus, accions i funcions elementals de la notació algorísmica, probablement a causa d'una quantitat excessiva de detalls que s'han de tenir en compte. Si aquest és el cas, buscarem les abstraccions que fan més tractable aquest enunciat tan ple de detalls, i a partir d'aquí, tindrem presents aquestes abstraccions, i rellegirem el problema sense tenir-ne en compte els detalls, que ja es consideren en nivells de descomposició posteriors.

Recordeu que...

… quan especifiquem només pensem en "què" cal fer, no en "com" cal fer-ho.

Com ja sabem, d'entrada hem de tenir clar què volem aconseguir. Altrament és evident que no trobarem la solució del problema. Hem d'interpretar l'enunciat correctament i definir amb claredat i sense ambigüitats què cal fer. Per això, es-

pecificarem l'algorisme que s'ha de dissenyar. Si cal utilitzar abstraccions, l'especificació s'ha d'expressar en termes d'aquestes abstraccions. En altres nivells de descomposició més concrets, ja s'especificaran els detalls que s'hi amaguen, si és necessari.

L'especificació ens diu quin és el problema que s'ha de resoldre; per tant, cal que especifiquem la precondició i la postcondició de cada algorisme, acció i funció que vulguem dissenyar.

Malgrat que l'especificació es fa en llenguatge natural, hem de tenir cura de fer-la amb precisió.

La **precondició** descriu l'estat inicial en què es troba l'entorn i del qual partim. La **postcondició** descriu l'estat final de l'entorn al qual hem d'arribar després que l'algorisme resolgui el problema.

2) Plantejar el problema

L'etapa de plantejament comença un cop tenim establertes la precondició i la postcondició d'un algorisme, acció o funció. L'especificació ha de correspondre al problema en si, o a una part del problema (subproblema). És ara quan s'ha de decidir com cal resoldre el problema i, per tant, per on cal seguir i aplicar gran part de la metodologia que coneixem.

En primer lloc, hem de veure si podem resoldre el problema directament amb facilitat, és a dir, si per a resoldre'l podem aplicar directament les estructures i esquemes que coneixem sobre les dades reals del problema. En aquest cas, ens trobem davant un problema concret i relativament senzill de resoldre. Malgrat això, si el problema correspon a un que segueix una seqüència, per a fer-ho amb unes certes garanties de correctesa, caldrà seguir les pautes que us hem proposat i que ara sintetitzem:

- Reconèixer la seqüència que s'ha de tractar. En alguns casos serà senzill, perquè treballarem explícitament amb la seqüència que segueix les dades de l'enunciat. En altres casos haurem de ser capaços de reconèixer que, per a resoldre el problema, cal generar una seqüència, que es concretarà amb la identificació del primer element i del l'element següent i de la 'marca' que determina el final de la seqüència.

Recordeu que...

… la seqüència pot venir donada per l'enunciat del problema mateix, o bé, pot ser que els objectius del problema ens demanin explícitament que la generem.

- Decidir l'esquema que s'aplicarà. Cal veure quin dels esquemes que coneixem (cerca i recorregut) ens ajudaran a resoldre el problema.

- Fer les adaptacions que convinguin en cada cas per a aplicar l'esquema escollit al problema concret.

Cal anar amb molta cura a l'hora d'aplicar l'esquema escollit. Per tal d'assegurar-nos-en, és convenient de reflexionar sobre les qüestions següents:

a) Com s'obtenen els diferents elements de la seqüència:
- Com s'obté el primer element.
- Com s'obtenen els elements següents.
- Com es detecta el final de la seqüència.

b) Com fem el tractament:
- Identificar el tractament que s'ha de fer per a cada element.
- Definir les variables necessàries.

  Sovint ens cal definir variables per a acumular els resultats que obtenim a mesura que tractem els elements; llavors cal inicialitzar-les de manera convenient en el tractament inicial.

- Decidir si és necessari de treballar amb taules.

  Si podem fer el tractament seqüencialment, llavors no definirem taules per a emmagatzemar (parcialment o totalment) les dades de la seqüència, ja que això ens limitaria les dades que s'han de tractar i ocuparíem espai innecessari. En cas, però, que ens calgui fer algun accés directe sobre (algunes o totes) les dades, ho haurem de definir convenientment.

- Determinar si cal tractar el darrer element.

  A vegades el darrer element és un element no vàlid que només serveix de marca per a detectar el final de la seqüència. Però, en d'altres casos, és un element que, a més de ser la marca final, és alhora un element vàlid de la seqüència i, per tant, cal anar en compte i aplicar-li el mateix tractament que a la resta d'elements. Llavors, cal tenir cura de fer els canvis adequats perquè l'esquema contingui aquest tractament.

- Decidir si cal fer un tractament final.

  Cal veure si encara cal un tractament final per tal d'arribar a la postcondició un cop ja s'han tractat tots els elements de la seqüència.

En el cas que el problema no segueixi cap seqüència, no podem aplicar els esquemes de recorregut i cerca, i atès que no tenim cap esquema previ per a aplicar, la resolució del problema consistirà a descobrir i dissenyar les accions i funcions necessàries que, combinades convenientment, assoleixin l'objectiu final. En aquests casos, cada acció o funció (elemental o no) resol un subobjectiu i, compostes de manera adient, han de resoldre l'objectiu final de l'algorisme.

Les pautes que s'han exposat fins ara també es poden aplicar a problemes complexos no expressables directament amb els elements de la notació algorísmica. Ampliem, ara, les indicacions que s'han de seguir i que ens serviran per a plantejar aquests tipus de problemes més complexos.

Recordeu que...

... els esquemes algorísmics no imposen cap restricció respecte al tipus dels elements de la seqüència i, per tant, es poden aplicar a seqüències de tipus no elementals definits a partir de tuples i taules, per a resoldre problemes més complexos.

Quan ens trobem davant un problema complex, cal trobar abstraccions convenients que simplifiquin els detalls que s'han de tenir en compte en el plantejament del problema i que ens permetin resoldre amb comoditat i prou claredat el problema (per exemple, que faciliti l'aplicació dels esquemes). Un cop hem escollit aquestes abstraccions, cal mantenir-les al llarg del plantejament del problema, és a dir, no s'ha d'entrar en detalls que corresponguin a un altre nivell del disseny descendent.

Fem una abstracció de dades.

Seguint la metodologia del disseny descendent, identificarem la seqüència d'elements abstractes. L'aplicació de l'esquema escollit ens portarà a definir unes accions i/o funcions correponents a les tasques d'obtenir el primer element, l'element següent i identificar la marca, en les quals encapsulem (amaguem) els detalls concrets amb tipus no elementals. No ens ha de representar cap problema suposar que disposem d'un tipus no elemental i d'un conjunt d'accions i funcions que d'entrada no existeixen perquè sempre els podem dissenyar més endavant.

Fem una abstracció de codi.

Quan a l'etapa de plantejament es formula un esquema sobre les dades abstractes, s'hi introdueixen tantes accions, funcions i tipus nous com calgui. D'aquesta manera, l'aplicació de la metodologia del disseny descendent condueix a la creació de nous enunciats de subproblemes (tipus, accions i funcions) que s'han de plantejar en el seu nivell corresponent (i, per tant, haurem d'entendre l'enunciat de cadascun per a plantejar-lo, formular-lo i avaluar-lo, i si el subproblema no és prou concret, donarà lloc a altres subproblemes que s'hauran de resoldre). Seguirem aquest procés successivament fins que sigui possible d'aplicar els esquemes directament sobre les dades reals del problema.

Per tant, en cada nivell més concret del disseny descendent hem de fer primer un plantejament del nivell i, a continuació, el plantejament-formulació-avaluació de cada un dels subproblemes que s'han de resoldre en el nivell de treball.

1) Per a cada abstracció corresponent al nivell de descomposició, cal representar els tipus que s'utilitzaran amb detall i que encara no s'han definit.

2) Per a cada acció i/o funció ja especificada (subproblema) que s'hagi de desenvolupar en aquest nivell cal:

- Reconèixer, si escau, la seqüència que s'ha de tractar. En aquests casos és més complicat, ja que cal identificar les dades abstractes adequades que formen la seqüència.

- Decidir l'esquema que s'aplicarà sobre la seqüència abstracta. Com sempre, cal veure quin dels esquemes que coneixem –cerca i recorregut– ens ajudarà a resoldre el problema.

- Fer les variacions que convinguin en cada cas per a aplicar l'esquema escollit al problema concret. L'aplicació s'ha de fer respectant les abstraccions, en cas que n'hi hagi, que corresponen a un nivell més concret.

- Especificar les accions i funcions auxiliars que hagin sorgit en el pas anterior.

El punt clau en l'abstracció de codi és determinar quins són els nivells d'abstracció adequats. Normalment, els objectes de què tracta el problema són el criteri que seguim a l'hora de determinar els nivells i descompondre el problema.

Fixeu-vos que...

... normalment fem una descomposició orientada a l'objecte.

En cada nivell de descomposició expressem els algorismes en funció d'uns tipus, accions i funcions que desenvolupem en el nivell següent. L'abstracció que hem fet en cada nivell permet que ens concentrem només en la resolució d'un subproblema concret; no ens preocupem de com es dissenyen els tipus, accions i funcions que ens calen, simplement sabem què fan i els utilitzem, i més endavant, en uns altres nivells, ja els dissenyarem.

La descomposició en nivells...

... també es pot veure com si tinguéssim un processador diferent per a cada un dels nivells que sigui capaç d'entendre tots els tipus, accions i funcions que encara s'han de desenvolupar.

Recordeu que cal rellegir l'enunciat a cada nivell de descomposició de disseny per a copsar els detalls que hi són rellevants.

Fixeu-vos que en tota aquesta exposició de la tipologia de problemes que podem resoldre en aquesta assignatura, encara faltaria afegir un cas. En plantejar un problema, ens podem trobar en un cas en què no reconeguem o no identifiquem cap seqüència que ens convingui. Aleshores, cal descobrir l'ordre de les accions que poden portar a terme l'objectiu que l'algorisme ha de complir.

Però, a vegades, el nombre d'accions elementals proposades pot ser excessiu i llavors convé de reconèixer les diverses etapes que assoleixen subobjectius parcials i que, quan es componen, proporcionen l'objectiu final. Cadascuna d'aquestes etapes s'encapsula mitjançant accions i funcions. D'aquesta manera, obtenim llegibilitat i intel·ligibilitat en les solucions formulades. L'orientació de la descomposició és funcional. Fixeu-vos que això que acabem de comentar és aplicable a subproblemes provinents d'una descomposició orientada a l'objecte (per exemple, el tractament d'un element abstracte d'una seqüència).

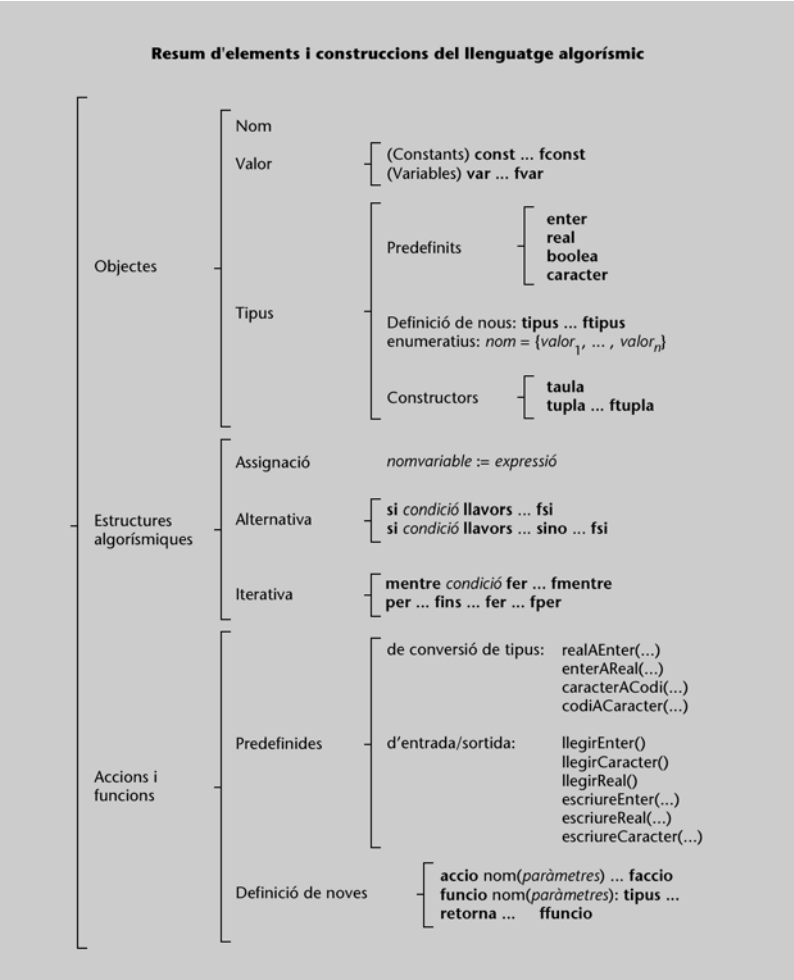
3) Formular la solució

Un cop sabem com volem resoldre el problema, només ens cal formular la solució en la notació algorísmica que hem definit. Així podrem expressar l'algorisme amb precisió i rigor.

Recordeu que...

... per tal d'evitar ambigüitats, quan formuleu els vostres algorismes, heu de seguir estrictament la notació algorísmica que hem definit.

Ja coneixem la sintaxi i semàntica de cadascun dels elements i les construccions del llenguatge algorísmic que trobareu sintetitzades a continuació:



4) Avaluar la solució

Ara per ara, no hem introduït cap mètode formal per a verificar la correctesa d'un algorisme, de manera que ho hem de fer informalment a partir de raonar la semàntica dels elements i les construccions del llenguatge algorísmic emprades i comprovar pas a pas que els hem anat component de manera adequada per a aconseguir la postcondició desitjada.

Els comentaris inserits durant el disseny ens han d'ajudar a verificar la correctesa de l'algorisme.

Ara bé, l'ús de la metodologia de disseny i l'aplicació estricta dels esquemes de recorregut i cerca ens asseguren la correctesa dels tractaments seqüencials. Recordeu que un esquema és una mena de plantilla de la qual en coneixem molt bé el funcionament per a solucionar un tipus de problemes concret i, per tant, si l'apliquem correctament, ens portarà a resultats correctes.

L'ús combinat del disseny descendent i dels mecanismes d'abstracció no sols beneficia el disseny, sinó que també afavoreix altres etapes del desenvolupament de programes:

Recordeu que com ja sabeu del mòdul "Introducció a la programació", les etapes de desenvolupament d'un programari són: l'anàlisi de requeriments, el disseny, la implementació de les proves i el manteniment.

- El procés de desenvolupament: gràcies al fet de parcel·lar la complexitat dels problemes i de descompondre'ls successivament en trossos més petits, en cada moment es treballa la resolució d'un únic problema de dimensions més manejables.

- La posada al punt: els trossos que componen els programes es poden provar per separat (cada acció i funció individualment) i a poc a poc.

- El manteniment: normalment, les modificacions posteriors que es facin durant la vida útil d'un programa tenen un impacte localitzat només en unes poques accions, la resta no se'n veuen afectades. Per tant, el fet de tenir el programa descompost, en facilita les modificacions. D'altra banda, el fet que els programes tinguin una estructura més clara fa que sigui més fàcil i segur modificar-los.

- La reutilització del codi: si desenvolupem programes que tracten amb objectes similars, segurament podrem aprofitar dissenys de tipus fets per a altres programes, i les funcions i accions que els tractin.

- La llegibilitat: els programes que resulten després d'aplicar les tècniques de disseny descendent i els esquemes estan estructurats i, per tant, són més fàcils de llegir i d'analitzar.

Exercicis d'autoavaluació

1. Dissenyeu un algorisme que esbrini quants cops apareix "LA" en una frase acabada amb punt i escrita amb lletres majúscules al dispositiu d'entrada.
Per exemple, a l'entrada tenim la frase següent:

LA BLANCA REGALA A LA LAIA UNA FLASSADA.

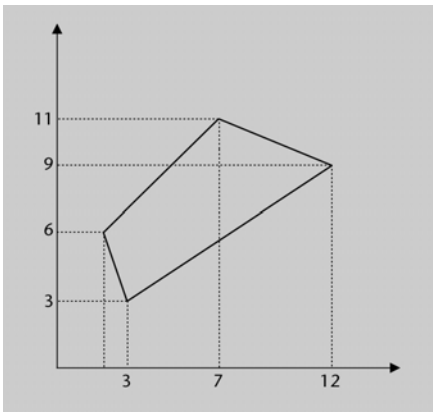
El programa ha de donar com a resultat: 6. Trobem "LA" en els casos següents:

LA[1] BLA[2]NCA REGALA[3] A LA[4] LA[5]IA UNA FLA[6]SSADA.

2. Donat un polígon especificat per una seqüència de vèrtexs en l'ordre del seguiment del seu contorn, dissenyeu un algorisme que en trobi l'àrea. La seqüència acaba amb el primer vèrtex amb què ha començat. A l'entrada hi ha, com a mínim, les dades d'un triangle. Cada vèrtex s'especifica per dues coordenades reals. Per exemple, si la seqüència és

3.0 3.0 2.0 6.0 7.0 11.0 12.0 9.0 3.0 3.0

El polígon que representa és el que reproduïm a continuació:



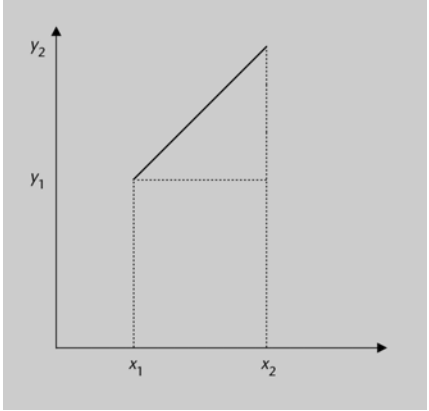
Un amic a qui agrada la geometria ens diu que el càlcul de l'àrea d'un polígon descrit pels seus vèrtexs amb el seguiment del contorn es pot fer mitjançant la fórmula següent:

$$\sum_{i=1}^{n} \frac{(x_{i+1} - x_i)\ (y_{i+1} + y_i)}{2}$$

en què n és el nombre d'arestes i, quan $i = n$, $i + 1$ és en realitat 1 (la manera correcta de fer-ho és amb el mòdul, però no el posem per a fer més clara la fórmula). En el nostre exemple, l'àrea A del polígon seria la següent:

$A = 1/2((2 - 3)(6 + 3) + (7 - 2)(11 + 6) + (12 - 7)(9 + 11) + (3 - 12)(3 + 9))$
$A = 1/2(-9 + 85 + 100 - 108) = 68/2 = 34$

Podria ser que el resultat fos negatiu, però si en canviem el signe, el valor de l'àrea continuarà essent el correcte. Disposem de la funció *abs*(*r*), que dóna el valor absolut d'un nombre real *r*.
De fet, cada terme del sumatori calcula l'àrea del trapezi que forma una aresta del polígon sobre l'eix *x*, tal com es veu en la figura següent:



Per als qui teniu curiositat pel tema –no és necessari que llegiu això per a dissenyar l'algorisme, ja que l'enunciat diu molt clar què s'ha de resoldre–, en la figura podeu veure que el trapezi està compost d'un triangle i un rectangle, i, aleshores, l'àrea A_t ha de ser la següent:

$$A_t = \underbrace{y_i\,(x_2 - x_1)}_{\text{Rectangle}} + \underbrace{\frac{(x_2 - x_1)\,(y_2 - y_1)}{2}}_{\text{Triangle}}$$

$$A_t = \frac{2y_i\,(x_2 - x_1) + (x_2 - x_1)\,(y_2 - y_1)}{2}$$

$$A_t = \frac{(x_2 - x_1)\,(y_2 + y_1)}{2}$$

Observem que l'àrea té signe positiu o negatiu segons si $x_2 > x_1$ o no, respectivament. Si els vèrtexs de les arestes d'un polígon es donen respectant el sentit del contorn (horari o antihorari), les àrees dels trapezis s'han de poder sumar algebraicament i donaran l'àrea neta del polígon un cop hàgim recorregut totes les arestes. Així, doncs, l'àrea del polígon s'ha de representar de la manera següent:

$$\sum_{i=1}^{n} \frac{(x_{i+1} - x_i)\,(y_{i+1} + y_i)}{2}$$

en què n és el nombre d'arestes i, quan i és igual a $n + 1$, $n + 1$ és 1.

3. Donada una frase acabada amb un punt, compteu el nombre de paraules capicues. Entre paraules hi ha un o més caràcters separadors com espais, comes o punts i comes. Cada paraula de la frase té com a molt 15 caràcters.

4. Una empresa de publicitat té un servidor d'Internet i vol facturar als seus clients les pàgines d'HTML instal·lades. Amb aquest propòsit, l'empresa vol que dissenyem un programa que indiqui el que cada client ha de pagar a partir del contingut de les pàgines instal·lades.

A l'entrada del programa hi haurà una seqüència de caràcters organitzada de la manera següent:

$$m_{11} \cdot m_{12} \cdot \ldots \cdot m_{1n_1} \cdot \mathit{FiTextHTML} \cdot m_{21} \cdot m_{22} \cdot \ldots \cdot m_{2n_2} \cdot \mathit{FiTextHTML} \cdot \ldots \cdot$$
$$\cdot \ldots \cdot m_{n1} \cdot m_{n2} \cdot \ldots \cdot m_{nn_n} \cdot \mathit{FiTextHTML} \cdot \mathit{FiPagines}$$

On m_{ij} són cadenes de caràcters sense separadors que poden tenir qualsevol nombre de caràcters. *FiTextHTML* és la paraula "FiTextHTML" i indica el final de la pàgina HTML; *FiPagines* indica el final de la seqüència i és la paraula "FiPagines" acabada amb un blanc. Es considera que un caràcter és separador si aquest és un espai (" "), un "=", o es tracta d'un control de línia (retorn de carro o salt de línia).
D'entre les paraules que hi ha a cada pàgina, n'interessa reconèixer la paraula "propietat", ja que a continuació d'aquesta apareixerà el nom del client que ha posat la pàgina i, per tant, a qui se li ha de facturar. Per a facturar la pàgina a l'empresa propietària, el programa haurà de tenir en compte el següent:

- Les paraules "propietat", "FiTextHTML" i "FiPagines" són òbviament gratuïtes, ja que només serveixen per a controlar la seqüència.
- Tota paraula en què el primer caràcter sigui "<" i el darrer ">" no comptarà, ja que es tracta de controls del llenguatge HTML que l'empresa considera gratuïts (només paraules, les frases entre els caràcters "<" i ">" no seran gratuïtes).
- Si es troba la paraula "<IMG", es facturaran 50 euros.
- Qualsevol altra paraula no inclosa en els punts anteriors i que tingui més d'un caràcter, es facturarà a 5 euros.

Observacions:

- No hi haurà més de 50 empreses.
- Una empresa pot tenir un nombre qualsevol de pàgines.

- La seqüència d'entrada estarà ben formada. En particular:
– Hi pot haver pàgines buides (S ≡ 'FiTextHTML FiTextHTML ... ').
– La seqüència pot ser buida (S ≡ 'FiPagines').
– Pot no existir propietari. Donat aquest cas, la pàgina es facturarà a nom d'una hipotètica empresa anomenada *NoIdentificats*.
- Ens interessa tractar només els primers 40 caràcters d'una paraula, com a molt.
- Les paraules de control delimitades per "<" i ">" no tindran més de 40 caràcters.

Un exemple de seqüència podria ser el següent:

```
<! propietat VagonsPopulars>
<HEAD>
<TITLE> La darrera innovació</TITLE>
</HEAD>
<BODY>
<meta name="description" value=" Especificació">
<meta name="keywords" value="disseny">
<meta name="resource-type" value="document">
<meta name="distribution" value="global">
<P>
<BR> <HR><A NAME=tex2html176 HREF="node16.html">
<IMG ALIGN=BOTTOM ALT="next"
SRC= "http://asdwww.cern.ch/icons/next_motif.gif">
</A> <A NAME=tex2html174 HREF="no
...
FiTextHTML
<HEAD>
<TITLE> Cursos de Postgrau</TITLE>
</HEAD>
<BODY>
<! propietat UOC >
...
<BGSOUND LaMarsellesa>
<FRAME Pals>
<SCRIPT javascript>
pots trobar la pàgina a
<A NAME=hola HREF = http://www.uoc.es/presentacio>
FiTextHTML


FiPagines
```

A la sortida tindrem:

```
                    VagonsPopulars 1130 UOC 315
```

Hi ha 42 paraules comptabilitzades a 5 euros per al cas de VagonsPopulars:

```
<! VagonsPopulars> La darrera innovació</TITLE> <meta name "description"
value Especificació"> <meta name "keywords" value "disseny"> <meta name
"resource-type" value "document"> <meta name "distribution" value "global">
<HR><A NAME tex2html176 HREF "node16.html"> <ALIGN BOTTOM ALT "next" SRC
"http://asdwww.cern.ch/icons/next_motif. <A NAME tex2html174 HREF "no ... .
```

I encara cal afegir 50 euros a l'import perquè hi ha un "<IMG". I les de la UOC són 21:

```
Cursos de Postgrau</TITLE> <! UOC... <BGSOUND LaMarsellesa> <FRAME Pals>
<SCRIPT javascript> pots trobar la pàgina a <A NAME hola HREF http://
www.uoc.es/presentacio>.
```

Solucionari

1. Hem de dissenyar un algorisme que esbrini quantes vegades apareix "LA" en una frase escrita en majúscules i acabada en punt i final.

Especificació

La frase es pot veure com una seqüència de caràcters acabada amb un punt. D'altra banda, necessitem un comptador que anomenarem *nbDeLA* per a indicar el nombre de *LA* que hi ha a la seqüència. Així, doncs:

nbDeLA: **enter**
{ Pre: $f = F$ i f és una frase acabada amb punt i f només conté majúscules }
comptarLA
{ Post: *nbDeLA* indica el nombre de "LA" que hi ha a F }

Plantejament general 1

Inicialment pot semblar que el problema es pugui pensar directament a partir de la seqüència de caràcters aplicant un esquema de recorregut. Però el fet d'haver de treballar caràcter per caràcter dificulta la nostra perspectiva del problema, ja que necessitarem veure dos caràcters alhora per a decidir si és una parella "LA" o no ho és.
Si ho pensem caràcter per caràcter, ens podem trobar amb els casos següents:
1) El caràcter llegit és diferent de "L" → Consultem el caràcter següent.
2) El caràcter llegit és igual a "L" → Hem de verificar que el caràcter següent sigui una "A".
Per al caràcter que ve a continuació, ens trobarem amb els casos següents:
a) És una "A" → Incrementem el comptador de "LA".
b) És una "L" → Hem de continuar verificant el caràcter següent a l'actual, ja que es podria donar el cas de, per exemple, "ELLA".
c) És diferent de "L" i "A" → No tenim en compte el caràcter actual, ja que no donarà "LA".

Solució 1

```
algorisme comptarLA
  var
    c: caracter;
    nbDeLA: enter;
  fvar
  { f = F i f és una frase acabada en punt i f només conté majúscules }
  nbDeLA := 0;
  c := llegirCaracter();
  mentre c ≠ '.' fer
    si c = 'L' llavors { el caràcter actual és una "L" }
      c: = llegirCaracter();
      si c = 'A' llavors { el caràcter anterior era una "L" i l'actual és una "A" }
        nbDeLA := nbDeLA + 1
        c := llegirCaracter();
      fsi
      { En cas que
        c = 'A' →  ja hem avançat i comptat.
        c = '.' →  millor no avançar.
        c = 'L' →  aprofitarem el primer condicional i, per tant, no avancem.
        c ≠ 'L' →  ja avançarem amb el sino del primer condicional }
    sino { c ≠ '.' }
      c := llegirCaracter();
    fsi
  fmentre
  { nbDeLA indica el nombre de "LA" que hi ha a F }
  escriureEnter(nbDeLA);
falgorisme
```

Avaluació 1

La correctesa de la solució presentada és molt difícil d'avaluar. Per exemple, hem de respondre qüestions del tipus: té en compte la seqüència buida, formada només pel punt?, té en

compte casos com "ELLA", la seqüència avança sempre, o es queda estancada en algun punt? Si seguiu bé l'algorisme us adonareu que totes les qüestions estan ben cobertes, encara que costa bastant de veure.

Plantejament general 2

El fet de pensar aquest problema en termes de caràcters ha creat un cert enrenou. En canvi, si convenim l'abstracció de la parella de caràcters, veurem que el problema és més fàcil d'analitzar i desenvolupar. El primer que hem de fer és imaginar-nos que tenim una seqüència de parelles de caràcters. Això ens porta a particularitzar l'esquema de recorregut tenint en compte l'abstracció convinguda:

1) Primer nivell

Solució

```
algorisme comptarLA
  var
    nbDeLA: enter;
    parella: tParella;
  fvar
  { f = F i f és una frase acabada en punt i f només conté majúscules }
  nbDeLA := 0;
  obtenirPrimeraParella(Parella);
  mentre no darreraParella(Parella) fer
    si esLA(parella) llavors nbDeLA := nbDeLA + 1; fsi
    obtenirParella(parella);
  fmentre
  { nbDeLA indica el nombre de "LA" que hi ha a F }
  escriureEnter(nbDeLA);
falgorisme
```

Així tenim una part de la solució expressada, i el nivell s'ha de completar amb l'especificació de les accions i funcions que resten per desenvolupar:

```
accio obtenirPrimeraParella(sor p: tParella)
  { Pre: s = S i s és una frase acabada en punt i la part esquerra de s és buida }
  { Post: p ha de contenir la primera parella de la seqüència de parelles
    i la part esquerra de s ha de contenir un caràcter }

accio obtenirParella(entsor p: tParella)
  { Pre: s = S i s és una frase acabada en punt i la part esquerra de f no és buida
    i la part dreta de s tampoc no és buida i p = P i P és l'anterior parella obtinguda }
  { Post: p ha de contenir la parella següent de la seqüència de parelles
    i la part esquerra de s ha de contenir un caràcter més }

funcio darreraParella(p: tParella): boolea
  { Pre: p = P }
  { Post: darreraParella(p) és cert si P és la darrera parella de la seqüència }

funcio esLA(P: tparella): boolea
  { Pre: p = P}
  { Post: esLA(P) és cert si P és la parella LA }
```

Fixeu-vos que per a verificar aquesta part no necessitem saber la definició del tipus *tParella,* i com que disposem de l'especificació de les accions i funcions, podem comprovar si la solució fins ara formulada és correcta.

2) Segon nivell

Plantejament

Per a continuar el disseny, cal definir la representació que ha de tenir una parella de caràcters. Una possibilitat és definir una tupla i l'altra és definir una taula. Escollirem una taula (una tupla també hauria estat una bona solució per a aquest problema).

```
tipus
   tParella = taula[2] de caracter;
ftipus
```

Ara hem de plantejar el problema de com fer que la seqüència de caràcters es pugui veure com la seqüència de parelles que ens convé. Vegem quina correspondència hi ha: la seqüència *f* està formada per $c_1c_2c_3c_4c_5 \ldots c_f$, on c_f és el caràcter final ".", i volem veure una seqüència de parelles $p_1, p_2 \ldots, p_f$, tal que $p_1 = c_1c_2$, $p_2 = c_2c_3$, ..., i p_f que tingui el punt com a segon caràcter.
Observem que es tracta de parelles de caràcters que s'encavalquen: cada parella que no sigui la primera comparteix un caràcter amb l'anterior.
La primera parella, entesa com els dos primers caràcters de la seqüència, (c_1c_2), exigiria que la seqüència de caràcters tingués com a mínim un caràcter diferent del punt: (c_1"."). Per tal d'ajustar-nos a la precondició (la seqüència té, com a mínim, el punt final) demanarem que el primer caràcter de la primera parella sigui un caràcter arbitrari que no formi part de la seqüència, i que el segon caràcter sigui el primer de la seqüència. Escollirem el caràcter arbitrari de manera que no ens espatlli el tractament (que no sigui pas una "L"). Per exemple, el caràcter espai. D'aquesta manera, la seqüència que ens convé és: $p_1 =$ " "c_1, $p_2 = c_1c_2$, i p_f que tingui el punt com a segon caràcter.
Aleshores passem a la solució en el segon nivell.

Solució

```
accio obtenirPrimeraParella(sor p: tParella)
   p[1] := ' ';
   p[2] := llegirCaracter();
faccio

accio obtenirParella(entsor p: tParella)
   p[1] := p[2];
   p[2] := llegirCaracter();
faccio

funcio darreraParella(p: tParella): boolea
retorna p[2] = '.';
ffuncio

funcio esLA(p: tParella): boolea
retorna p[1] = 'L' i p[2] = 'A';
ffuncio
```

2. Hem de dissenyar un algorisme que calculi l'àrea d'un polígon a partir de les coordenades dels seus vèrtexs.

Especificació

Especifiquem l'algorisme. Abreviarem les especificacions i anomenarem *pol* la seqüència d'entrada de dades.

```
var area: real; fvar
   { Pre: pol = POL i POL és una seqüència de reals
     que representa com a mínim un triangle }
   areaPoligon
   { Post: a area hi ha l'àrea del polígon representat en la seqüència POL }
```

1) Primer nivell

Plantejament

Cal identificar els objectes que siguin més convenients per a l'anàlisi. En el nostre cas, són els següents: arestes, vèrtexs, i reals.
Tot i que a l'entrada hi ha una seqüència de reals, en la primera etapa del disseny és convenient de fer la formulació de l'algorisme en termes d'arestes. Imaginem, doncs, que disposem del tipus *aresta* i que a l'entrada hi tenim una seqüència d'arestes. Les qüestions clàssiques de com s'obté el primer element, l'element següent i el darrer element estaran encapsulades mitjançant les accions i funcions que trobareu més endavant.

Si repassem l'enunciat, veiem que cada terme del sumatori és l'àrea del trapezi que es correspon amb cada aresta. El tractament que s'ha d'aplicar consisteix, doncs, a acumular aquestes àrees per a poder fer-ne el sumatori. Òbviament, això correspon a l'esquema de recorregut sobre una seqüència d'arestes. Però també haurem de tenir en compte algunes reflexions perquè, per exemple, haurem de modificar lleugerament l'esquema per a tractar la darrera aresta, que fa de marca de fi de seqüència.
Una altra observació és que podem treure factor comú de la divisió per dos del sumatori i posposar-la per al tractament final, que ha de consistir a dividir per dos les àrees acumulades i trobar-ne el valor absolut. Així, doncs, estalviarem alguns càlculs si calculem l'àrea doble per cada aresta. En la solució que teniu més endavant, el tractament final és la darrera assignació, però aquesta ja inclou el tractament de la darrera aresta.
Així, doncs, tenim tots els elements per a expressar una solució en termes de les arestes del polígon i les àrees dels trapezis corresponents sense entrar en detalls més concrets.

Solució

```
algorisme areaPoligon
   var area: real; a: aresta; fvar
   { Pre: pol = POL i POL és una seqüència de reals
     que representa com a mínim un triangle }
   obtenirPrimeraAresta(a);
   area := 0;
   mentre no darreraAresta(a) fer
      area := area + areaTrapezi(a);
      obtenirAresta(a);
   fmentre
   area := abs((area + areaTrapezi(A)) / 2.0))
   { Post: a area hi ha l'àrea del polígon representat en la seqüència POL }
falgorisme
```

Les accions i/o funcions pendents de desenvolupar són les següents:

```
accio obtenirPrimeraAresta(sor a: aresta)
   { Pre: a l'entrada hi ha una seqüència d'arestes pol i pol = POL
     i pol té com a mínim tres arestes a la part dreta i la part esquerra és buida }
   { Post: a serà la primera aresta de POL i a la part esquerra de Pol hi ha una aresta }

accio ObtenirAresta(entsor a: aresta)
   { Pre: pol = POL i la part dreta de pol, pDPol, no és buida i té com a mínim una aresta;
     la part esquerra de pol té com a mínim una aresta.
     a = A i A és la darrera aresta obtinguda }
   { Post: a té la primera aresta de pDPol
     i el valor de a és la darrera aresta de la part esquerra de pol }

funcio darreraAresta(a: aresta): boolea
   { Pre: a = A }
   { Post: darreraAresta(A) és cert si a és la darrera aresta de POL }

funcio areaTrapezi(a: aresta): real
   { Pre: a = A }
   { Post: areaTrapezi(A) és l'àrea doble del trapezi format per l'aresta A i l'eix x }
```

2) Segon nivell

Un cop estiguem convençuts del primer nivell, passem al següent, que consisteix a definir els tipus i les accions i/o funcions que han quedat pendents. En aquest nivell és convenient de veure la seqüència com una seqüència de vèrtexs.
Per al plantejament de l'obtenció de la seqüència de vèrtexs, cal observar que l'abstracció d'*aresta* està formada per dues parelles de vèrtexs encavalcades en la seqüència:

$$\overbrace{v_1,\ \underbrace{v_2,}_{a_2}}^{a_1}\ \overbrace{v_3,\ \underbrace{v_4,}_{a_4}}^{a_2}\ v_5\ ,\ldots$$

Així, doncs, en el plantejament aplicarem les mateixes idees (primer element, element següent i darrer element) que en la solució del problema anterior de comptar els "LA" i que aquí no repetirem. La darrera aresta mereix una atenció especial. A partir de l'enunciat podem veure que la darrera aresta es caracteritza perquè el seu extrem final v_f és idèntic al vèrtex

inicial de la primera aresta (v_1), que és, a més, el primer element de la seqüència. Amb aquestes observacions passem directament a la solució.
Quan definim el tipus *aresta* hem de tenir en compte que necessitarem els seus dos vèrtexs per tal de poder aplicar la fórmula de l'àrea del trapezi i també ens caldrà saber quin és el darrer vèrtex de la seqüència. Així, doncs:

```
tipus
   aresta = tupla vi, vf, dv: vertex; ftupla
ftipus

{ Pre: A l'entrada hi ha una seqüència d'arestes pol i pol = POL
  i pol té, com a mínim, tres arestes a la part dreta i la part esquerra és buida }
accio obtenirPrimeraAresta(sor A: aresta)
   obtenirPrimerVertex(A.vi);{ per la precondició sabem que
                               hi ha quatre vèrtexs com a mínim }
   obtenirVertex(A.vf);        { comptant el que marca el final }
   A.dv := A.vi;
faccio
{ Post: A serà la primera aresta de POL i a la part esquerra de pol hi ha una aresta }

{ Pre: pol = POL i la part dreta de pol, pDPol, no és buida i té com a mínim una aresta;
  la part esquerra de pol té com a mínim una aresta.
  a = A i A és l'anterior aresta obtinguda }
accio obtenirAresta(entsor a: aresta)
   a.vi := a.vf;
   obtenirVertex(a.vf);
faccio
{ Post: A té la primera aresta de pDPol
   i el valor de A és la darrera aresta de la part esquerra de pol }

{ Pre: a = A }
funcio darreraAresta(a: aresta): boolea
retorna vertexIguals(a.vf, a.dv);
ffuncio
{ Post: darreraAresta(A) és cert si a és la darrera aresta de POL }

{Pre: a = A }
funcio areaTrapezi(a: aresta): real
retorna areaTrapeziVertexs(a.vi, a.vf);
ffuncio
{ Post: areaTrapezi(A) és l'àrea doble del trapezi format per l'aresta A i l'eix x }
```

3) Tercer nivell

Un cop finalitzat el segon nivell, cal definir la seqüència de vèrtexs a partir de la seqüència de reals. En aquest cas, els vèrtexs són parelles de reals disjuntes:

$$\overbrace{x_1,\ y_1,}^{v_1}\ \overbrace{x_2,\ y_2,}^{v_2}\ \ldots$$

```
tipus
   vertex = tupla x, y: real; ftupla
ftipus

{ Pre: v1 = V1 i v2 = V2 }
funcio areaTrapeziVertexs(v1, v2: vertex): real
retorna (v2.x – v1.x) * (v2.y + v1.y);
ffuncio
{ Post: areaTrapeziVertexs(V1,V2) és l'àrea doble del trapezi format per l'aresta que va de
  V1 a V2 i l'eix x }

{ Pre: pol = POL i pol té com a mínim dos reals (un vèrtex) a la part dreta }
accio obtenirPrimerVertex(sor v: vertex)
   v.x := llegirReal();
   v.y := llegirReal();
faccio
{ Post: la part esquerra de pol té un vèrtex més i
  v té el vèrtex que abans era a la part dreta }
```

```
  { Pre: pol = POL i pol com a mínim té dos reals (un vèrtex) en la seva part dreta }
  accio obtenirVertex(sor v: vertex)
    v.x: = llegirReal();
    v.y: = llegirReal();
  faccio
  {Post: la part esquerra de pol té un vèrtex més i v té el vèrtex que abans era a la part dreta }
  { Pre: v1 = V1 i v2 = V2}
  funcio vertexsIguals(v1, v2: vertex): boolea
  retorna (v1.x = v2.x) i (v1.y = v2.y);
  ffuncio
  { vertexsIguals(v1,v2) és cert si V1 i V2 són el mateix vèrtex }
```

3.

Especificació

```
var nMotsC: enter; fvar
{ A l'entrada hi ha la seqüència de mots f de la frase F. F no és buida }
comptaCapicuesFrase
{ nMotsC té el nombre de mots capicues de F }
```

1) Primer nivell: frase

Plantejament

– Abstraccions utilitzades: mot.
– Identificació de la seqüència: frase vista com a seqüència de mots. L'obtenció dels mots primer i següent i el reconeixement de la marca s'encapsulen mitjançant accions i funcions.
– Identificació de l'esquema: recorregut. Cal tractar el mot que fa de marca de final.

Aplicació directa de l'algorisme

```
algorisme comptaCapicuesFrase
  var
    mot: tMot;
    nMotsC: enter;
  fvar
  nMotsC := 0;
  obtenirPrimerMot(mot);
  mentre no motFinal(mot) fer
    si motCapicua(mot) llavors nMotsC := nMotsC + 1; fsi
    obtenirMot(mot);
  fmentre
  si motCapicua(mot) llavors nMotsC := nMotsC + 1; fsi
  escriureEnter(nMotsC);
falgorisme
```

Accions pendents de desenvolupar

L'algorisme funcionarà si les accions i funcions que s'han de desenvolupar compleixen les especificacions següents:

```
accio obtenirPrimerMot(sor m: tMot)
{ Pre: A l'entrada hi ha la seqüència de mots f de la frase F.
  La part esquerra de f és buida. La part dreta, DF, no és buida }
{ Post: m representa el primer mot de DF.
  El mot obtingut estarà a la part esquerra de la seqüència f }

accio obtenirMot(entsor m: tMot)
{ Pre: A l'entrada hi ha la seqüència de mots f de la frase F.
  La part dreta, DF, no és buida. L'anterior mot llegit és a m }
{ Post: m representa el primer mot de DF.
  El mot obtingut estarà a la part esquerra de la seqüència f }

funcio motFinal(ent m: tMot): boolea
{ Pre: m = M }
{ Post: motFinal(M) és cert si m és el darrer mot d'una frase }
```

funcio motCapicua(**ent** m: tMot): **boolea**
{ Pre: *m* = *M* }
{ Post: *motCapicua*(*M*) és **cert** només si *m* és un mot capicua }

2) Segon nivell: mot

- Abstraccions utilitzades: cap.
- Identificació de la seqüència: mot vist com una seqüència de caràcters. L'obtenció dels mots primer i següent es farà mitjançant la funció predefinida *llegirCaracter*() i el reconeixement del caràcter que marca el final serà el caràcter espai o la coma, o el punt i coma, o el punt.

Definició del tipus

Necessitarem guardar els elements següents:

- El contingut del mot: la taula *c* i *long*, ja que la taula pot estar parcialment plena.
- L'últim caràcter llegit de la seqüència: *CaracterLlegit* serà sempre el caràcter del següent mot que es llegeix en el cas que no s'hagi arribat al final de la seqüència.
- Un indicador de final: *Final* serà **cert** si es tracta del darrer mot.

```
const maxCar: enter = 15; fconst
tipus
tMot = tupla
          long: enter;
          c: taula[maxCar] de caracter;
          caracterLlegit: caracter;
          final: boolea;
       ftupla
ftipus
```

Plantejament i solucions de les accions i funcions del nivell

Identificació esquema: recorregut.

```
accio obtenirPrimerMot(sor m: tMot)
  var c: caracter; fvar
  m.long := 0;
  c := llegirCaracter();
  mentre separador(c) i c ≠ '.' fer { saltem els primers separadors }
    c := llegirCaracter();
  fmentre
  m.caracterLlegit := c;
  obtenirMot(m);
faccio
```

Identificació de l'esquema: recorregut.

```
accio obtenirMot(entsor m: tMot)
  var c: caracter; fvar
  m.long := 0;
  c := m.caracterLlegit;
  mentre no separador(c) i c ≠ '.' fer { obtenim el mot }
    m.long := m.long + 1;
    m.c[m.long] := c;
    c := llegirCaracter();
  fmentre
  mentre separador(c) i c ≠ '.' fer { saltem els separadors }
    c := llegirCaracter();
  fmentre
  m.caracterLlegit := c;
  m.final := c = '.'
faccio
```

Aplicació directa de l'algorisme

```
funcio motFinal(m: tMot): boolea
retorna m.final;
ffuncio
```

Identificació de l'esquema: cerca.

```
funcio motCapicua(m: tMot): boolea
  var
    i, j: enter;
    trobat: boolea;
  fvar
  i := 1; j := m.long;
  trobat := fals;
  mentre (no trobat) i (i < j) fer
    si m.c[i] ≠ m.c[j] llavors
      trobat := cert;
    sino i := i + 1; j := j − 1;
    fsi
  fmentre
retorna no trobat
ffuncio

funcio separador(ent c: caracter): boolea
retorna (c = ' ') o (c = ',') o (c = ';')
ffuncio
```

4. De l'enunciat deduïm que podem organitzar els nivells de descomposició del disseny a partir dels objectes del problema. A l'entrada del programa tenim una seqüència de pàgines. Una pàgina és una seqüència de paraules separades per espais i controls de línia. Una paraula és una seqüència de caràcters sense espais ni controls de línia. La seqüència d'entrada és de tipus caràcter. Podem distingir, doncs, tres nivells de descomposició del disseny:
a) Seqüència de pàgines de web acabada amb "FiPagines".
b) Pàgina: seqüència de paraules acabada amb la paraula "FiTextHTML".
c) Paraula o mot: seqüència de caràcters acabat amb separador (espai, símbol =, caràcters de control de línia).

1) Primer nivell: seqüència de pàgines

Rellegim l'enunciat i l'adaptem seguint l'abstracció de pàgina.

Especificació

L'objectiu és confeccionar una llista de les empreses clients que participen en el servidor i, per cada empresa participant, obtenir la quantitat que se'ls ha de facturar. Per a cada pàgina, podem obtenir el nom de l'empresa client a la qual pertany i valorar-ne el preu a partir del contingut. En un primer nivell, formulem l'algorisme en termes d'una seqüència de pàgines. Així, doncs, l'entrada es pot veure com una seqüència de pàgines:

$$s = S = <P_1 \cdot P_2 \cdot P_n \cdot \text{'FiPagines'}>$$

'FiPagines' indica el final de la seqüència i es pot considerar com una pàgina especial i única que només té la paraula "FiPagines".

La precondició es pot expressar com:

{ Pre: $s = S = <P_1 \cdot P_2 \cdot ... P_n \cdot$ 'FiPagines'> **i** (tota P_i, $1 \le i \le n$, és una pàgina) }

La postcondició consistirà a tenir una seqüència t d'empreses clients amb l'import facturat. La seqüència t tindrà la forma següent:

$$t = <E_1 \cdot F_1 \cdot E_2 \cdot F_2 \cdot ... \cdot E_m \cdot F_m>$$

On E_i és el nom d'una empresa client present en la seqüència S, i F_i és l'import de la facturació que s'ha d'aplicar a l'empresa E_i. L'especificació de l'algorisme serà, doncs:

{ Pre: A l'entrada es té una seqüència $s = S = <P_1 \cdot P_2 \cdot ... P_n \cdot$ 'FiPagines'>
i (tota P_i, $1 \le i \le n$, és una pàgina) }
facturaPagines
{ Post: Crea la seqüència de sortida $t = <E_1 \cdot F_1 \cdot E_2 \cdot F_2 \cdot ... \cdot E_m \cdot F_m>$
i (si $E_i \neq$ 'NoIdentificats', E_i és un nom d'empresa client que hi ha a S)
i (F_i és l'import corresponent a l'empresa E_i de tots els mots
no gratuïts que hi ha a les pàgines de S que pertanyen a E_i)
i (totes les E_i presents a t són diferents)

i (si no hi cap cap E_i que sigui igual a "NoIdentificats",
m és el nombre d'empreses diferents de S) }
i (si existeix una E_i igual a "NoIdentificats",
$m - 1$ és el nombre d'empreses identificades diferents de S) }

Plantejament de l'algorisme principal

Identifiquem l'esquema a aplicar, i fem abstraccions noves.

La solució consistirà a fer que per cada pàgina que obtenim de la seqüència, identifiquem l'empresa client, en calculem l'import de la factura i ho acumulem en una estructura de dades que guarda les empreses clients identificades amb els imports facturats fins al moment. Es tracta, doncs, d'aplicar l'esquema de recorregut sobre la seqüència de pàgines. El tractament per cada pàgina consistirà a actualitzar una estructura de dades que manté la facturació de totes les empreses client que han aparegut o apareixeran més endavant en S a mesura que anem progressant. Anomenarem aquest objecte, *Factures*, i ja en veurem els detalls en el nivell del disseny corresponent.

Entenem que la darrera pàgina consistirà només en la paraula gratuïta 'FiPagines' i, per tant, no caldrà tractar-la.

Aplicació directa de l'algorisme

Particularitzem la solució de l'esquema de recorregut per a aquest problema.

Amb tot això, l'algorisme principal resulta de la manera següent:

```
algorisme facturaPagines;
   var
      factures: tFactures;
      pagina: tPagina;
   fvar
   inicialitzarFactures(Factures);
   obtenirPagina(Pagina);
   mentre no darreraPagina(Pagina) fer
      facturaPagina(Pagina, Factures);
      obtenirPagina(Pagina);
   fmentre
   escriureFactures(Factures);
falgorisme
```

Especificació de les accions

```
accio obtenirPagina(sor p: tPagina);
{ Pre: La part dreta de S conté una subseqüència que és una pàgina,
  com a mínim, o la pàgina especial "FiPagines".
  Anomenem pàgina E aquesta subseqüència. P representa la pàgina E }
{ Post: presenta la pàgina E4 }

funcio darreraPagina(p: tPagina): boolea
{ Pre: p = P }
{ Post: darreraPagina(p) és cert quan P representa la darrera pàgina }

accio inicialitzarFactures(sor F: tFactures)
{ Pre: }
{ Post: Hi ha 0 empreses propietàries a F }

accio facturaPagina(ent p: tPagina; entsor f: tFactures)
{ Pre: p = P i f = F i F conté l'import total per propietaris
  de pàgines processades anteriorment per facturaPagina }
{ Post: En f s'ha acumulat l'import de P corresponent al propietari de P.
  Si P no té propietari, l'import s'ha acumulat
  en el propietari especial "NoIdentificats" }

accio escriureFactures(ent f: tFactures)
{ Pre: f = F }
{ Post: Crea una seqüència de sortida amb l'estructura <n1i1 n2i2 ... nmim>,
  on ni és una subseqüència de caràcters que representa un propietari
  de pàgines F i acaba en blanc, i ii és una subseqüència
  de caràcters dígits acabada en blanc que representa
  l'import total que hi ha a F pel propietari ni }
```

2) Segon nivell: *Pàgines* i *Factures*

En aquest nivell tenim dues abstraccions: *pagina* i *factures*.

a) Passem primer a treballar l'abstracció *pagina*:

– Abstraccions utilitzades: mot.
– Identificació seqüència: pàgina vista com una seqüència de mots. L'obtenció dels elements primer i següent es farà mitjançant les accions *llegirPrimerMot* i *llegirMot*, i el reconeixement del mot que marca el final serà el mot "FiTextHTML". En el cas especial d'estar en la darrera pàgina, aquesta contindrà només "FiPagines".

Definició dels tipus

L'únic que interessa d'una pàgina és conèixer-ne el propietari i l'import de la factura. En el cas de la darrera pàgina, l'únic que interessa és saber que és la darrera. Definirem, doncs, el tipus *tPagina* de la manera següent:

```
tipus
tPagina = tupla
            propietari: tMot;
            import: enter;
            final: boolea;
          ftupla
ftipus
```

Plantejaments i solucions

Identificació de l'esquema: recorregut amb identificació prèvia de si es tracta de la darrera pàgina.

```
accio obtenirPagina(sor p: tPagina)
  var
    mot, motNoIdentificats,
    motFiPagines, motFiTextHTML, motPropietat: tMot;
  fvar
  inicialitzaMotNoIdentificats(motNoIdentificats);
  inicialitzaMotFiPagines(motFiPagines);
  inicialitzaMotFiTextHTML(motFiTextHTML);
  inicialitzaMotPropietat(motPropietat);
  p.Import := 0;
  p.Propietari := motNoIdentificats; { L'inicialitzem a "NoIdentificats"
                                       per si de cas no té propietari }
  llegirPrimerMot(mot);
  si motsIguals(mot, motFiPagines) llavors
    p.final := cert;
  sino
    p.final := fals;
    mentre no motsIguals(mot, motFiTextHTML) fer
      si motsIguals(mot, motPropietat) llavors
        llegirMot(mot);
        p.Propietari := mot;
      sino
        comptabilitzaMot(mot, p.Import);
        llegirMot(mot);
      fsi
    fmentre
  fsi
faccio { obtenirPagina }
```

Encara que en l'enunciat s'ha deixat clar que la seqüència d'entrada està ben formada, fixeu-vos que en la solució donada, en el cas pitjor en què després de "propietari" no hi hagués el nom del propietari i s'arribés al final de la seqüència, el propietari seria "FiTextHTML".

```
funcio darreraPagina(p: tPagina): boolea
retorna p.final;
ffuncio { darreraPagina }
```

b) Considerem ara la facturació d'empreses. Aquí l'objectiu és que l'objecte *Factures* contingui el nom de les empreses clients presents a la seqüència *S*, i per cada empresa client, l'import acumulat de les facturacions de totes les pàgines de *S* que pertanyen a l'empresa client en qüestió.

Definició del tipus

Cal mantenir una taula en què consti el nom de cada empresa client i el valor de l'import de la factura. L'enunciat ens assegura que no hi haurà més de 50 empreses diferents. Hem de tenir en compte que hi pot haver pàgines sense identificació de propietari que seran per a l'empresa fantasma "NoIdentificats". Així, doncs, tenim 51 empreses diferents.

Donat el propietari d'una pàgina, haurem de cercar en la taula si ja hi és o si s'hi ha d'afegir. La cerca que utilitzarem per a saber si una empresa client ja hi és, necessitarà una posició més de la taula, com veurem més endavant. Així, doncs, la declaració de tipus serà la següent:

```
const
   maxEmpreses: enter = 51;
fconst

tipus
   tFactura = tupla
                  nomClient: tMot;
                  import: enter;
              ftupla

   tFactures = tupla
                   nEmpreses: enter;
                   f: taula[ maxEmpreses + 1] de tFactura;
               ftupla
ftipus
```

Plantejament i solucions

Identificació de l'esquema: cerca.

```
accio facturaPagina(ent p: tPagina; entsor f: tFactures)
   var i: enter; fvar
   { Cerquem l'empresa per sentinella }
   f.f[F.nEmpreses + 1].nomClient := p.propietari;
   i := 1;
   mentre no motsIguals(f.f[i].nomClient, p.propietari) fer
      i := i + 1;
   fmentre
   si i = f.nEmpreses + 1 llavors
      { Donem d'alta l'empresa i inicialitzem l'import }
      f.nEmpreses := f.nEmpreses + 1;
      f.f[i].import := p.import;
   sino { Afegim el nou import }
      f.f[i].import := f.f[i].import + p.import;
   fsi
faccio { facturaPagina }
```

Fixeu-vos que...

... hem fet servir un esquema de cerca per sentinella; per aquest motiu hem necessitat una posició de més. El sentinella és sempre l'empresa que cerquem a la taula.

Aplicació directa

```
accio inicialitzarFactures(sor f: tFactures)
   f.nEmpreses := 0;
faccio
```

Identificació de l'esquema: recorregut.

```
accio escriureFactures(ent f: tFactures);
   var i: enter; fvar
   i := 1;
   mentre i ≤ f.nEmpreses fer
      escriuMot(f.f[i].NomClient);
      escriureEnter(f.f[i].Import);
      i := i + 1;
   fmentre
faccio { escriureFactures }
```

Accions pendents de desenvolupar

accio inicialitzaMotFiPagines(**sor** m: tMot)
{ Pre: }
{ Post: *m* és el mot "FiPagines" }

accio inicialitzaMotFiTextHTML(**sor** m: tMot)
{ Pre: }
{ Post: *m* és el mot "FiTextHTML" }

accio inicialitzaMotPropietat(**sor** m: tMot)
{ Pre: }
{ Post: *m* és el mot "propietat" }

accio inicialitzaMotNoIdentificats(**sor** m: tMot)
{ Pre: }
{ Post: *m* és el mot "NoIdentificats" }

accio llegirPrimerMot(**sor** m: tMot)
{ Pre: A l'entrada hi ha la seqüència de mots *s* de la frase *S*.
La part esquerra de *s* és buida. La part dreta, *DF*, no és buida }
{ Post: *m* representa el primer mot de *DF*.
El mot obtingut estarà a la part esquerra de la seqüència *s* }

accio obtenirMot(**entsor** m: tMot)
{ Pre: A l'entrada tenim la seqüència de mots *s* de la frase *S*.
La part dreta, *DF*, no és buida. El darrer mot llegit és a *m*. }
{ Post: *m* representa el primer mot de *DF*.
El mot obtingut estarà a la part esquerra de la seqüència *s* }

accio comptabilitzaMot(**ent** mot: tMot; **entsor** import: **enter**)
{ Pre: *mot* = *M* i *import* = *I* i *l* és el nombre de caràcters de *M* }
{ Post: *mot* = *M* i *import* = *I* + 50 si *M* és "<IMG",
o *import* = *I* + 5 si *M* no està afitada per "<" i ">" i *l* > 1
i *M* ≠ '<IMG',
o *import* = *I* si *M* està afitada per "<" i ">" o *l* ≤ 1 i *M* ≠ '<IMG' }

funcio motsIguals(m1: tMot; m2: tMot): **boolea**
{ Pre: *m1* = *M1* **i** *m2* = *M2* }
{ Post: *motsIguals*(*m1*,*m2*) és **cert**
quan *M1* i *M2* representen exactament el mateix mot }

accio escriuMot(**ent** m: tMot)
{ Pre: *t* = *T* i *t* és la seqüència de sortida i *m* = *M* }
{ Post: *t* és la subseqüència *T* seguida d'una subseqüència
que representa el mot *M* }

3) Tercer nivell: mot

– Abstraccions utilitzades: cap.
– Identificació de seqüència: mot vist com una seqüència de caràcters. L'obtenció dels elements primer i següent es farà mitjançant la funció predefinida *llegirCaracter*() i el reconeixement del caràcter que marca el final serà el caràcter espai, o l'igual, o el salt de línia.


Vegeu la solució de l'exercici anterior per als comentaris sobre el tipus.

Definició dels tipus

En aquest nivell treballem amb els tipus predefinits del llenguatge i amb el tipus *tMot* que definim a continuació:

```
const
   maxCar: enter = 40;
fconst

tipus
   tMot = tupla
             long: enter;
             c: taula[maxCar] de caracter;
             caracterLlegit: caracter;
          ftupla
ftipus
```

Plantejaments i solucions

```
accio inicialitzaMotFiPagines(sor m: tMot)
   m.long := 9;
   m.c[1] := 'F'; m.c[2] := 'i';
   m.c[3] := 'P'; m.c[4] := 'a'; m.c[5] := 'g'; m.c[6] := 'i';
   m.c[7] := 'n'; m.c[8] := 'e'; m.c[9] := 's';
faccio
```

```
accio inicialitzaMotFiTextHTML(sor m: tMot)
   m.long := 10;
   m.c[1] := 'F'; m.c[2] := 'i';
   m.c[3] := 'T'; m.c[4] := 'e'; m.c[5] := 'x'; m.c[6] := 't';
   m.c[7] := 'H'; m.c[8] := 'T'; m.c[9] := 'M'; m.c[10] := 'L';
faccio
```

```
accio inicialitzaMotPropietat(sor m: tMot)
   m.long := 9;
   m.c[1] := 'p'; m.c[2] := 'r'; m.c[3] := 'o';
   m.c[4] := 'p'; m.c[5] := 'i'; m.c[6] := 'e';
   m.c[7] := 't'; m.c[8] := 'a'; m.c[9] := 't';
faccio
```

```
accio inicialitzaMotNoIdentificats(sor m: tMot)
   m.long := 14;
   m.c[1] := 'N'; m.c[2] := 'o'; m.c[3] := 'I';
   m.c[4] := 'd'; m.c[5] := 'e'; m.c[6] := 'n';
   m.c[7] := 't'; m.c[8] := 'i'; m.c[9] := 'f';
   m.c[10] := 'i'; m.c[11] := 'c'; m.c[12] := 'a';
   m.c[13] := 't'; m.c[14] := 's';
faccio
```

Identificació de l'esquema: recorregut.

```
accio llegirPrimerMot(sor m: tMot)
   var c: caracter; fvar
   m.long := 0;
   c := llegirCaracter();
   mentre separador(c) i c ≠ '.' fer { saltem els primers separadors }
      c := llegirCaracter();
   fmentre
   m.caracterLlegit := c;
   obtenirMot(m);
faccio
```

Identificació de l'esquema: recorregut.

```
accio llegirMot(sor m: tMot)
   var
      i: enter; cNoSeparador: boolea;
      c: caracter;
   fvar
   m.long := 0;
   c := m.caracterLlegit;
   mentre no (separador(c) i (m.long < maxCar)) fer { obtenim el mot }
      m.long := m.long + 1;
      m.c[m.long] := c;
      c := llegirCaracter();
   fmentre
   mentre no separador(c) fer { saltem els caràcters que sobren }
      c := llegirCaracter();
   fmentre
   mentre separador(c) fer { saltem els separadors }
      c := llegirCaracter();
   fmentre
   m.caracterLlegit := c;
faccio
```

Observeu que...

... l'acció *llegirMot* té en compte la possibilitat que la paraula llegida sigui més llarga que *maxCar*. La paraula es llegeix, però els caràcters del mot que són més enllà de *maxCar* a la taula no es guarden. Com que només ens interessa comprovar els mots que tenen *maxCar* caràcters, com a molt, el truncament de la paraula no té cap efecte en la resta de l'algorisme i permet d'acceptar paraules tan llargues com es vulgui.

Aplicació directa de l'alternativa

```
accio comptabilitzaMot(mot: tMot; entsor import: enter)
   var motImatge: tMot; fvar
   posaMotImatge(motImatge);
   si motsIguals(mot, motImatge) llavors
      import := import + 50;
   sino
      si no(motEntreMenorsMajors(Mot) i longitutMot(Mot) > 1) llavors
         import := import + 5;
      fsi
   fsi
faccio
```

Identificació de l'esquema: cerca.

```
funcio motsIguals(ent m1: tMot; ent m2: tMot): boolea
   var
      i: enter;
      iguals: boolea;
   fvar
   si m1.long = m2.long llavors
      si m1.l > 0 llavors
         i:= 1;
         mentre (m1.c[i] = m2.c[i]) i (i < m1.l) fer i := i + 1; fmentre
         iguals := m1.c[i] = m2.c[i];
      sino iguals := cert;
      fsi
   sino iguals := fals;
   fsi
retorna iguals;
ffuncio { motsIguals }
```

Identificació esquema: recorregut.

```
accio escriuMot(ent m: tMot)
   var i : enter; fvar
   i := 1;
   mentre i ≤ m.long fer
      escriureCaracter(m.c[i]);
      i := i + 1;
   fmentre;
   escriureCaracter(' ');
faccio { escriuMot }
```

Accions pendents de desenvolupar

El disseny d'aquest nivell ha donat peu a unes funcions auxiliars que detallem a continuació.

```
accio posaMotImatge(sor m: tMot)
{ Pre: }
{ Post: m és el mot <IMG }

funcio motsEntreMenorsMajors(ent m: tMot): boolea
{ Pre: m = M }
{ Post: motsEntreMenorsMajors és cert quan el primer caràcter del mot és < i el darrer és > }

funcio longitutMot(ent m: tMot): enter
{ Pre: m = M }
{ Post: longitutMot(m) dóna el nombre de caràcters que té M }

funcio separador(ent c: caracter): boolea
{ Pre: c = C }
{ Post: separador(c) és cert si C és espai o C és un igual
  o C és algun caràcter de retorn o de salt de línia }

accio posaMotImatge(sor m: tMot)
   m.long := 4;
   m.c[1] := '<'; m.c[2] := 'I'; m.c[3] := 'M'; m.c[4] := 'G';
faccio
```

```
funcio motEntreMenorsMajors(ent m: tMot): boolea
retorna m.c[1] = '<' i m.c[m.long] = '>';
ffuncio

funcio longitutMot(ent m: tMot): enter
retorna m.long;
ffuncio

{ c = C }
funcio separador(ent c: caracter): boolea
retorna c = ' ' o c = '=' o c = codiACaracter(10) o c = codiACaracter(13);
ffuncio
```

La funció *separador*...

... haurà de tenir en compte els convenis seguits per als caràcters de control de retorn de línia i de salt de línia. Aquests caràcters en ASCII són els codis 13 i 10.

Glossari

abstracció
Simplificació d'una realitat tenint en compte només una part d'aquesta, considerant els atributs i qualitats rellevants per a la tasca que s'ha de dur a terme.

complex
No simple. Dit de tot allò que comprèn parts o elements discernibles.

descomposició d'un problema
Separació i identificació de subproblemes més elementals que el problema original.

disseny descendent
Metodologia que descompon un problema complex en subproblemes menys complexos que l'original. La descomposició és guiada per abstraccions naturals i convenients al problema, i dóna com a resultat un conjunt de subproblemes independents i més concrets. Es procedeix de manera idèntica per a cadascun dels subproblemes fins que s'arriba a subproblemes prou elementals.

nivell d'abstracció
Nivell en què es concreta una abstracció utilitzada per a descompondre un problema o subproblema. *Nivell d'abstracció* i *nivell de descomposició* són dos conceptes diferents que la majoria de cops coincideixen, però no ha de ser necessàriament així.

nivell de descomposició
Nivell que correspon a una etapa de la descomposició de subproblemes en subproblemes més concrets. El primer nivell correspon al desenvolupament del problema original. La descomposició dels subproblemes generats en el primer nivell s'han de desenvolupar en un segon nivell, etc.

Bibliografia

Castro, J.; Cucker, F.; Messeguer, X.; Rubio, A.; Solano, L.; Vallés, B. (1992). *Curs de programació.* Madrid, etc.: McGraw-Hill.

Sebastià Vila, M. (1995). *Programació fonamental. Problemes.* Barcelona: Edicions UPC.

Wirth, N. (1982). *Introducción a la programación sistemática.* Buenos Aires: El Ateneo.